

Thinking in Java

Bruce Eckel
President, MindView Inc.

Prentice Hall PTR
Upper Saddle River, New Jersey 07458
<http://www.phptr.com>

Library of Congress Cataloging-in-Publication Data

Eckel, Bruce.

Thinking in Java / Bruce Eckel.

p. cm.

Includes index.

ISBN 0-13-659723-8

1. Java (Computer program language) I. Title.

QA76.73.J38E25 1998

005.13'3--dc21

97-52713

CIP

Editorial/Production Supervision: Craig Little

Acquisitions Editor: Jeffrey Pepper

Manufacturing Manager: Alexis R. Heydt

Marketing Manager: Miles Williams

Cover Design: Daniel Will-Harris

Interior Design: Daniel Will-Harris, www.will-harris.com

© 1998 by Prentice Hall PTR

Prentice-Hall Inc.

A Simon & Schuster Company

Upper Saddle River, NJ 07458

The information in this book is distributed on an "as is" basis, without warranty. While every precaution has been taken in the preparation of this book, neither the author nor the publisher shall have any liability to any person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by instructions contained in this book or by the computer software or hardware products described herein.

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Prentice Hall books are widely used by corporations and government agencies for training, marketing, and resale. The publisher offers discounts on this book when ordered in bulk quantities. For more information, contact the Corporate Sales Department at 800-382-3419, fax: 201-236-7141, email: corpsales@prenhall.com or write: Corporate Sales Department, Prentice Hall PTR, One Lake Street, Upper Saddle River, New Jersey 07458.

Java is a registered trademark of Sun Microsystems, Inc. Windows 95 and Windows NT are trademarks of Microsoft Corporation. All other product names and company names mentioned herein are the property of their respective owners.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-659723-8

Prentice-Hall International (UK) Limited, London

Prentice-Hall of Australia Pty. Limited, Sydney

Prentice-Hall Canada Inc., Toronto

Prentice-Hall Hispanoamericana, S.A., Mexico

Prentice-Hall of India Private Limited, New Delhi

Prentice-Hall of Japan, Inc., Tokyo

Simon & Schuster Asia Pte. Ltd., Singapore

Editora Prentice-Hall do Brasil, Ltda., Rio de Janeiro

Превод на български език, с любезното разрешение на автора, © Спас Маринов Касабов, 1999, 2000

Какво има вътре

Thinking in Java 2nd edition	
Revision 3.....	1
Второ издание.....	2
Bruce Eckel.....	2
Читателски отзиви:.....	3
Bruce Eckel's Hands-On Java Seminar Multimedia CD	
It's like coming to the seminar!	
Available at http://www.BruceEckel.com	10
На личността която, даже сега, създава следващия голям език за програмиране.....	11
Какво има вътре.....	12
Предговор 1	
Предговор към 2ро издание.....	4
Въведение 6	
Необходими начални знания.....	6
Учене Java.....	7
Цели.....	8
Онлайн документация.....	9
Глави.....	9
Упражнения.....	15
Мултимедиен CD ROM.....	15
Сорс.....	16
Стандарти при кодирането.....	17
Версии на Java.....	17
Семинари и наставничество.....	18
Грешки.....	18
Бележка по дизайна на корицата.....	19
Благодарности.....	20
За Интернет.....	22
1: Въведение в обектите 23	
Абстракцията.....	24
Всеки обект си има интерфейс.....	26
Скриване на код.....	27
Повторно използване на код.....	29
Наследяване:	
повторно използване на интерфейса.....	30
Подтискане на функциите на базовия клас.....	31
Отношенията "е" и "прилича на".....	31
Взаимозаменяеми обекти с полиморфизъм.....	32
Динамично свързване.....	33
Абстрактни базови класове и интерфейси.....	34

Обекти и техните времена на живот.....	35
Колекции и итератори.....	36
Йерархия с един корен.....	38
Библиотеки от колекции и поддръжка за лесно използване.....	39
Downcasting и templates/generics.....	89
Домакинската дилема:	
кой ще чисти?.....	40
Събирачите на боклук срещу ефективността и гъвкавостта	41
Обработка на изключенията:	
работка с грешките.....	42
Многонишково изпълнение	43
Упоритост.....	44
Java и Internet.....	45
Какво е Мрежата?	45
Клиент/сървър обработки..	45
Мрежата е огромен сървър	46
Програмиране на клиентската страна.....	
Plug-ins.....	48
Скриптове и езици за тях....	49
Java.....	50
ActiveX.....	51
Сигурност.....	51
Internet срещу Intranet.....	52
Програмиране при сървъра. .53	
Отделна аrena: приложенията	54
Анализ и проектиране.....	55
Стоим на курса.....	55
Фаза 0: Да направим план.....	56
Фаза 1: Какво ще произвеждаме?	57
Фаза 2: Как ще го направим?.58	
Фаза 3: Да строим!	59
Фаза 4: Итерация.....	59
Планирането се изплаща.....	61
Java или C++?.....	61
2: Всичко е обект	64
Обектите манипулираме с манипулятори	64
Трябва вие да създадете всичките обекти.....	65
Къде живее паметта.....	65
Специален случай: първични типове.....	67
Числа с висока точност.....	69
Масиви в Java.....	69
Никога не се налага да разрушавате обект.....	70
Обхвати.....	70

Обхват на обекти.....	71
Създаване на нови типове данни: class.....	72
Полета и методи.....	72
Стойности по подразбиране за примитивни членове.....	73
 Методи, аргументи и връщани стойности.....	74
Списъкът аргументи.....	75
Построяване на Java програма.....	76
Видимост на имената.....	76
Използване на други компоненти.....	77
Ключовата дума static	78
Вашата първа Java програма.....	79
Коментари и вградена документация.....	82
Коментар на документацията.	83
Синтаксис.....	83
Вграден HTML.....	84
@see: отнасяне към други класове.....	85
Директива за документиране на клас.....	85
@version.....	85
 @author.....	85
Директиви за документиране на променлива.....	86
Директиви за документация на метод.....	86
@param.....	86
 @return.....	86
@exception.....	86
 @deprecated.....	87
Пример за документация.....	87
Стил на кодиране.....	88
Резюме.....	88
Упражнения.....	88
 3: Управление хода на програмата	90
Използваме Java оператори.....	90
Приоритет.....	91
Присвояване.....	91
Aliasing при извикване на методи.....	93
Математически оператори....	94
Унарни минус и плюс оператори.....	95
Авто инкремент и декремент. .	96
Оператори за отношение.....	97
Проверка на еквивалентност на обекти.....	97
Логически оператори.....	98
Феноменът short-circuiting	100
Побитови оператори.....	101
Shift оператори.....	101
Триместен if-else оператор...	105
Операторът запетая.....	105

String операторът +.....	106
Обичайни капани при използването на операторите.....	106
Casting оператори.....	107
Литерали.....	108
 Разширяване.....	109
Java няма "sizeof"	110
Приоритетът по нов начин.....	110
Резюме на операторите.....	111
Управление на изпълнението.....	119
true и false.....	119
if-else.....	119
return.....	120
 Итерация	120
do-while.....	121
for.....	121
Операторът запетая.....	122
 break и continue.....	123
Безславното "goto".....	124
 switch.....	128
Детайли на изчислението	129
 Резюме.....	131
Упражнения.....	131

4: ИНИЦИАЛИЗАЦИЯ И ПОЧИСТВАНЕ

133

Гарантирана инициализация с конструктора.....	134
Пренатоварване на методи	135
Различаване на претоварените методи.....	138
Претоварването и примитивите	138
Претоварване на връщаните стойности.....	141
Конструктори по подразбиране	142
Ключовата дума this.....	143
Викане на конструктори от конструктори.....	144
 Значението на static.....	145
Почестване: финализация и събиране на боклука.....	146
За какво е finalize()?.....	147
Вие трябва да почистите.....	148
Как работи събирачът на боклук.....	151
Инициализация на членове	154
Задаване на инициализация	156
Инициализация в конструкторите.....	157
Ред на инициализация.....	157
 Инициализация на статични данни.....	158
Явна инициализация на static	161
Инициализация на нестатични екземпляри.....	162
Инициализация на масиви	163

Многомерни масиви.....	167
Резюме.....	169
Упражнения.....	170

5: Скриване на реализацията	171
package: библиотечната единица.....	172
Създаване на уникални имена на пакети.....	174
Автоматична компилация..	177
Колизии.....	177
Библиотека собствени инструменти.....	178
Капанът с classpath.....	179
Използване на импортиране за промяна на поведението.....	180
Относно пакетите.....	182
Спецификатори на достъпа в Java.....	182
“Приятелски”.....	182
public: интерфейсен достъп..	183
Пакетът по подразбиране	184
private: не може да пипате това!	185
protected: “вид приятелски” .	186
Интерфейс и реализация	187
Достъп до клас.....	188
Резюме.....	191
Упражнения.....	193
6: Многократно използване на класове	194
Синтаксис на композицията	195
Синтаксис за наследяването.....	198
Инициализиране на базовия клас.....	200
Конструктори с аргументи	201
Хващане на изключениета на базовия конструктор.....	202
Комбиниране на композиция и наследяване.....	202
Гарантиране на провилно почистване.....	203
Ред при почистването на боклука.....	206
Скриване на имената.....	206
Избиране на композиция	
vs. наследяване.....	207
protected.....	209
Постъпкова разработка.	209
Upcasting.....	210
Защо “upcasting”?.....	211
Отново КОМПОЗИЦИЯ vs. НАСЛЕДЯВАНЕ.....	212
Ключовата дума final	212
Final данни.....	212
Празни final.....	215
Final аргументи.....	215
Final методи.....	216

Final класове.....	217
Final предпазливост.....	218
Инициализация и товарене на класовете.....	218
Инициализация с наследяване	219
Резюме.....	221
Упражнения.....	221

7: Полиморфизъм 223

Upcasting.....	224
Защо ъпкастинг?.....	225
Особеността.....	226
Свързване при извикването на метод.....	226
Постигане на точното поведение.....	227
Разширяемост.....	230
Подтискане vs. претоварване.....	233
Абстрактни класове и методи.....	234
Интерфейси.....	237
“Многоократно наследяване” в Java.....	240
Разширяване на интерфейс с наследяване.....	242
Групиране на константи.....	243
Инициализиране на полета в интерфейсите.....	245
Вътрешни класове.....	246
Вътрешни класове и ъпкастинг	247
Вътрешни класове в методи и обхвати.....	249
Връзката към външния клас...	254
static вътрешни класове.....	256
Обръщания към обект от външния клас.....	258
Наследяване от вътрешни класове.....	259
Могат ли вътрешни класове да се подтискат?	260
Идентификатори на вътрешния клас.....	262
Защо вътрешни класове: рамки на управлението.....	262
Конструктори и полиморфизъм.....	269
Ред на извикване на конструкторите.....	269
Наследяването и finalize().....	271
Поведение на полиморфни методи в конструкторите.....	274
Проектиране с наследяване.....	276
Чисто наследяване vs. разширяване.....	278
Даункастинг и идентификация на типа по време на изпълнение.....	280
Резюме.....	282
Упражнения.....	283

8: Притежаване на обектите 284

Масиви.....	284
Масивите са първокласни обекти.....	286
Колекции от примитиви.....	289
Връщане на масив.....	289
Колекции.....	290
Неудобство: неизвестен тип.	291
Понякога работи правилно при всички случаи.....	293
Правене на Vector чувствителен към типа.....	294
Параметризиранi типове	295

Итератори.....	296
Типове колекции.....	299
ArrayList.....	299
Сkapване на Java.....	299
 BitSet.....	300
Stack.....	302
Map.....	303
Създаване на “ключови” класове.....	306
 Properties: тип HashMap....	309
Пак енумератори.....	309
Сортиране.....	310
Колекциите на Java 2	315
Използване на Collectionи....	319
Използване на Listове.....	323
Използване на Setове.....	326
Използване на Mapове.....	329
Избиране на реализация.....	331
Избор между Listове.....	332
 Избор между Setовете.....	334
Избор между Mapове.....	336
 Неподдържани операции....	339
Сортиране и търсене.....	341
Arrays.....	342
 Comparable и Comparator	343
Listове.....	345
Ютилита.....	346
Правене на Collection или Map неизменяеми.....	347
 Синхронизиране на Collection или Map.....	348
Резюме.....	349
Упражнения.....	350
 9: Обработка на грешки чрез изключения	352
Основни изключения.....	354
Аргументи на изключението..	355
Хващане на изключение	355
Блокът try	356
Обработчици на изключения	356
Прекратяване vs. продължаване.....	357
 Специфициране на изключението.....	358
Хващане на кое да е изключение.....	359
Преизхвърляне на изключение	360
Стандартни изключения в Java	363
Специалният случай RuntimeException.....	364
Създаване на ваши собствени изключения.....	366
Ограничения при изключениета.....	369

Финализиране с finally....	372
За какво е finally?.....	373
Капан: загубеното изключение	375
Конструктори.....	376
Търсене на обработчик .	380
Правила за изключенията.....	381
Резюме.....	381
Упражнения.....	382

10: Входно-изходна система на Java

383

Вход и изход.....	384
Типове на InputStream.....	384
Типове OutputStream.....	387
Добавяне на атрибути и полезни интерфейси.....	391
Четене от InputStream	
с FilterInputStream.....	392
Писане в OutputStream	
с FilterOutputStream.....	394
Отделен:	
RandomAccessFile.....	396
Класът File.....	397
A directory lister.....	397
Аnonимни вътрешни класове	399
Сортиран списък на директорията.....	401
Проверка за и създаване на директории.....	402
Типично използване на IO потоци.....	404
Входни потоци.....	406
1. Буфериран вход от файл	406
2. Вход от паметта.....	407
3. Форматиран вход от паметта.....	407
4. Номериране на редове и файлов изход.....	408
Изходни потоци.....	408
5. Запазване и възстановяване на данни.....	409
6. Четене и писане на файлове с произволен достъп.....	409
Кратък начин за операции с файлове.....	410
7. Кратък начин за вход от файл.....	410
8. Кратък път за форматирано писане във файл.....	410
9. Начин за писане в даннов файл.....	411
Четене от стандартния вход .	411
Скачени потоци.....	412
StreamTokenizer.....	412
StringTokenizer.....	415
IO потоци на Java 1.1	417
Източници и стоци на данни.	418
Променяне на поведението на потоците.....	419

Непроменени класове.....	420
Пример.....	421
Генератор на справки.....	424
Пренасочване на стандартния IO.....	425
Компресия.....	426
Проста компресия с GZIP.....	427
Много файлове със Zip.....	429
Java archive (jar) утилитито....	431
Сериализация на обекти	433
Намиране на класа.....	437
Управление на сериализацията	438
Ключовата дума transient	442
Алтернатива на Externalizable.....	443
Промяна на версиите.....	446
Използване на устойчивоста	446
Проверка на стила.....	452
Резюме.....	460
Упражнения.....	461
11: Идентификация на типа по време на изпълнение	463
Нуждата от RTTI.....	463
Обектът Class.....	466
Класни литерали.....	468
Проверка преди каст.....	469
Използване на класни литерали.....	471
Динамично instanceof.....	473
Синтаксис на RTTI	474
Рефлексия: информация за клас по време на изпълнение.....	476
Екстрактор на методите.....	478
Резюме.....	482
Упражнения.....	483
12: Подаване и връщане на обекти	484
Подаване на манипулатори	485
Псевдоними.....	485
Създаване на локални копия.....	487
Предаване по стойност.....	488
Клониране на обекти.....	489
Правене на клас клонираме.	490
Използване на трик с protected.....	490
Прилагане на интерфейса Cloneable.....	491
Успешно клониране.....	492
Ефектът от Object.clone().....	494
Клониране на композиран обект.....	496
Дълбоко копиране на ArrayList	497
Дълбоко копиране чрез сериализация.....	499
Добавяне на клонируемост надолу по йерархията.....	501
Защо този странен дизайн?	502

Управление на клонирането.....	502
Копи-конструкторът.....	507
Защо работи в C++ и не работи в Java?.....	510
Класове само за четене.....	511
Създаване на класове само за четене.....	512
Недостатъкът на непроменимостта.....	513
Неизменяеми Stringове.....	515
Неявни константи.....	515
Претоварване на '+' и StringBuffer.....	516
Класовете String и StringBuffer.....	517
Stringовете са специални.....	524
Резюме.....	524
Упражнения.....	526
13: Създаване на прозорци и аплети	527
Основният аплет.....	529
Пускане на аплети от Web браузър.....	530
Автоматична генерация на HTML файлове.....	532
Използване на Appletviewer.....	532
Тестване на аплети.....	533
Правене на бутона.....	534
Хващане на събитие.....	535
Текстови полета.....	538
Текстови площи.....	539
Етикети.....	540
Отметки.....	542
Радиобутони.....	543
Падащи списъци.....	544
Списъчни кутии.....	546
Пана с ушички.....	547
Кутии за съобщения.....	548
Менюта.....	549
Диалогови кутии.....	550
Управление на разположението.....	551
FlowLayout.....	551
BorderLayout.....	552
GridLayout.....	553
GridBagLayout.....	553
BoxLayout.....	554
Алтернативи на action....	557
Ограничения при аплетите	562
Предимства на аплетите.....	563
Приложения с прозорци	564
Комбинирано приближение/аплет.....	565
Менюта.....	566
Диалогови кутии.....	570
Файлови диалози.....	573
Моделът на събитията.....	575
Типове събития и слушатели.	577
Използване на слушателски адаптери за простота.....	581

Правене на прозорци и аплети.....	582
Правене на анонимен клас слушател на прозорец.....	584
Пакетиране на аплет в JAR файл.....	585
Преразглеждане на по-раншните примери.....	585
Демонстриране методите на рамката.....	586
Текстови полета.....	587
Текстови полета.....	589
Отметки и радиобутони.....	591
Падащи списъци.....	593
Списъци.....	594
Менюта.....	596
Диалогови кутии.....	600
Избиране на изглед и усещане.....	602
Свързване на събитията динамично.....	604
Отделяне на основната логика от UI логиката.....	606
Препоръчвани подходи при кодирането.....	609
Чертата: добрият начин да го правите.....	609
Реализиране на главния клас като слушател.....	610
Смесване на подходите.....	612
Наследяване на компонент.....	613
Грозен интерфейс на компонент.....	618
JFC APIта.....	623
Цветовете.....	623
Печатане.....	623
Печатане на текст.....	628
Печатане на графика.....	628
Стартиране на Frameове в аплети.....	629
Джобът.....	630
Теглене и Пускане.....	633
Визуално програмиране и Bean-ове.....	638
Какво е Bean?.....	640
Извличане на BeanInfo	
с Introspector.....	642
По-усложнен Bean.....	647
Пакетиране на Bean.....	650
По-сложна поддръжка за Bean	652
Повече за Beans.....	653
Въведение в Swing.....	653
Ползите от Swing.....	654
Лесно преобразуване.....	654
Работна рамка за дисплей.....	655
Хвърчащи подсказки.....	656
Ограничителни линии.....	657
Бутони.....	658
Групи бутони.....	659
Икони.....	660
Менюта.....	661
Изскучащи менюта.....	665
Списъчни кутии и комбинирани кутии.....	666
Пълзгачи и ленти за напредъка	667
Дървета.....	668
Таблици.....	670

Панели с ушички.....	672
Още Swing.....	673
Използване на URLobe от аплет.....	674
Четене на файл от сървър.....	675
Инструмент за оглед на методите.....	676
Резюме.....	680
Упражнения.....	681

14: Много нишки 683

Отговарящ потребителски интерфейс.....	684
Наследяване от Thread.....	686
Нишковост за отговарящ интерфейс.....	688
Подобряване на кода с вътрешен клас.....	690
Комбиниране на нишката с главния клас.....	692
Правене на много нишки.....	694
Нишки-демони.....	697
Споделяне на ограничени ресурси.....	698
Неправилно ползване на ресурси.....	699
Как Java споделя ресурси... 703	
Синхронизиране на броячите.....	704
Синхронизирана ефективност.....	707
Java Beans разгледани пак. . 707	
Блокиране	711
Как се блокира нишка.....	712
Спане.....	714
Спираче и продължаване 715	
Чакай и бъди уведомен....	716
Блокиране по IO.....	718
Тестване.....	719
Мъртъв блокаж.....	721
Остарялост на stop(), suspend(), resume() и destroy() в Java 2721	
Приоритети.....	725
Групи нишки.....	729
Управление на групите нишки.....	731
Runnable отново.....	735
Твърде много нишки.....	738
Резюме.....	740
Упражнения.....	742

15: Разпределена обработка

744

Идентификация на машина 745	
Сървъри и клиенти.....	746
Тестване на програми без мрежа.....	747
Port: уникално място в машината.....	747
Socket-и.....	748

Прости сървър и клиент.....	749
Обслужване на много клиенти.....	754
Датаграми.....	758
RMI (Remote Method Invocation).....	764
Отдалечени интерфейси.....	765
Реализация на отдалечен интерфейс.....	766
Установяване на регистъра.....	767
 Създаване на стубове и скелетони.....	769
Използване на отдалечения обект.....	770
Въведение в CORBA.....	770
Основи на CORBA.....	771
Interface Definition Language (IDL) на CORBA.....	772
 Услуга за имената.....	773
Пример.....	773
Написване на IDL сорса ..	773
 Създаване на стъбове и скелетони.....	774
Реализация на сървъра и клиента.....	774
Някои услуги на CORBA....	775
Активиране на процеса на службата за имената.....	777
Активиране на сървъра и клиента.....	777
 Java аплети и CORBA.....	778
CORBA vs. RMI.....	778
Jini: разпределени услуги.....	779
Jini в контекст.....	779
Какво е Jini?.....	780
Как работи Jini.....	781
Процесът discovery.....	781
Процесът на присъединяване.....	782
Процесът на преглеждане....	782
Разделяне на интерфейса и реализацията.....	783
Абстракция на разпределени системи.....	784
JavaSpaces.....	785
Резюме.....	785
Упражнения.....	785
 16: Програмиране за фирмата	787
Java Database Connectivity (JDBC).....	788
Каране примера да работи.	791
Стъпка 1: Намиране JDBC драйвера.....	791
Стъпка 2: Конфигуриране на базата данни.....	791
Стъпка 3: Проба на конфигурацията.....	792
Стъпка 4: Генерация на SQL поръчка.....	793
Стъпка 5: Променете и вмъкнете запитването си.....	793
GUI версия на преглеждащата програма.....	794

Защо API-то на JDBC изглежда така сложно.....	797
Сървлети.....	797
Основния сървлет.....	798
Java Server Pages.....	799
Основни операции.....	800
Вграждане на блокове код... Извличане на полета и стойности.....	800
Манипулиране на сесията в JSP802	
Създаване и промяна на Cookies.....	802
Предаване управлението на други сървлети.....	803
RMI във фирмата.....	804
Corba във фирмата.....	804
Enterprise Java Beans (EJB)804	
Защо ни трябва EJB?.....	805
Как да сложа 'Е'-то на съществуващите мои JavaBeans?.....	806
Компоненти в Enterprise Java Beans.....	807
EJB Сървър	807
EJB Container.....	807
Enterprise JavaBeans.....	807
Session Beans.....	808
Entity Beans.....	808
Bean Managed Persistence808	
Container Managed Persistence.....	808
Bean Managed vs. Container Managed Persistence.....	809
Описатели на разгръщането811	
JNDI.....	812
JTA/JTS.....	812
CORBA.....	812
Ролите на Enterprise Java Beans812	
Доставчик на Enterprise Bean-овете.....	813
Монтажник на приложениета813	
Deployer.....	813
EJB Server/Container доставчик	813
Системен администратор 813	
Разработване с Enterprise Java Beans.....	814
Разработка на прост Stateless Session Bean.....	814
Разработка на прост Stateful Session Bean.....	814
Разработка на BMP Entity Bean.....	814

Разработка на CMP Entity Bean.....	814
------------------------------------	-----

Дискусионни въпроси, интересни проблеми и литература.....	814
Защо толкова много правила?.....	814

Какво не е дефинирано в EJB спецификацията.....	815
---	-----

Други позовавания.....	815
------------------------	-----

Достъп до цялостни услуги	815
---------------------------	-----

Java Naming and Directory Interface (JNDI).....	816
---	-----

Java Messaging Service (JMS)	816
------------------------------	-----

JavaMail.....	816
---------------	-----

Сигурност във фирмата ..	816
--------------------------	-----

Резюме.....	816
-------------	-----

Упражнения.....	816
-----------------	-----

17: Шаблони за проектиране

817

Концепцията за шаблон.	817
------------------------	-----

Синглетонът.....	819
------------------	-----

Класификация на шаблоните	820
---------------------------	-----

Фабрики: капсулиране създаването на обекти.....	821
---	-----

Полиморфни фабрики.....	823
-------------------------	-----

Абстрактни фабрики.....	826
-------------------------	-----

Шаблонът "наблюдател".	828
------------------------	-----

Симулиране на рециклиатор	831
---------------------------	-----

Подобряване на дизайна	834
------------------------	-----

"Прави още обекти".....	834
-------------------------	-----

Шаблон за създаване на прототипи.....	837
---------------------------------------	-----

Trash подкласове.....	841
-----------------------	-----

Информация за Trash от външен файл.....	842
---	-----

Рециклиране с прототипиране.....	844
----------------------------------	-----

Абстракция на използването.....	845
---------------------------------	-----

Многократно диспечиране	849
-------------------------	-----

Реализация на двойното диспечиране.....	850
---	-----

Шаблонът "посетител" ...	855
--------------------------	-----

Повече сдържаване?.....	861
-------------------------	-----

RTTI считана вредна?.....	862
---------------------------	-----

Резюме.....	864
-------------	-----

Упражнения.....	865
-----------------	-----

A: Java Native Interface (JNI)

867

Java Native Interface.....	867
----------------------------	-----

Викане на нативен метод.....	868
------------------------------	-----

Генератор на С хедъри: javah.....	869
-----------------------------------	-----

Сигнатури на функциите; имената.....	869
--------------------------------------	-----

Прилагане на ваша DLL....	870
---------------------------	-----

Достъп до JNI функции:	
------------------------	--

аргументът JNIEnv.....	871
------------------------	-----

Достап до Java стрингове 872

Подаване и използване на Java обекти.....	872
JNI и изключениета в Java.....	874
JNI и тредингът.....	875
Използване на съществуващ код.....	875
Пътят на Microsoft.....	875

В: Насоки за програмиране на Java	877
--	------------

С: Препоръчва се за четене	882
-----------------------------------	------------

Индекс	884
---------------	------------

Bruce Eckel's Hands-On Java Seminar Multimedia CD

It's like coming to the seminar!

Available at <http://www.BruceEckel.com>.....894

Предговор

Казах на моя брат Тод, който минава от хардуер към програмиране, че следващата голяма революция ще бъде в генното инженерство.

Ще има микроорганизми проектирани да произвеждат храна, гориво или пластмаси; те ще прочистят замърсяването и изобщо ще позволят манипулирането в реалния свят за част от днешната цена. Аз претендирах, че революцията в генното инженерство ще затъни компютърната революция.

После разбрах, че правя грешка обща за писателите-научни фантасти: загубвайки се в технологията (което е лесно в научната фантастика разбира се). Опитният писател знае че историята никога не е за вещите; тя е за хората. Генетиката ще има много голямо отражение върху нашия живот, но аз не съм много сигурен че ще омоловажи компютърната революция – или най-малкото информационната революция. Информацията е в говоренето един на друг: да, колите и обущата и особено генетично създадените лекарства са много важни, но в последна сметка са само салтанати. Истинско значение има само нашият начин за отнасяне към света. А толкова много от него е комуникация.

Тази книга е точно такъв случай. Повечето хора мислеха, че съм много щедър или малко луд да слагам цялата книга в Мрежата. „Защо да я купуват тогава?“ питаха те. Ако бях по-консервативен нямаше да го направя, но аз наистина не искам да правя книги по стария начин. Не знаех какво ще стане, но това се оказа най-умното нещо, което някога съм правил с книга.

Хората взеха да пращат корекции. Това беше забележителен процес, защото те бяха надникнали във всеки ъгъл и бяха хванали както технически, така и стилни грешки и аз бях в състояние да поправя много грешки, които иначе биха останали незабелязани. Хората бяха просто ужасни с това нещо, често казвайки „Сега, не го приемай като критика“ и ми даваха след това колекция от грешки, които не бих открил иначе, сигурен съм. Чувствам, че това беше вид групова обработка и това правеше книгата специална.

След това взех да чувам „Добре, че сте я сложили в Мрежата, но аз искам реално отпечатана и подвързана книга, от истински издател.“ Аз много се трудих да я направя печатуема от всеки в подходящ формат, но това не намали поръчките за книги в печатницата. Повечето хора не обичат да четат цялата книга от екран и да им се моткат спонове хартии, без значение колко хубаво отпечатани (плус това не толкова евтини в смисъл на тонер за лазерен принтер, мисля). Изглежда, че компютърната революция няма да изтласка печатарите от бизнеса, най-после. Обаче един студент вметна, че за в бъдеще ще трябва книгите да се публикуват в Мрежата първо, а после да се печатат, ако има достатъчен интерес. В момента по-голямата част от всички видове книги са финансов провал и може би новият начин ще направи тази индустрия по-печеливша.

Тази книга ме просвети и по друг начин. Отначало смятах Java за „просто още един език за програмиране,“ какъвто в много отношения той е. Но когато с течение на времето го проучих по-дълбоко, аз започнах да разбирам, че основното намерение е различно от това на всички езици, които някога бях виждал.

Програмирането е на път да се справи със сложността: сложността на проблема, който решавате лежи върху сложността на машината, на която го решавате. Поради тази сложност повечето от програмните проекти не успяват. Все още никой от програмните езици които съм чувал не си поставя за основна цел да се справи със сложността. Разбира се, много от

главните решения са вземани като е имана предвид сложността, но в някоя точка винаги са се появявали съображения, които са ставали основни. Обратно, тези "други съображения" са довеждали до това програмистите да разбиват стена с глава. Например, C++ трябваше да бъде ефикасен и обратно съвместим със C (за да позволи лесна миграция на C програмисти). Това бяха много полезни цели и на тях се дължи немалка част от успеха на C++, но те също доведоха до повищена сложност и доведоха до това някои проекти да не могат да бъдат завършени (разбира се, би могло да се обвиняват програмистите и управлението на проектите, но ако езикът може да помогне срещу грешките с автоматичното им прихващане, защо да не го прави?). Като друг пример, Visual Basic (VB) беше свързан с Бейсик, който не беше проектиран като разширяем език, така че всичките разширения доведоха до ужасен и неуправляем синтаксис. От друга страна, C++, VB и други езици като Smalltalk имаха като цел на част от усилията при проектирането си справяне със сложността и това ги направи забележително ефективни при решаване на някои видове проблеми.

Това което ме впечатли най-много когато започнах да разбирам езика беше, че като неотменна цел стоеше да се опростят нещата за програмиста. Като да кажем "не се интересуваме от нищо друго освен от намаляване на времето и усилията за получаване на хубав код." В ранните дни това бе резултирало в не много голяма бързина на кода (въпреки многото обещания за бързината, която Java щял да достигне някой ден) но също доведе до вълнуващо намаление на времето за разработка; половина или по-малко от това за еквивалентна програма на C++. Този резултата сам по себе си може да спести чудесни количества време и пари, но Java не спира тук. Тя продължава с обхващането на други сложни задачи, които са станали важни като многонишково изпълнение и мрежово програмиране, чрез библиотеки и черти на езика, които правят тези задачи тривиални. И накрая той се заема с проблеми с наистина голяма сложност: програми за смесени платформи, динамична промяна на кода и даже сигурността, всеки от които лежи в спектъра на сложността между "пречка" и "спиращ шоуто." Така напук на проблемите с бързината, които видяхме, обещанието на Java е зашеметяващо: той може да ни направи значително по-ефективни програмисти.

Едно от местата, където виждам най-голяма възможност за влияние е Мрежата. Мрежовото програмиране винаги е било трудно, а Java го прави лесно (и разработчиците се трудят за това непрекъснато). Мрежовото програмиране е чрез което ние можем да комуникираме помежду си по-лесно и евтино отколкото да се обадим по телефона (само електронната поща революционизира много видове бизнес). Като си говорим повече започват да се случват вълнуващи неща, може би по-вълнуващи даже от обещанията на генната инженерия.

Във всеки случай: създавайки програмите, работейки в екипи за създаване на програми, изграждайки потребителски интерфейси така, че програмите да може да комуникират с потребителя, изпълняйки програмите на различни типове машини и лесно създавайки програми за комуникация по Мрежата – Java увеличава честотната лента досъпна за общуване между хората. И аз мисля че резултатите от революцията в комуникациите няма да се проявят в преминаване на големи количества битове наоколо. Ние трябва да видим истинската революция понеже ние всички ще можем да си говорим лесно – всеки със всеки, но също на групи или като цяла планета. Слушал съм че следващата революция ще е формирането на глобален разум съставен от достатъчно много хора и достатъчно много връзка между тях. Java може да подбуди или да не подбуди този процес, но само възможността за това ме кара да мисля, че правя нещо ценно, опитвайки се тук да уча на Java.

Предговор към 2^{ро} издание

Много чудесни коментарии имаше за първото издание на тази книга, което естествено беше много радващо. Понякога обаче някой се оплаква и по някаква причина най-честото оплакване е "книгата е твърде голяма." Някой припомни оплакването на австрийския император от работата на Моцарт: "Твърде много ноти!" (Не се сравнявам с Моцарт по никакъв начин). Освен това мога да предположа, че оплакването идва от някой, който тепърва

има да се запознае с големината на езика и не е виждал останалите книги по темата – например любимата ми справочна книга е *Core Java* на Cay Horstmann и Gary Cornell (изд. Prentice-Hall) която толкова се разрасна, че беше разделена на два тома. Въпреки това аз съм се старал да махна частите, които са останали или поне не основни. Това може да се направи без неудобства, защото стария вариант е в Мрежата (на адрес www.BruceEckel.com). Ако искате стария материал, той още е там и това е голямо облекчение за автора. Например може да забележите че главата “Проекти” вече я няма; два от проектите отидоха в други части и последния вече не бе подходящ да остане. Така че по всички правила книгата ще е по-тънка.

Да, ама не.

Разработването на езика продължава, в частност с разработването на всевъзможни потребителски интерфейси. Очевидно обхващането им не е в обсега на тази книга, но някои неща не могат да се пренебрегнат. Най голямото от тях е Java за сървъри (предимно Servlets и Java Server pages – JSPs), което наистина е чудесно решение за Мрежата, във връзка с което намерихме, че много броузери не са достатъчно добри, за да изпълняват клиентската страна. В добавка има отделен проблем за лесно създаване на програми за работа с бази данни, транзакции, по сигурността и др. под., които са свързани с Enterprise Java Beans (EJBs). Тези теми са обхванати в нова глава *Enterprise Programming*, която не може да бъде игнорирана.

Ще намерите също, че главата за мрежи е разширена с Jini (произнася се “genie” (джини) и не е акроним, а име) и JavaSpaces, две технологии които промениха начина на мислене за мрежовото програмиране. И разбира се книгата беше променена да използва навсякъде Swing GUI – отново, ако искате старата Java 1.0/1.1 може да я вземете от мрежата на онзи адрес.

Освен няколкото нови черти добавени в Java 2 и корекциите правени по книгата, другата главна промяна е галата за *колекции* (8), която сега е фокусирана върху Java 2 колекциите, които се използват в книгата. Подобрил съм също няя глава с по-дълбоко вникване в колекциите, в частност как работят хеш-функциите (така че сега знаете как правилно да създадете такава функция). Има и други размествания и промени, включващи премахването на някои приложения, които считам за ненужни повече, но ги имаше в излишък.

Извинявам се на онези, които пак не могат да понесат дългината на книгата. Вярвате или не, аз работих много за да направя книгата кратка. Напук на обема, има алтернативи, които могат да ви задоволят. Едната е, че има електронна форма и можете да я сложите в лаптоп и по този начин да не мъкнете много тежест. Ако отслабвате, има Palm Pilot версии на книгата. (Един ми каза, че би бил книгата на неговия Palm в леглото, със задно осветление, за да не ядоса жена си. Аз само мога да се надявам това да може го отнесе в страната на сънищата лесно.) Ако искате на хартия, аз знам хора, които си печатат една по една главите, слагат една в куфарчето си и я четат във влака.

Въведение

Като всеки човешки език Java дава начин за изразяване на концепции. Ако е успешен, той ще даде по-голяма леснота и гъвкавост във все по-нарастващите и усложняващи се проблеми за решаване.

Не може да гледате на езика като на сбор от различни характерни черти; някои неща са без определено значение, ако са изолирани от другите. Може да използвате сумата от частите само ако мислите за проектиране, не просто за кодиране. И за да разберете Java по този начин трябва да разбирате проблемите с езика и с програмирането изобщо. Тази книга дискутира програмните проблеми, защо са проблеми и начинът който Java дава за решаването им. По този начин нещата от езика дадени в някоя глава са продиктувани от възгledа ми за решаване на някой проблем. По този начин, по малко по малко, искам да ви придвижа до точката, където начинът на мислене с Java става ваш естествен език.

Във всеки случай, аз ще имам пред вид, че вие искате да си изградите модел и дълбоко разбиране на езика; ако намерите гатанка, вие ще можете да я пречупите през своя модел и да намерите отговора.

Необходими начални знания

Тази книга предполага, че имате известни начални знания за програмирането; знаете, че програмата се състои от оператори, идеите за подпрограма, функция и макрос,управляващи оператори като "if" и за цикъл като "while" и т.н. Вие обаче можете да имате тази информация от много места, като например макроезици или работа с Perl. Щом сте достигнали точката да нямаете проблеми с идеите на програмирането, вие ще можете спокойно да работите с тази книга. Разбира се, книгата ще бъде по-лесна за С програмистите и още повече за C++ програмистите, но не се притеснявайте, ако не сте работили с тези езици (но имате желание да работите здраво). Аз ще въвеждам идеите на обектно-ориентираното програмиране и основните управляващи механизми на Java, така че ще се запознаете с тях, а също и в първите упражнения – с операторите за управление.

Въпреки че често ще се правят справки с характеристиките на С и C++, това няма да е съществено за изложението, а за да даде възможност програмистите да сравнят Java в перспектива с тези езици от които, в края на краишата, произлиза Java. Аз ще се опитам да направя тези справки лесни и да обясня всичко, което не би било ясно за не-C/C++ програмисти.

Учене Java

Приблизително по същото време, когато излезе моята първа книга *Използване на C++* (Osborne/McGraw-Hill 1989) аз започнах да уча този език. Ученето на програмни езици беше станало моя професия; Виждал съм поклащане на глави, озадачени лица и загадъчни въпроси от сбушателите си по целия свят от 1989г. насам. Когато започнах да давам уроци на по-малки групи хора аз открих нещо по време на занятията. Даже ония които се есмихаха и си поклащаха главите не бяха наясно с много неща. Като председател по въпросите на C++ в Software Development Conference миналите няколко години (а сега също и за Java) забелязах, че лекторите имат тенденция да дават на обучаваните твърде много неща за твърде

малко време. Така че независимо от вариациите в слушателския състав човек губи към края част от тях. Може би това поставя твърде много въпроси, но като един от хората, които са противници на традиционния начин на четене на лекции (и за повечето хора тази нагласа произтича, вярвам, от досада), аз исках да се опитам да държа всеки на темпото.

Създадох си различни начини на представяне на материията с времето. Така например приключвах учеенето с упражнения и повторение (техника, която е добра и за проектирането на програми на Java). Накрая направих курс, използвайки целия си опит – опит, който радостно ще споделям дълго време. Той се заема с предмета в дискретни, лесни за смиление стъпки и на семинар с достъп до техниката (идеалната ситуация за учене) има упражнения след всеки урок. Аз сега чета този курс на публични Java семинари, за които може да се заинтересувате на <http://www.BruceEckel.com>. (въвеждащия семинар е достъпен също и на CD ROM. Информация – на същия адрес.)

Обратната връзка, която имам от семинарите ми дава възможност да променя и реорганизирам изложението докато то. Но тази книга не е точно семинар без частта, която се прави на машината – аз се опитах да включа колкото мога повече информация в тези страници и така да я структурирам, че да ви извлека до следващата тема. Най-вече книгата е предназначена за самотния читател, счупкал се с един нов за него език.

Цели

Като моята предишна книга *Thinking in C++*, тази книга също беше структурирана около изучаването на програмния език. В частност моята мотивация е да създам нещо, което ще ми помага в моите семинари. Когато мисля за дадена глава на книгата аз го правя имайки пред вид как ще стане един добър урок. Целта ми е да получа материала на порции, които могат да се предадат за едно събиране, следвани от упражнения, които могат да бъдат направени в обстановката на семинара.

Целите ми в тази книга са да:

1. Представя материала в кратки стъпки, смилаеми преди да се продължи нататък.
2. Използвам примери, които са колкото е възможно по-кратки. Това понякога пречи да се изследват проблеми, които са близки до реалните, но съм забелязал, че начинаещите са по-щастливи да разберат всеки детайл от решението на задачата, отколкото ако са впечатлени от дълбочината на решавания от тях проблем. Също има ясна граница на обема на кода, който може да бъде предаден в една класна стая. Поради тази причина не може да няма критики към мен заради „дребни“ примери, но аз ги приемам като помощ в опита ми да създам нещо педагогически полезно.
3. Внимателно да подбера последователността на представяне на чертите на езика така, че читателят да не вижда неща, които още не са обяснявани. Разбира се, това не винаги е възможно, но винаги съм се старал да има поне кратко въведение.
4. Да дам това, което е полезно да се знае за езика, а не всичко. Аз вярвам, че има йерархия в знанията и с 95% от тях никога не се сблъсква обикновения програмист. За пример от езика С, ако запомните наизуст таблицата за приоритет на операциите (аз никога не я научих), можете да пишете умен код. Ако обаче я използвате, това ще затрудни човека, четящ кода. Така че забравяме за приоритета и използваме скоби навсякъде, където нещата не са ясни без тях.

5. Държа фокуса във всяка лекция така, че времето между две лекции – и съответните упражнения – да бъде малко. Това не само държи слушателите по-активни и ангажирани по време на семинара, но и дава на читателя повече чувство за завършеност.
6. Дам солидна основа за решаване на сложни задачи и четене на други книги.

Онлайн документация

Езикът Java и библиотеки идват от Sun Microsystems (може да се снемат бесплатно) в електронна форма, читаема с практически всеки броузър и приблизително същата степен на документираност имат и доставките от други разработчици. Почти всички книги за Java дублицират тази документация. Така че вие или вече имате документация или можете лесно да си я свалите и тази книга няма да я повтаря, освен където това е необходимо, понеже обикновено е много по-бързо да си свалиш документацията от Мрежата, отколкото да ровиш в книга за нея. (Плюс че ще бъде последна версия.) тази книга ще дава допълнителни описания на класовете само където това е необходимо за разбиране на конкретните примери.

ГЛАВИ

Тази книга е проектирана като е имано пред вид само едно нещо: как хората учат Java. Обратната връзка от семинарите ми позволи да ведя кои неща са трудни за усвояване и се нуждаят от подробно разглеждане. В местата където твърде амбициозно бях включил прекалено много нови неща изведнъж аз можах да узная – чрез процеса на представяне на материала – че при това положение те всичките трябва да бъдат обяснени и това на свой ред лесно довежда обучаваните до неразбиране или отказ да учат. В резултат на това знание съм хвърлил много труд да направя въвеждането на колкото е възможно по-малко нови неща във всяка отделна стъпка.

Целта, тогава, е във всяка глава да се опише нова черта на езика или няколко близки, свързани черти, без да се използват такива, които са непознати до този момент. По този начин може да бъде асимилиран материалът напълно при всяка стъпка, преди да се продължи нататък.

Ето кратко описание на главите в тази книга, което съответства на моя опит от семинарите и упражненията:

Глава 1: Въведение в обектите

Тази глава е за всичко, което е предмет на обектно-ориентираното програмиране, вкл. основния въпрос “Що е обект?”, интерфейс и разликите му с приложение, абстракция и капсулиране, съобщения и функции, наследяване и композиция и важния за всичко полиморфизъм. Също има въведение в създаването на обекти с конструктори, къде живее обектът, къде да ги сложим като са веднъж създадени и магическият “събирач на боклук” който почиства от обектите щом те вече не са нужни. И други неща се въвеждат, включително обработката на грешки с изключения, многонишковост за потребителски интерфейси, които “не губят говор”, мрежи и Мрежата. Ще научите какво прави Java специален, защо той бива толкова успешен и за обектно-ориентираните анализ и проектиране.

Глава 2: Всичко е обект

Тази глава ви довежда до точката, където ще можете да напишете първата си програма така че главата дава общ поглед върху основните неща, включително концепцията за “дръжка” (манипулятор, който по същество е указател – б.пр.) към обект; как да се създаде обект; във-

дение към първичните типове и масиви; обхват и как обектите биват разрушавани от боклучаря; как всичко във Java е нов даннов тип (class) и как да създавате маши собствени класове; функции, аргументи и връщани стойности; видимост на имената и използване на класове зададени извън текущия файл; ключовата дума **static**; коментари и вградена документация.

Глава 3: Управление хода на програмата

Тази глава започва с всичките оператори, които идват от С и С++. В добавка ще намерите общите клопки, превръщания, промоции и приоритет. Това е последвано от основните оператори за управление хода на програмата, които са общи за практически всички езици за програмиране: избор с if-else; цикли с for и while; излизане от цикъл с break и continue както и етикетирани такива в Java (които заместват "липсващия goto" в Java); избор чрез switch. Въпреки че повечето такъв материал има общи точки в С и С++, има някои разлики. В добавка всичките примери ще са пълни Java примери така че по-добре ще усвоите как изглежда Java.

Глава 4: Инициализация и разчистване

Тази глава започва с въвеждането на конструктор, който гарантира правилна инициализация. Дефиницията на конструктор води до концепцията за презареждане на функции (бихме могли да искаем няколко конструктора). Това е последвано от дискусия за разчистването (преди излизане – б.пр.), което не винаги е толкова просто, колкото изглежда. Нормално просто зарявате обекта когато повече не е необходим и като мине боклучаря, почиства всичко, освобождавайки паметта. Тази част изследва боклучаря и някои негови особености, идиосинкарзии. Главата завършва с по-близък поглед към инициализацията: автоматична инициализация на членове, задаване на инициализация, реда на инициализация, **static** инициализация и инициализация на масиви.

Глава 5: Скриване на реализацията

Тази глава разглежда как се свързват заедно парчетата код и как някои части от библиотека са видими докато други са скрити. Тя започва с разглеждането на ключовите думи **package** и **import**, които позволяват опаковането на файлово ниво и позволяват създаването на библиотеки от класове. Въпросът с каталожните пътища и файловите имена също е разгледан. Останалата част от главата разглежда ключовите думи **public**, **private** и **protected**, концепцията за "приятелски" достъп и какво различните нива на достъп означават, когато са в различен контекст.

Глава 6: Повторно използване на класове

Концепцията за наследяване се споделя от практически всички ООР езици. Това е начин да се вземе съществуващ клас и да се разшири неговата функционалност (както и да се промени, темата на Глава 7). Наследяването е често начин да се използува повторно код, като се остави "базовия клас" същия и само се променят някои неща тук и там за постигане на целите. Наследяването обаче не е единствения начин да се получат нови класове от съществуващи такива. Може също да се вгради един клас в друг чрез композиция. В тази глава ще научите за тези две техники на повторно използване на кода в Java и как да ги използвате.

Глава 7: Полиморфизъм

Самички бихте могли да изразходвате девет месеца за откриване и разбиране на полиморфизма, крайъгълен камък на ООР. Чрез малки, прости примери ще се научите как да създавате семейство типове чрез наследяване и как да манипулирате обектите им чрез базовия клас. Полиморфизмът на Java позволява да се третират всички обекти от един тип генетично, което значи, че вашият код не зависи от специфична за типа информация. Това прави програмите ви разширяеми, така че произвеждането и управлението на програмните проекти е по-лесно и управляемо. В добавка Java дава трети път да се устрои повторно използване на кода чрез интерфейс, което е чиста абстракция на интерфейс на обект.

Веднъж като сте станали наясно с полиморфизма интерфейсите са лесни за разбиране. Тази глава също въвежда вътрешни класове на Java 1.1.

Глава 8: Владеене на обекти

Доста проста е всяка програма, която има фиксирано число обекти с известно време на живот. Изобщо, вашите програми ще създават най-различни обекти които ще бъдат ясни чак по времето, когато програмите работят. В добавка вие не бихте искали да знаете даже типа на обектите чак до пускането на програмата. За да решите общия проблем трябва да можете да създавате неограничено количество обекти, навсякъде, по всяко време. Тази глава дълбоко разглежда колекциите, които Java 2 доставя за да владеете обектите докато работите с тях: прости масиви и по-засуки колекции (структури от данни) като **ArrayList** и **HashMap**.

Глава 9: Обработка на грешки с изключения

Основна философия в Java е, че лошо написан код не трябва да бъде пускан. Компилаторът хваща проблемите доколкото е възможно, но понякога проблемите – били те програмни грешки или естествени състояния, възникващи в някои случаи при изпълнение на програмата – могат да бъдат хванати и обработени само по време на изпълнение. Java има *exception handling* да се разправя с всякакви проблеми, които възникват, когато програмата работи. Тази глава разглежда как ключовите думи **try**, **catch**, **throw**, **throws** и **finally** работят в Java; кога трябва да изхвърлите изключение и какво да правите, когато го хванете. В добавка ще видите стандартните изключения в езика, как да създавате собствени, какво става с изключенията в конструкторите и как се намира кодът, който да обработи изключенията.

Глава 10: Входна-изходна система на Java

Теоретично всяка програма прави три неща: вход, обработка и изход. Това предполага, че IO (вход/изход) е доста важна част от темата. В тази глава ще видите различни входно-изходни класове които Java има за четене на файлове, блокове памет и конзолата. Разликата между "стария" IO и "новия" Java 1.1 IO ще бъде показана. В добавка тук ще се покаже вземане на обект, неговия "streaming" (така че да може да бъде пратен на диск през мрежата) и реконструкцията му, което се прави за вас от Java версия 1.1. Също се разглеждат компресираните библиотеки на Java 1.1, които се използват в Java Archive файлов формат (JAR).

Глава 11: Идентификация на типа по време на изпълнение

Идентификацията на типа по време на изпълнение в Java (RTTI) ви дава възможност да установите точния тип на обекта, ако само имате достъп до базовия тип. Нормално ще се стремите да не се занимавате с конкретни типове и ще оставите полиморфизъмът (който е вграден в езика) да върши това. Но понякога е много полезно да може да узнаете конкретния тип на обекта. Често тази информация позволява да се направят някои специални операции по-ефективни. Тази глава обяснява за какво е RTTI, как да се използва и как да се отървем от него, когато вече не е необходимо. В добавка се разглежда рефлексията - черта на Java 1.1.

Глава 12: Предаване и връщане на обекти

Доколкото единствения начин да комуникирате с обектите в езика е чрез "handles," концепцията за предаване на обект като параметър на функция и връщането на обект като резултат от функция има интересни продължения. Тази глава обяснява как да управлявате обекти когато влизате или излизате от функция и също показва класа **String** който използва различен подход към проблема.

Глава 13: Създаване на прозорци и аплети

Java идва със "Swing" GUI библиотеката, която е множество от класове, които поддържат създаването и управлението на прозорци по преносим начин; тези програми могат да бъдат както аплети, така и самостоятелни програми. Тази глава е въведение в Swing създаване на аплети за World Wide Web. Важната "Java Beans" технология е въведена. Тя е от основно значение за създаване на Rapid-Application Development (RAD) инструменти за създаване на програми.

Глава 14: Многонишковост

Java има възможност да изпълнява едновременно няколко подзадачи в рамките на една програма, наречени **нишки**. (Докато нямате няколко процесора във вашата инсталация само изглежда че се работят едновременно.) Въпреки че могат да се използват навсякъде, нишките са най-полезни например в случай, че искате да направите потребителски интерфейс такъв, че да може да се натискат клавиши или въвеждат данни докато се изпълнява някакъв друг конкретен процес. Тази глава поглежда и към синтаксиса и семантиката на многонишковостта в Java.

Глава 15: Мрежово програмиране

Всички черти на Java и библиотеките сякаш се обединяват когато започнете да разработвате програми за работа в мрежи. Тази глава изследва комуникациите по Мрежата и класовете, които езикът доставя за улеснение на тази работа. Тук също се разглежда как да се създаде Java аплет като да взаимодейства с *common gateway interface* (CGI) програма, показва как да се пишат CGI програми на C++, обхващаща Java DataBase Connectivity (JDBC) и *Remote Method Invocation* (RMI) в Java 1.1.

Глава 16: Шаблони

Тази глава въвежда много важния и още не традиционен подход с "шаблони". Изучава се примерна еволюция на процеса на проектиране, започвайки от начално състояние и движейки се през логиката и придвижквнето на проекта до по-подходящи състояния на разработка. Ще видите начин, който разработката може да материализира с течение на времето.

Глава 17: Проекти

Таза глава включва проекти, които са свързани с материала от цялата книга или по някаква друга причина не са могли да бъдат включени м по-ранните глави. Тези проекти са значително по-сложни от останалите примери в главите и често демонстрират допълнителни черти и техники на езика и библиотеките.

Има теми, които не могат да се включат в никоя глава. Те са засегнати в приложениета.

Приложение A: Използване на не-Java код

Една напълно преносима Java програма има сериозни недостатъци: скоростта и невъзможността да се използват специфични за дадена платформа услуги. Когато платформата е известна, възможно е драматично да се ускорят някои операции, като се направят *native methods*, които са функции, написани на друг програмен език (в момента се поддържат само C/C++). Има и други пътища Java да поддържа не-Java код, включително CORBA. Това приложение прави достатъчно въвеждане във въпроса, така че да може да се построят прости програми, които взаимодействват с не-Java код.

Приложение B: Сравнение на C++ с Java

Ако сте C++ програмист вие вече имате основната идея за обектно-ориентираното програмиране и несъмнено синтаксисът на Java ви е доста познат. Това има смисъл понеже Java произлезе от C++. Има обаче изненадващо много разлики между C++ и Java. Тези разлики са направени с намерение да се подобри езикът и ако ги разберете, вие ще знаете защо новият език е така успешен. Това приложение ви води през основните черти, които правят Java различен от C++.

Приложение C: Правила за програмиране на Java

Това приложение съдържа правила, които да ви помогнат при разработването на проекти на ниско ниво и писането на код.

Приложение D: Изпълнение

Това ще ви помогне да намерите причините за задръствания и да повишите скоростта на изпълнение на програмите си.

Приложение E:**Малко за събирането на отпадъците**

Тази глава описва операциите и подходите при освобождаването на паметта, заета от вече ненужни обекти.

Приложение F:**Препоръчано за четене**

Списък на книги за Java които смятам, че биха били полезни.

Упражнения

Открил съм, че упражненията са особено полезни за пълното разбиране на изложението и затова ще намерите такива в края на всяка глава.

Повечето упражнения са такива, че могат да се направят за разумно време в обстановката на семинара, под погледа на инструктора, осигурявайки че всички студенти са абсорбирали материала. Някои са по-напреднали – за по-напредналите обучавани – за да се избегне досада. По-голямата част са такива, че да проверят и допълнят знанията. Някои представляват предизвикателства, но няма големи предизвикателства. (Може да се предполага, че вие ще ги намерите – или по-вероятно те ще ги намерят).

Мултимедиен CD ROM

Отделно е достъпен Мултимедиен CD ROM придружаващ тази книга, но той не е като CD-тата които обикновено ще намерите опаковани с книгите. Те често съдържат само сорса на книгата. (За тази книга той е достъпен безплатно от Web страницата www.BruceEckel.com.) Този CD ROM е отделен продукт и съдържа целия “Hands-On Java” семинар. Това са повече от 15 часа лекции на Bruce Eckel, синхронизирани с 500 слайда информация. Семинарът е базиран на тази книга, така че това е идеалния придружител.

CD ROM-ът съдържа две версии на книгата:

1. Подходяща за печат, идентична на тази сложена в Мрежата.
2. За лесни справки директно на екрана, подходящо форматирана и с необходимите връзки, достъпна само от CD-ROM-а. Връзките включват:
 - 230 броя за главите, секциите и подзаглавията
 - 3600 броя индексни

CD ROM-ът съдържа над 600MB данни. Вярваме, че това поставя нов стандарт за тази стойност.

Има всичко от печатуемата версия на книгата и всичко (с важното изключение на персонално пожеланите за разглеждане въпроси!) от петдневен пълноценен семинар с упражнения. Вярваме, че това поставя нов стандарт за качество.

CD ROM-ът е достъпен само чрез поръчка на адрес: www.BruceEckel.com.

Copс

Целият изходен код на тази книга е достъпен като copyrighted freeware, предлаган като единен пакет от адрес <http://www.BruceEckel.com>. за да сте сигурни, че разполагате с текуща версия, това е официалният адрес за дистрибуция на книгата. Може да намерите други (“отразени” – б.пр.) страници с книгата (някои от тях са цитирани на

<http://www.BruceEckel.com>), но ще видите на оригиналната страница дали това са последни версии. Кодът може да се възпроизвежда в класна стая или друга ситуация на обучение.

Първичната цел на авторското право е да осигури, че материалът е точно цитиран и да предотврати публикуването в печатна медия без разрешение. (Ако източникът е цитиран, цитирането на изводки в повечето медии не е проблем.)

Във всеки първичен текст ще намерите следния запис:

```
//:! :CopyRight.txt
Copyright (c) Bruce Eckel, 1999
Source code file from the 2nd edition of the book
"Thinking in Java." All rights reserved EXCEPT as
allowed by the following statements:
You can freely use this file
for your own work (personal or commercial),
including modifications and distribution in
executable form only. Permission is granted to use
this file in classroom situations, including its
use in presentation materials, as long as the book
"Thinking in Java" is cited as the source.
Except in classroom situations, you cannot copy
and distribute this code; instead, the sole
distribution point is http://www.BruceEckel.com
(and official mirror sites) where it is
freely available. You cannot remove this
copyright and notice. You cannot distribute
modified versions of the source code in this
package. You cannot use this file in printed
media without the express permission of the
author. Bruce Eckel makes no representation about
the suitability of this software for any purpose.
It is provided "as is" without express or implied
warranty of any kind, including any implied
warranty of merchantability, fitness for a
particular purpose or non-infringement. The entire
risk as to the quality and performance of the
software is with you. Bruce Eckel and the
publisher shall not be liable for any damages
suffered by you or any third party as a result of
using or distributing software. In no event will
Bruce Eckel or the publisher be liable for any
lost revenue, profit, or data, or for direct,
indirect, special, consequential, incidental, or
punitive damages, however caused and regardless of
the theory of liability, arising out of the use of
or inability to use software, even if Bruce Eckel
and the publisher have been advised of the
possibility of such damages. Should the software
prove defective, you assume the cost of all
necessary servicing, repair, or correction. If you
think you've found an error, please submit the
correction using the form you will find at
www.BruceEckel.com. (Please use the same
form for non-code errors found in the book.)
///:~
```

Може да използвате текста във вашите проекти или класна стая (включително презентационни материали) ако горния текст за авторско право е възпроизведен там.

Стандарти при кодирането

В текста на тази книга идентификаторите (имената на функции, променливи и класове) ще бъдат **ярки**. Повечето ключови думи също, освен за много често използвани, такива като "class."

Аз използвам определен стандартен стил на записване при упражненията в тази книга. Този стил следва указанията които ще намерите на www.javasoftware.com и изглежда да е поддържан от повечето разработчици. Той беше разработен в течение на години и беше вдъхновен от стила на Bjarne Stroustrup в неговата оригинална *The C++ Programming Language* (Addison-Wesley, 1991; 2nd изд.). Темата за стила е подходяща за могочасов горещ дебат, аз само ще отбележа, че не мисля да диктувам стила; имам си свои причини да използвам именно този стил. Понеже езикът няма ограничения за стила, може да продължите да използвате какъвто си искате стил.

Програмите в тази книга просто са вмъкнати с текстов процесор така, както са били за компилация, така че не би трябвало да се получават грешки. Грешките които трябва да предизвикат грешки при компилация са коментирани с `//!` Така че могат лесно да бъдат открити с автоматизирано търсене. Грешките открити и докладвани на автора ще се оправят първо на началната страница, а после на огледалните (което отново ще намерите на <http://www.BruceEckel.com>).

Версии на Java

Въпреки че съм тествал упражненията в тази книга с продуктите на няколко доставчици на Java, аз изобщо приемам поведението на продукта на Sun като стандарт при определянето какво поведение трябва да има дадената програма.

Sun има три главни версии на Java: 1.0, 1.1 и 2. Версия 2 изглежда че накрая извежда Java във време на разцвет, в частност що се отнася до потребителските интерфейси. Тази книга е фокусирана върху и тествана с Java 2. Ако искате да учите старите версии, които не са разгледани в тази книгапървото издание може да се свали без заплащане от www.BruceEckel.com.

Може да забележите, че когато говоря за по-раншни версии на езика, аз не споменавам подверсии. В тази книга ще има справки към Java 1.0, Java 1.1 и Java 2 само, като предпазна мярка срещу печатни грешки при бъдещо ново подреждане на подверсии.

Семинари и наставничество

Моята компания провежда петдневни, с упражнения, публични и частни семинари базирани на материала от тази книга. Избран материал от всяка глава представя урок, който се следва от упражнение, в което всеки обучаван получава персонално внимание. Лекциите и въвеждащите слайдове са също налични на CD-ROM за да се достави най-малкото част от опита в семинарите без разходи за пътуване. За повече информация виж:

<http://www.BruceEckel.com>

или email:

<mailto:Bruce@EckelObjects.com>

Компанията ми също провежда консултантска дейност в помощ на проектите ви – особено на първия Java проект на компанията ви.

Грешки

Колкото и трикове да използва писателят за откриване на грешки, някои винаги остават и дразнят взискателния читател. Ако намерите каквото и да е, за което вярвате, че е грешка, моля изпратете оригиналния файл (който може да намерите на <http://www.BruceEckel.com>) с ясни коментари за грешката (ползвайте формулара даден на WEB страницата) и нужната според вас корекция на <mailto:Bruce@EckelObjects.com> така че да може да бъде оправена в електронния екземпляр и при следващото печатане на книгата. Като изпращате корекция моля използвайте следния формат:

1. Пишете "TIJ Correction" (без кавички и нищо друго) на subject линията – за да може моят пощаљон да сложи съобщението на нужното място.
2. В тялото на съобщението използвайте, моля, формата:

```
find: one-line string to search for
comment:
multi-line comment, best starting with "here's how I think it should read"
###
```

където '###' означава край на коментара. По този начин моите инструменти за корекции ще открият точния текст и в съседен прозорец ще извадят вашето предложение за корекция. (ВСИЧКО ТОВА СЕ ОТНАСЯ САМО ЗА АНГЛИЙСКИЯ ОРИГИНАЛ - б.пр.)

Предложения за конкретни упражнения или за разглеждане на специфични теми са добре дошли. Помощта ви е високо ценена.

Бележка по дизайна на корицата

Корицата на *Thinking in Java* е вдъхновена от American Arts & Crafts Movement, което започна някъде към началото на столетието и достигна зенита си между 1900 и 1920. Това е реакция, възникнала в Англия както на промишлената революция, така и на много орнаментния стил на Викторианска епоха. Подчертано икономичния дизайн на Arts & Crafts, природните форми като част от новостите в дизайна, ръчната изработка и важността на занаятчиета, даже самото то не можа да спре използването на съвременни инструменти. Има много общо със днешната ситуация: изващата смяна на столетието, еволюцията от простичкото начало на компютърната революция към нещо по-префинено и значително за отделната личност и подчертаването на софтуерното занаятчийство в сравнение с манифактурата на програми.

Виждам Java по същия път: като опит да издигне встриди софтуериста от механиката на операционните системи и да го направи "софтуерен занаятчия". (без оттенък на презрение – б.пр.)

И авторът, и дизайнерът на книгата/корицата (които са приятели от детинство) намират вдъхновение от това движение, и мебелите, лампите и другите неща на двамата, които са или оригинални, или вдъхновени по същия начин.

Другата тема, която предлага корицата е кутия, която един натуралист би могъл да използува като съхранява видовете насекоми, които събира. Тези насекоми са обекти, сложени в кутии-обекти, кито се съхраняват в "покриващ" обект – основната концепция на обектно-ориентираното програмиране. Разбира се, един програмист не би могъл да направи асоциация с "bugs" и ето ги бъговете хванати и предполагаемо убити в стъкленицата за видове, и накрая затворени в малко прозорче на дисплея, като да прилага способността

на Java да хваща, прави видими и неутрализира грешките (което е наистина един от най-мощните атрибути на езика).

Благодарности

Преди всичко, благодаря на Doyle Street Cohousing Community за връзките ми с тях за две години, като се стигна до написването на тази книга (а и въобще). Благодаря много на Kevin и Sonda Donovan за даването на тяхното чудно място във великолепния Crested Butte, Colorado за лятото, когато работех над книгата. Също благодаря на приятелите от Crested Butte и Rocky Mountain Biological Laboratory които направиха да се чувствам добре дошъл. World Gym в Emeryville и неговия ентузиазиран състав ми помогнаха да остана нормален в последния стадий на написване на книгата.

За пръв път използвам услугите на агент и нямам поглед назад. Благодаря на Claudette Moore от Moore Literary Agency за нейното огромно търпение и постоянство да постигна това, което исках.

Първите ми две книги бяха публикувани с Jeff Pepper като редактор в Osborne/McGraw-Hill. Jeff се появи в точното време и точното място в Prentice-Hall и очисти пътя и направи всичко необходимо аз да преживея най-хубавото издаване на книга, каквото някога съм преживявал. Благодаря, Jeff – то значи много за мене.

Специално съм задължен на Gen Kiyooka и компанията му Digigami, който любезно ми е предоставил моя Web сървъри на Scott Callaway който я е управлявал. Това беше неоценима подкрепа когато учех за Web.

Благодаря на Cay Horstmann (съавтор на *Core Java*, Prentice Hall 1997), D'Arcy Smith (Symantec) и Paul Tuma (съавтор на *Java Primer Plus*, The Waite Group 1996) за помощта при изясняването на концепциите на езика.

Благодаря на хората, които са говорили на моя Java track в Software Development Conference, на студентите от моите семинари които задаваха въпросите, които исках да чуя, за да направя изложението по-ясно.

Специални благодарности към Larry и Tina O'Brien, които превърнаха тази книга и семинарите ми в CD ROM. (Може да намерите повече за него на <http://www.BruceEckel.com>.)

Задължен съм на много хора, които изпращаха корекции за книгата, но специално на: Kevin Raulerson (намерил тонове големи насекоми), Bob Resendes (просто чудесен), John Pinto, Joe Dante, Joe Sharp (и тримата бяха баснословни), David Combs (много граматични и за яснотата корекции), Dr. Robert Stephenson, Franklin Chen, Zev Griner, David Karr, Leander A. Stroschein, Steve Clark, Charles A. Lee, Austin Maher, Dennis P. Roth, Roque Oliveira, Douglas Dunn, Dejan Ristic, Neil Galarneau, David B. Malkovsky, Steve Wilkinson и много други.

Проф. Ir. Marc Meurrens положи много усилия за издаването на книгата в Европа.

Има голям поток от кадърен технически персонал в моя живот с които аз станах приятел и които бяха влиятелни и необичайни с това, че правеха йога и други форми на душевно усъвършенствуване, които аз намирам много вдъхновяващи и поучителни. Такива са Kraig Brockschmidt, Gen Kiyooka и Andrea Provaglio, които помага за разбирането на Java и програмирането изобщо в Италия.

Не е голяма изненада за мен, че разбирането на Delphi ми помогна да разбера Java, доколкото има много общи концепции и решения. Моите приятели по Delphi ми помогнаха да надникна отвътре в тази чудесна прог LatI рамна среда. Те са Marco Cantu (друг италианец – може би като си пропит с латински дава дарба за езици?), Neil Rubenking (който се занимаваше с yoga/vegetarian/Zen но откри компютрите) и разбира се Zack Urlocker, дългогодишен другар, с който съм пропътувал света.

Подкрепата и вникването на моя приятел Richard Hale Shaw бяха от голяма помощ (също и на Kim). Richard прекарахме много месеци в преподаване и усилия да направим обучението перфектно. Благодаря също на KoAnn Vikoren, Eric Faurot, Deborah Sommers, Julie Shaw, Nicole Freeman, Cindy Blair, Barbara Hanscome, Regina Ridley, Alex Dunne и други на длъжност и в състава наMFI.

Книгата, корицата и фотото на корицата бяха от моия приятел Daniel Will-Harris, прочут автор и дизайнер (<http://www.Will-Harris.com>), който си играеше да прави букви-ваденки в ученическите ни години докато ние чакахме изобретяването на програмите за текстообработка и мънкаше срещу моите алгебрични проблеми. Обаче аз направих сам страниците в предпечатната подготовка, така че типографските грешки са си мои. Microsoft® Word 97 for Windows беше използван за набирането и предпечатната подготовка. Шрифтът е *Bitstream Carmina* за основния текст и заглавията са с *Bitstream Calligraph 421* (www.bitstream.com). Знациите в началото на главите са *Leonardo Extras* от P22 (<http://www.p22.com>). На корицата е *ITC Rennie Mackintosh*.

Благодаря на доставчиците, които ме снабдиха с компилатори: Borland, Microsoft, Symantec, Sybase/Powersoft/Watcom и, разбира се, Sun.

Специални благодарности на всичките ми учители и ученици (които са мои учители също така). Най-забавния писател, от който съм се учили е Gabrielle Rico (автор на *Writing the Natural Way*, Putnam 1983). Винаги ще ценя високо ужасната седмица в Esalen.

Множеството подкрепили ме приятели включва, но не се изчерпва с: Andrew Binstock, Steve Sinofsky, JD Hildebrandt, Tom Keffer, Brian McElhinney, Brinkley Barr, Bill Gates в *Midnight Engineering Magazine*, Larry Constantine и Lucy Lockwood, Greg Perry, Dan Puttermann, Christi Westphal, Gene Wang, Dave Mayer, David Intersimone, Andrea Rosenfield, Claire Sawyers, още италианци (Laura Fallai, Corrado, Ilsa и Cristina Giustozzi), Chris и Laura Strand, Almquists-ови, Brad Jerbic, Marilyn Cvitanic, Mabrys-ови, Haflingers-ови, Pollocks-ови, Peter Vinci, семействата Robbins, семействата Moelter (и McMillans-ови), Michael Wilk, Dave Stoner, Laurie Adams, Cranstons-ови, Larry Fogg, Mike и Karen Sequeira, Gary Entsminger и Allison Brody, Kevin Donovan и Sonda Eastlack, Chester и Shannon Andersen, Joe Lordi, Dave и Brenda Bartlett, David Lee, Rentschlers-ови, Sudeks-ови, Dick, Patty и Lee Eckel, Lynn и Todd и семействата им. И, разбира се, Мама и Тати.

За Интернет

Благодара на тези, които ми помогнаха да пренапиша примерите с използване на Swing библиотеката: Jon Shvarts, Thomas Kirsch, Rahim Adatia, Rajesh Jain, Ravi Manthena, Banu Rajamani, Jens Brandt, Nitin Shivaram и всеки, който изрази поддръжка. Това наистина ми помогна да започна проекта с летящ старт.

1: Въведение в обектите

Защо обектно-ориентираното програмиране има такова ударно влияние върху програмистката общност?

Обектно-ориентираното програмиране е привлекателно на много нива. За менажерите то обещава по-евтини и управляеми проекти. За аналитите и проектантите процесът се опростява и става по-управляем. За програмистите елегантността е привлекателна, а също и повишенната производителност, те могат да експериментират и да я увеличат още повече. Види се, всички печелят.

Ако има обратна страна това е цената на кривата на обучението. Мисленето в термините на обекти драматично се различава от процедурното мислене и проектирането на обекти е много по-голямо предизвикателство от процедурите, особено ако се цели да се правят повторно използваеми обекти. В минарото новакът в обектно-ориентираното програмиране беше изправен пред две плашещи алтернативи:

1. Да избере език като Smalltalk при което трябва да изучи много и големи библиотеки аокато стане продуктивен.
2. Да избере C++ с фактически никакви библиотеки¹ и да се бори с дълбочините на езика поради нуждата да пише свои собствени библиотеки.

Да се правят хубави обекти е наистина трудно – ако е въпросът, трудно е да се направи добре каквото и да е. Намерението е обаче няколко добри проектанта да произведат обектите, които други ще използват. Успешните ООП езици предлагат не само синтаксис и компилатор, но и програмна среда с добре разработени библиотеки. По този начин най-първата задача на повечето програмисти е да използват тези обекти. Целта на тази глава е да покаже какво е това ООП и колко просто може да бъде.

Тази глава ще въведе много от идеите на Java и обектно-ориентираното програмиране, но да сте наясно, че още няма да можете да пишете завършени Java след четенето на тази глава. Всичките детайлни описание и примери ще се дават с течение на курса.

Абстракцията

Всички програмни езици доставят абстракция. Може да се спори дали сложността на проблемите, които може да се решават зависи от вида и качеството на абстракцията. Под "вид" лазбирам: от какво се абстрагираме в рамките на езика? Езикът асемблер е малка абстракция на лежащата отдолу машина. Много от последвалите т.н. "императивни" езици (като FORTRAN, BASIC и C) бяха абстракции на асемблера. Тези езици са голям напредък в сравнение с асемблерите, но и при тях все още се налага да се мисли в термините на машината, заместо на решавания проблем. Програмистът трябва да прокара съответствие между машинния модел" (в "пространството на решенията) и модела на фактически решавания проблем (в "проблемното пространство"). Усилието, необходимо за това, което е външно за

¹ Fortunately, this has changed significantly with the advent of third-party libraries and the Standard C++ library.

решавания проблем, довежда до програми, които са скъпи за произвеждане и трудни за поддържане, а като страничен ефект се получава цяла индустрия на "програмните методи".

Алтернативата на моделирането е моделиране на решавания проблем. Ранните езици като LISP и APL избраха конкретни свои гледни точки към света ("всички проблеми са списъци" и "всички проблеми са алгоритмични"). PROLOG свежда всички проблеми до вериги от решения. Бяха създадени езици за програмиране на/с ограничения и за работа изключително със знаци. (Последните се доказаха като твърде ограничителни.) Всеки от тези езици е добър за решаване на конкретния вид проблеми, за който са създадени, но стават тромави при всеки опит да се надникне навън.

Обектно-ориентирания подход прави стъпка по-нататък с даването на възможност да се моделират отделни части от проблемното пространство. Това представяне е толкова общо, че не поставя никакви ограничения върху вида на решавания проблем. Ние се отнасяме към нещата в проблемното пространство от пространството на решенията като към "обекти". (Разбира се, ще са нужни и обекти, които нямат аналоги в проблемното пространство.) Идеята е програмата на може да се настройва към езика на проблема чрез въвеждане на нови типове обекти така, че като четете кода който представлява решението му, да четете думи, които описват проблема. Това е по-гъвкава и мощна абстракция от всички предишни. Така ООП позволява да се опише проблема в термините на проблема, а не на решението. Има обаче връзка и с компютъра. Всеки обект изглежда мъничко като отделен компютър; има състояние, има операции, които може да се поръчат да изпълни. Това обаче не изглежда лоша аналогия на обектите от реалния свят; те всички имат своя характеристика и поведение.

Alan Kay сумира петте основни характеристики на Smalltalk, първият успешен ООП език и един от тези, на които е основан Java. Тези характеристики представят подхода на ООП в чист вид:

1. **Всичко е някакъв обект.** Мислете си го като фантастична променлива - може да се запомнят данни, но също може и да се поръчват обработки на данните. Теоретично може да се вземе който и да е обект от концепцията на решението на проблема (кучета, постройки, услуги т.н.) и да се представи като обект в програмата.
2. **Програмата се състои от обекти, които си казват един на друг какво искат да правят чрез съобщения.** За да поръчате нещо на обект му изпращате "съобщение". По-конкретно, за съобщението може да се мисли като за извикване на конкретна функция от конкретен обект.
3. **Всеки обект си има своя собствена памет попълвана от други обекти.** Или, прави се нов вид обект като се пакетират съществуващите обекти. Така може да построите сложността на програмата скривайки я зад простотата на обектите.
4. **Всеки обект има тип.** Всеки обект е екземпляр на клас, където "клас" е синоним на "тип." Най-отличаващата характеристика на даден клас е "какви съобщения може да му се пращат?"
5. **Всички обекти от даден тип могат да приемат едни и същи съобщения.** Фактически това е пресилено твърдение, както ще видите по-нататък. Понеже обектът от тип кръг е също и от тип форма, кръгът гарантирано приема съобщенията за форма. Това значи че като напишете код който говори на формите автоматично включвате всичко, което има форма. Тази заменяемост е една от най-мощните концепции на ООП.

Някои проектанти са решили, че ООП не решава проблемите и адвокатстват на комбинацията на няколко подхода в многопарадигмните програмни езици.²

² See *Multiparadigm Programming in Leda* by Timothy Budd (Addison-Wesley 1995).

Всеки обект си има интерфейс

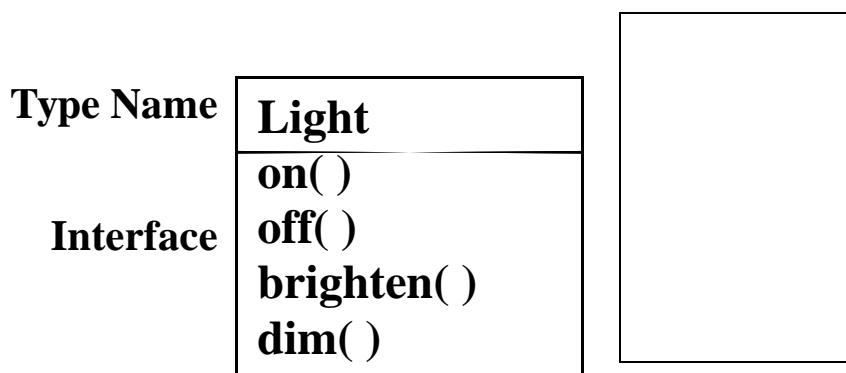
Вероятно Аристотел пръв е изследвал понятието "тип". Известно е, че той е говорил за "класа на рибите и класа на птиците." Концепцията че, макар да са уникални, обектите имат общи свойства с други обекти, е използвана в първия ООП език, Simula-67, с неговата основна ключова дума **class** която въвежда в програмата нов тип (така **клас** и **тип** се употребяват често като синоними³).

Simula, както предполага името, беше създаден за симулация на ситуации като в класическия "проблем на банковия касиер." В него има много касиери, клиенти, сметки, транзакции и т.н.. Членовете (елементите) от всеки клас имат общи неща помежду си: всяка сметка има баланс, every teller всеки касиер може да приема депозити и т.н. В същото време всеки член има негово конкретно собствено състояние; всяка сметка има свой си баланс, всеки касиер си има име. Така касиерите, клиентите, сметките, транзакциите и т.н. могат да бъдат представени всяка с отдельна същност в компютърна програма. Тази същност е обект и всеки обект принадлежи на клас, който определя чертите и поведението.

Така, въпреки че се създават нови типове, практически всички ООП езици използват ключовата дума "class". Когато видите "тип" мислите "клас" и обратно.

Като създавате веднъж типа, може да създавате и манипулирате колкото си искате обекти от полето на решавания проблем. Разбира се, едно от предизвикателството на ООП е да се създава едно-към-едно изображение между нещата от проблемното пространство (мястото където проблемът фактически съществува) и пространството на решението (мястото, където моделирате проблема, като компютър например).

Как обаче караме обекта да върши полезна работа? Трябва да има начин да му се поръча да свърши нещо полезно, като да извърши транзакция, начертава нещо на еcran или включи верига. Всеки обект може да удовлетвори само определени поръчки. Поръчките към обекта са определени от неговия интерфейс и тиха, който определя интерфейса. Идеята типът да бъде еквивалентен на интерфейса е основна за ООП.



Прост пример може да бъде представяне на електрическата крушка:

```
Light lt = new Light();
lt.on();
```

Името на типа/класа е **Light** и поръчките, които може да давате на обекта **Light** са да включвате, изключвате, усилвате или намалявате светлината. Вие създавате "дръжка" за **Light** просто декларирайки име (**lt**) за нея и построявате обекта **Light** с ключовата дума **new**, приравнявайки я към дръжката (някои биха предпочели "манипулатор" - б.пр.) със знака **=**. За да се изпрати съобщение на обекта споменавате името на дръжката, следвано от името на

³ Some people make a distinction, stating that type determines the interface while class is a particular implementation of that interface.

операцията, разделени с точка ("."). От мястото на програмист, работещ с предварително създадени класове това изглежда (и е) напълно достатъчно за ООП работа.

Скриване на код

Полезно е да се разделят ролите на *създатели на класове* (тези които създават класове) и *програмисти-клиенти*⁴ ("консуматори" на класове), които ги използват в своите програми). Целта на последните е да съберат пълно сандъче с инструменти, които да използват за бързо писане на приложения. Целта на създателя на класове е да остави видимо само най-необходимото видимо и да скрие всичко останало. Защо? Ако е скрито, клиентът не може да го използува, което значи, че създателят на класа може да го променя, без това да влияе на когото и да било друг.

Интерфейсът задава какви какви поръчки може да отправяте към конкретен обект. Трябва обаче някъде да има код, който да изпълни поръчката. Така заедно със скритите данни върви и *приложението*. От процедурна гледна точка това не е толкова сложно. Всеки тип има функция, асоциирана с конкретна възможна поръчка и когато се даправи поръчка, кодът се изпълнява. Това често сумарно се изразява с думите "изпращане на съобщение" (поръчка) към обект и обектът прави необходимото (изпълнява код).

Във всяко взаимоотношение е важно е да има граници, които се уважават от всички страни. Когато създавате библиотеки вие установявате взаимоотношения с клиентите, които са други програмисти, използващи вашата библиотека за създаване на програми или по-големи библиотеки.

Ако всички членове на клас са достъпни за всеки, клиентите биха могли да правят всичко с тях и няма начин да се наложи някакво определено поведение в тази дейност. Даже и да не искате клиентите да променят някои неща, не може да го предотвратите.

Има две причини да се контролира досъпа до членовете на клас. Първата е да се предотврати намесата на клиентите във вътрешните механизми на работа на класа и други важни части, които не засягат приложението на класа от гледна точка на приложение на интерфейса му. Това на практика е услуга за потребителите, понеже те не могат да знаят кое е важно да не се пипа и кое може да се променя.

Втората причина е разработчикът на библиотеката да може да променя без да мисли за това как промяната ще се отрази на други програми. Например може да решите да направите някакъв клас за да улесните писането на реализация, а после да решите да го пренапишете, за да се изпълнява по-бързо. Ако интерфейсът и реализацията са внимателно разделени и скрити това може да се направи и от клиента ще се иска само ново свързване на неговите програми.

Java използва три явни ключови думи и една подразбираща се за установяване на необходимото: **public**, **private**, **protected** и подразбиращата се "приятелски," която е в сила, ако не посочите някоя от другите. Тяхната употреба и значение са забележително праволинейни. Тези спецификатори на достъпа определят кой може да използва следващата ги дефиниция. **public** означава, че нещото е достъпно за всеки. **private** от друга страна, означава че всичко е недостъпно освен за вас, създателя на класа. **private** е тухлена стена между вас и програмистът-клиент. Ако някой опита достъп до такъв член, получава грешка по време на компилацията. "приятелски" работи с нещо, наречено "package," начинът за правене на библиотеки в Java. Ако нещо е "приятелско", то е достъпно само в рамките на package. (Затова това ниво на достъп е известно като "package access"). **protected** е също като **private** с това изключение, че наследяващия клас има достъп до **protected** членовете но не и до **private** членовете.

⁴ I'm indebted to my friend Scott Meyers for this term.

Повторно използване на код

След като един клас е написан и изprobван тойще представлява (в идеалния случай) полезно късче код. Това изважда наяве че написването на класа изисква опит и познаване на въпроса каквото не всеки има. Веднък направили го обаче, то плаче да бъде използвано повторно. Повторното използване на код е най-мощния лост, който ООП предоставя.

Най-простия начин за повторно използване на код е прокото използване на обекта, но също може този обект да се постави в друг обект. Наричаме това "създаване на член-обект". Новият клас може да се дъзда от каквото и да са типове обекти, толкова, колкото са необходими за да се получи необходимото поведение. Тази концепция е наречена **композиция**, доколкото се композира нов клас от съществуващи класове. Понякога това се нарича "има" зависимост, както в "колата има багажник".

Композицията дава много гъвкавост. Член-обектиг в новия клас са обикновено **private** и това не дава възможност да се използват направо от потребителя. Това дава възможност да бъдат променяни без да се променя клиентския код. Може да променяте също много обекти по време на изпълнение, което дава голяма гъвкавост. Наследяването, което ще се опише малко по-късно, не дава такава гъвкавост, понеже има ограничения във класовете.

Понеже наследяването е толкова важно за ООП може да се създаде впечатление, че то трябва да се използува навсякъде. Това може да доведе до тромав и излишно сложен дизайн. Вместо това трябва да се търси първо композицията, понеже е по-проста и гъвкава. Ако се възприеме този подход, дизайнът ще стои по-чист. Трябва да има очевидна причина за да се използва наследяването.

Наследяване: повторно използване на интерфейса

Самата концепция за обекти удобен инструмент. Тя позволява да се пакетират данни и код според концепция, така че може да представите идеята с езика на проблемното пространство, а не с идиомите на подлежащата машина. Тези концепции са изразени в първичната идея за даннов тип (използвайки ключовата дума **class**).

Би било жалко, обаче, да се създаде някакъв тип и после да трябва да се създава нов с подобна функционалност. Би било по-добре ако може да вземем съществуващ тип, да го клонираме и да прибавим нужните свойства на клонинга. Това е, което фактически се получава с наследяването, с това изключение, че оригиналният клас (наречен базов или супер или родителски клас) е променен, промененият "клонинг" (наречен извлечен или извлечен или под- или дъщерен клас) също отразява помените. Наследяването се прилага в Java с ключовата дума **extends**. Когато правите нов клас казвате че той **extends** съществуващ клас.

Чрез извлечане се създава нов тип, който не само всички членове на наследения клас (макар и тези които са **private** да са скрити и недостъпни), но - което е по-важно - повтаря интерфейса на базовия клас. Това значи, че всички съобщения, които се разбират от базовия клас се разбират и от извлечения. Тъй като типа на класа е известен от съобщенията, които той разбира, извлечения клас е от същия тип като базовия клас. Тази еквивалентност на типовете чрез интерфейсите им е главния вход към разбирането на ООП.

Тъй като базовия и извлечения клас имат един и същ интерфейс, трябва да има някакъв код, който се изпълнява с интерфейса. Ще рече, трябва да има метод, който се изпълнява при

приемането на дадено съобщение. Ако просто наследите клас без нищо друго да пишете, методите на базовия клас директно минават и в наследения. Тоест извлеченият обект има не само същия тип, но и същото поведение, което не изглежда особено интересно.

Има два пътя да направите извлечения клас различен от този, който се наследява. Първият е много праволинеен: просто пишете нови функции в новия клас. Тези нови функции не са част от интерфейса на базовия клас. Значи ако базовият клас не прави всичко, което искате, просто добавяйте необходимото в новия клас. Тази примитивна употреба за интерфейс понякога е най-добрания начин да се реши проблем. Трябва да се погледне, обаче, по-отблизо възможността базовия клас да се нуждае от тези функции.

Подтискане на функциите на базовия клас

Въпреки че **extends** предполага, че ще се добавят нови функции, това не е непременно вярно. Другия начин да се направи втория клас различен е да се промени поведението на съществуваща функция на базовия клас. Това се означава с подтискане на функцията.

За да се подтисне функцията просто се дава нова дефиниция за нея в новия клас. Казваме: "Използвам същата интерфейсна функция, но искам да прави нещо различно."

Отношенията "е" и "прилича на"

Може да възникне известен дебат относно интерфейса: наследяването да подтиска ли само функциите на базовия клас? Това значи че извлеченият тип е точно същия като на базовия клас, защото има същия интерфейс. Като резултат обект от единия клас може да замести обект от другия клас. Това може да се нарече *чиста субституция*. По усещане това е идеалният начин да се третира наследяването. Често се казва че отношението между базовия клас и извлечения клас в този случай е *е*, понеже може да се каже "кръгът е фигура." Пробата за уместност на наследяването е дали може да се прокара е отношение между двата класа и да му се придае смисъл.

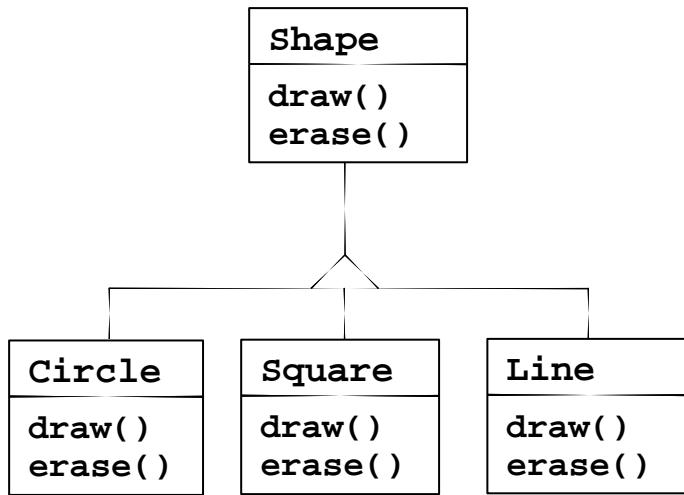
Има случаи, когато се налага да се добави интерфейс в извлечения тип и по този начин да се получи нов тип. Новият тип все още може да замества базовия, но работата не изглежда наред, защото стария тип не може да използва новите функции. Това може да се опише като отношение *е подобен на*; новия тип има интерфейса на стария, но има и нови функции и не може да се каже, че е точно същия. Например един климатик. Да допуснем, че в къщи всичко е нагласено да се управлява охлаждането, т.е. има интерфейс на охлаждането. Да допуснем че климатикът се е развалил и сте го заменили с топлинна помпа, която може да топли и охлажда. Помпата е подобна на климатика, но може повече. Понеже е прокарано управление само за охлаждането, може да се управлява само тази част на новия обект. Интерфейсът на новия обект е бил разширен и системата не знае за това.

Като се погледне принципа на субституцията лесно може да се помисли, че това е единственият начин да се свърши работата и е добре, ако програмата ви ги върши то него. Ще се случи обаче да изглежда също толкова необходимо да се добавят нови функции. С проверка и двата случая трябва да станат приемливо очевидни.

Взаимозаменяеми обекти с полиморфизъм

Наследяването обикновено цели създаване на фамилия класове с един интерфейс. Изразяваме това с диаграма на обрънато дърво:⁵

⁵ This uses the *Unified Notation*, which will primarily be used in this book.



Едно от най-важните неща, които могат да се правят с такава фамилия от класове е, че може да се третира обект от извлечен клас като обект от базов клас. Това е важно, защото значи, че може да се напише единствено парче код, който да игнорира детайлите на типа и да "разговаря" с базовия клас. Кодът е развързан от информацията, специфична за типа и с това е по-лесен за написване и разбиране. И ако нов тип –**Triangle**, например – се добави чрез наследяване, вашият код ще работи също така добре с новия тип (произлязъл от – б.пр.) **Shape** както и със съществуващите типове. Така програмата е разширяема.

Да видим горния пример. Ако напишете функция на Java:

```

void doStuff(Shape s) {
    s.erase();
    // ...
    s.draw();
}
  
```

Тази функция говори с всякакъв обект от рода на **Shape**, така че е независима от спецификата на триенето и чертането. В някаква друга програма използваме функцията **doStuff()**:

```

Circle c = new Circle();
Triangle t = new Triangle();
Line l = new Line();
doStuff(c);
doStuff(t);
doStuff(l);
  
```

Извикванията на **doStuff()** автоматично работят точно, без значение точния тип на обекта.

Това е наистина вълнуващ трик. Да видим линията:

```

doStuff(c);
  
```

Тук става следното: манипулатора на **Circle** е даден на функция, която очаква **Shape** манипулатор. Доколкото **Circle** е **Shape** той може да се третира като едно **doStuff()**. Тоест каквото и да е съобщение, което **doStuff()** може да изпрати на **Shape**, **Circle** може да го възприеме. Така че написаното е напълно сигурно и логично нещо.

Наричаме този процес на третиране на производният клас като базовия *upcasting*. Думата *cast* е използвано в смисъла на леене в калъп и идва от типичния начин на рисуване на дървото на наследяванията – с корена нагоре. Така casting-ът към базовия тип е преместване по диаграмата на наследяванията нагоре: *upcasting*. (някои вероятно биха предпочели да използваме "превръщане към тип" вместо casting -б.пр.)

Една ООП съдържа upcasting някъде, понеже това е начинът по който се развързвате от необходимостта да знаете на конкретния тип с който се работи. Погледнете кода в `doStuff()`:

```
s.erase();
// ...
s.draw();
```

Забележете че той не казва "Ако е **Circle**, прави това, ако е **Square**, прави друго и т.н." Ако се пише код от този тип, в който се проверява винаги точния тип на **Shape**, за да се изпълни нещо, такъв код е тежък и трябва да се променя с всяко добавяне на тип. В този пример просто казва: "То е **Shape**, знае се как да го `erase()`, прави се коректно."

Динамично свързване

Поразителното в `doStuff()` е, че някак си стават правилните неща. Извикването на `draw()` за **Circle** предизвиква изпълнението на код различен от случая на `draw()` за **Square** или **Line**, но когато `draw()` съобщението е изпратено на безименен **Shape**, осигурява се коректно поведение според конкретния вид на **Shape** манипулатора. Това е забележително, защото по време на компилацията `doStuff()` не се знае точния тип. Така че би трябвало да се очаква да се извика `erase()` за **Shape**, `draw()` за **Shape** а не за специфичните **Circle**, **Square** или **Line**. И все пак правилните неща се случват. Ето как става това.

Когато се изпрати съобщение на обект без да се знае точният му вид и се получи всичко както трябва, това се нарича **полиморфизъм**. Процесът, изролзуван от ООП езика за да се постигне това се нарича **динамично свързване**. Компилаторът и run-time поддръжката осигуряват детайлите; всичко, което трябва да се знае е, че това става и по-важното - как да се организира по този начин.

Някои езици изискват употребата на специална ключова дума за да се получи динамично свързване. В C++ тя е **virtual**. В Java няма нужда да се помни подобна дума, понеже винаги се свърза динамично. Така че може да се смята, че всичко ще е както трябва, даже и в случая на upcasting.

Абстрактни базови класове и интерфейси

Често при проектирането се иска да се използува базов клас само заради интерфейса му. Тоест няма да се създават обекти от него, а само ще се наследи от класове, за да се използува интерфейсът му. Това се постига като се направи класът **абстрактен** чрез използването на ключовата дума **abstract**. Ако някой се опитва да направи обект от клас, който е **abstract**, компилаторът попречва на това. Това е инструмент с който се налага определен начин на проектиране.

Също може да се използува **abstract** за да се означи метод, който още не е написан - все едно да се каже "ето (име на - бел.пр.) функция за всички наследници, но в този момент още не съм я написал." Един **abstract** може да бъде зададен само в **abstract** клас. Когато такъв клас е наследен, абстрактният метод трябва да се напише, иначе наследникът става също абстрактен (**abstract**). Създаването на **abstract** позволява да се сложи метод в интерфейса без да се пише (възможно) безполезен код в него.

Ключовата дума **interface** придвижва концепцията за **abstract** клас стъпка напред, предотвратявайки въобще всякакви дефиниции на функции. **interface** е много полезен и често използуван инструмент, понеже дава перфектно отделение на интерфейса и приложението. Освен това може да комбинирате много интерфейси заедно, ако искате. (Не може да наследявате повече от един обикновен **class** или **abstract class**.)

Обекти и техните времена на живот

Технически погледнато, ООП е точно за абстрактните типове данни, наследяването и полиморфизма, но и други неща са също поне толкова важни. Останалата част от тази секция ще се занимава с тях.

Един от най-важните фактори е начинът по който се създават и разрушават обекти. Къде са данните на обекта и как се управлява времето на живот на обекта? Има различни философии на въпроса, които работят. C++ има за най-важни въпросите за управление на ефективността, така че дава на програмиста избор. За максимална скорост на изпълнение още по време на писане на програмата обектите се слагат на стека (понякога са наричани *автоматични* или с *обхват променливи*) или в поле от статична памет. Това слага приоритета на въпросите за алокиране и освобождаване на памет и този контрол може да бъде извънредно ценен в някои ситуации. По този начин, обаче, се убива гъвкавостта, понеже трябва да се знае всичко за обектите *по време на писането на програмата*. Ако се опитвате да решавате по-общ проблем като CAD например, складово стопанство или контрол на въздушния транспорт, това е твърде ограничаващо.

Другият подход е да се създават обектите в област, наречена *heap*. При този подход не се знае точно колко обекти ще има по време на изпълнение, колко е тяхното време на живот и точният им тип. Тези неща се определят по време на изпълнение, както се случи в програмата. Ако трябва нов обект, той просто се създава на хийпа щом стане нужен. Понеже паметта се управлява динамично, значително повече време трябва да се алокира или освободи памет, отколкото в случая, когато тя е върху стека. (Често алокирането на памет върху стека е асемблерска инструкция за запис в стековия указател, респективно освобождаването - запис в указателя променящ го в противната посока.) Динамичният подход има тенденция да са усложнени, така че допълнителното време за отделяне и освобождаване на памет ще е малко в сравнение с общото време на изпълнение. В добавка голямата гъвкавост е основна предпоставка за решаването на общия програмен проблем.

C++ позволява на програмиста да определи дали обектите ще се създават по време на написването на програмата (т.е. по време на компилацията - б.пр.) или по време на изпълнение, позволяйки по този начин управление на ефективността. Би могло да се помисли, че, понеже е по-гъвкаво, обектите трябва винаги да се създават в хийпа, а не на стека. Има обаче и друго нещо и то е времето на живот на обекта. Ако се създаде обект на стека или в статичната памет, компилаторът може да определи времето на живот на този обект и да определи кога да го разруши. Ако обаче се създаде обект в хийпа, компилаторът не знае това. Програмистът ема две възможности за разрушаване на обекта: може програмно да се определи кога да се разрушат обекта или програмната среда може да доставя услугата *събиране на боклук* която автоматично открива кога обектът е вече ненужен и го разрушава. Разбира се, събирачът на боклук е много по-удобен, но се изисква всички програми да са съгласувани с него и има допълнителен разход на ресурси за неговата работа. Това не отговаря на изискванията към езика C++ и затова не е включено, но Java има събирач на боклук (както и Smalltalk; Delphi няма, но може да се добави такъв. Съществуват събирачи за C++ произведени от "странични" доставчици).

Останалата част от тази секция разглежда допълнителни фактори, засягащи обектите и времето им на живот.

Колекции и итератори

Ако не знаете колко обекта ще ви трябвата за решаването на даден проблем или колко време те ще просъществуват вие също не знаете къде да ги сложите. Как да знаете колко

място да отделите? Това не може да стане, понеже информацията ще е достъпна чак по време на изпълнение.

Решението на повечето проблеми в ООП изглежда лекомислено: създавате друг тип обект. Новият обект който решава конкретния проблем съдържа манипулатори на други обекти. Разбира се, може да се направи нещо подобно и с масив, което е достъпно в повечето езици. Но наличното тук е повече. Този нов обект, наречен изобщо **колекция** (също наричан **контейнер**, но Swing GUI библиотеката използва този термин в друг смисъл, така че в тази книга ще се използва "колекция"), ще се разширява от самосебе си за да поеме всичко, което ще решите да сложите в нея. Така че не е необходимо да се знае колко обекта ще се слагат в колекцията. Само се създава обекта колекция и се оставя да се грижи за детайлите.

За щастие един добър ООП език идва с набор добри колекции. В C++ това е Standard Template Library (STL). Object Pascal има колекции в неговата Visual Component Library (VCL). Smalltalk има много завършено множество от колекции. Java също има колекции в стандартната си библиотека. В някои библиотеки общата колекция се счита за добра за всички нужди, в други (C++ в частност) има различни типове колекции за различните нужди: вектор за смислен достъп до всеки елемент, свързан лист за смислена итерация по елементите, например да можете да изберете подходящ тип за своите нужди. Може да включват мрежи, опашки, хеш таблици, дървета, стекове и т.н.

Всички колекции имат начин да слагат неща вътре и да ги вадят навън. Начинът да се сложи нещо во колекцията е очевиден. Има функция наречена "push" или "add" или нещо подобно. Извличането на неща от колекцията не е винаги толкова непосредствено; ако е нещо подобно на масив, като вектор например, може да е възможно да се използува индексиращ оператор или функция. Но в много ситуации това няма значение. Също, функция за избиране само на един елемент е много ограничаваща. Ако искате да работите с или да сравнявате няколко неща в колекцията?

Решението е **итератор**, чието предназначение е да избира елементи от колекцията и да ги представя на потребителя на итератора. Като клас итераторът също дава някакво ниво на абстракция. Тази абстракция може да се използува за разделяне на детайлите на колекцията от кода, който извлича елементите. Колекцията, чрез итератора, се свежда просто до последователност (от елементи -б.пр.). Итераторът позволява да се работи с тази последователност без да се познават детайлите на истинската структура – тоест дали е вектор, свързан списък, стек или нещо друго. Това дава гъвкавостта лесно да се променя подлежащата структура без да се проминя кода на приложната програма. Java започна (във версии 1.0 и 1.1) със стандартен итератор, наречен **Enumeration**, за всичките си класове-колекции. Java 2 добави **Iterator** който прави много повече от стария **Enumeration**.

От гледна точка на проектирането всичко, което е нужно, е последователност, която решава конкретния проблем. Ако една единствена последователност решава проблема, няма нужда от повече. Има две причини за необходимост от избор на колекция. Първо, колекциите дават различни интерфейси и поведение. Стекът има различен интерфейс и поведение от опашката, която е различна от множеството или списъка. Един от тези типове би могло да дава по-добро решение на вашия проблем от другите. Второ, различните колекции имат различна ефективност в различните ситуации. Най-добрият пример са векторът и списъкът. Двете са прости последователности, които могат да имат еднакви интерфейси и поведение. Но някои операции могат да имат много различна цена. Достъпът до случайни елементи от масива е за едно и също време винаги. За свързания списък (лист -б.пр.) такава операция би отнела много време, ако елементът е дълбоко в списъка. От друга страна, вмъкването на елемент някъде по средата е лесно в списъка и отнемащо много време в масива. Тези и други операции имат различна ефективност в зависимост от подлежащата структура. Във фазата на проектирането може да се започне със списък и после, когато се настройва производителността, да се премине към масив. Поради абстракцията чрез итераторите това може да стане чрез минимална промяна на кода.

Накрая, запомнете, че колекцията е просто шкаф от памет за слагане на обекти вътре. Ако шкафът удовлетворява всичките нужди, няма значение как е направен (основна концепция с повечето типове обекти). Ако работите в програмна среда, която има допълнителни разходи на ресурси, сължащи се на други фактори (работата под Windows, например, или цената на събираща на боклук), тогава разликата в ефективността на свързания списък и масива може да няма значение. Може да се нуждаете само от един тип последователност. Може даже да си въобразите "перфектна" абстракция на колекция, която може автоматично да сменя типа в зависимост от използваната подлежаща система.

Йерархия с един корен

Едно от нещата в ООП, което стана доста забележително след въвеждането на C++ е въпросът дали всичко в края на краишата ще е наследник на един единствен клас . В Java (както и в практически всички останали ООП езици) отговорът е "да" и прародителят на всички класове е просто **Object**. Това показва, че ползите от йерархията с един корен са много.

Всички обекти в такава йерархия имат общ интерфейс, така че те в края на краишата са от един тип. Алтернативата (като в C++) е, че не може да се каже, че всичко е от някакъв фундаментален тип. От гледна точка на обратната съвместимост това повече подхожда за преход от C и може да изглежда по-малко ограничаващо, но когато се иска да се направи напълно ООП се налага самостаятелно да се направи същото, което вече го има в другите ООП езици. И каквато и нова библиотека да се вземе, тя ще бъде с някакъв нов несъвместим интерфейс. Това изисква усилието (и може би многократното наследяване) да се вмести новия интерфейс. Заслужава ли си това допълнителната "гъвкавост" на C++ ? Ако се нуждаете от нея – ако имате голяма инвестиция в C – много даже си заслужава. Ако се започвате от нулата, други варианти като Java често могат да бъдат по-перспективни.

All objects in a singly-rooted hierarchy (such as Java provides) can be guaranteed to have certain functionality. You know you can perform certain basic operations on every object in your system. A singly-rooted hierarchy, along with creating all objects on the heap, greatly simplifies argument passing (one of the more complex topics in C++).

Йерархията с един корен прави много по-лесно вграждането на събиращ на боклука. Необходимата поддръжка може да се постави в базовия клас и тогава трябва само събирачът да изпрати съответните съобщения до всички обекти в системата. Без йерархията с един корен и система, която управлява обектите чрез манипулятори е много трудно да се направи събиращ на боклука.

Доколкото информацията по време на изпълнение е във всички обекти, никога програмата не може да завърши с обект, чийто тип не може да се определи. Това е важно специално при системните операции като обработката на изключения и за правене на по-гъвкави програми.

Може да се чудите защо, като е толкова полезна, йерархията с един корен не е представена в C++. Това е старата надпревара между ефективността и управлението. Йерархията с един корен налага ограничения върху проектирането и в частност на съществуващия C код. Тези ограничения представляват проблем само в някои случаи, но като цяло не е наложена йерархията с един корен в C++. В Java, който започна от нулата и нямаше изисквания за обратна съвместимост с никой език, логично бе представена йерархията с един корен, както и в повечето други ООП езици.

Библиотеки от колекции и поддръжка за лесно използване

Тъй като колекциите са нещо, което се използва често, полезно е да има библиотеки от повторно използваеми колекции, от където се взема каквото е подходящо и се използува.

Java има такава колекция, въпреки че едоста ограничена в Java 1.0 и 1.1 (Библиотеката от колекции на Java 2 обаче удовлетворява повечето нужди).

Downcasting и templates/generics

За да бъдат колекциите повторно използвани те съдържат фундаменталния тип в Java който беше споменат по-рано: **Object**. Йерархията с един корен значи, че всичко е **Object**, така че колекция която държи **Object**-и може да държи каквото и да е. Това я прави лесна за повторно използване.

За да се използва такава колекция просто се добавят манипулатори към нея, а след това се изискват обратно. Но, доколкото колекцията съдържа само **Object**-и, когато се добавя манипулатор става upcasting към **Object**, като по този начин се губи идентичността. Когато се взема обратно се полечава манипулатор на **Object**, а не този, който е бил добавен. Как се връщаме към нещото с полезен интерфейс, което сме сложили в колекцията?

Пак се използва casting, но този път не е нагоре към по-общ тип, а надолу по йерархията към по-специфичен тип. Това се нарича *downcasting*. С upcasting, например, се знае, че **Circle** е от типа на **Shape** така че може безопасно да се направи upcast, но не се знае дали **Object** е непременно **Circle** или **Shape** така че манипулаторът не може сигурно да се преобразува, ако не знаете точно с какъв тип имате работа.

Това не е чак толкова опасно, понеже ако се направи опит за неправилно преобразуване, по време на изпълнение ще се получи изключение, което ще опишем накратко. Когато извличате манипулатори от колекцията трябва да има начин да се помни типът им, за да се правят подходящи преобразувания.

Downcasting-ът и проверките по време на изпълнение довеждат до изразходване на допълнително време и допълнителни усилия на програмиста. Би ли имало смисъл да се създаде колекция, която да помни типовете и т.н., та да не се налага всеки път да се прави? Решението е параметризирани типове, които са класове, които компилаторът може автоматично да приспособява за конкретния случай. Например, с параметризирана колекция, компилаторът може да я направи да работи само с **Shape**-ове и за извлича само **Shape**-ове.

Параметризираните типове са важна част от C++, частично защото C++ няма йерархия с един корен. В C++ ключовата дума за прилагане на параметризираните типове е **template**. Java в момента няма параметризирани типове, но те могат да се направят – тромаво, обаче – използвайки йерархията с един корен. От една страна думата **generic** (ключовата дума използвана в Ada за неговите templates) беше в списъка на думите “резервирали за бъдещо приложение.” Някои от тези думи мистериозно се плъзнаха в “Бермудския триъгълник” на клучовите думи и е трудно да се каже какво би могло да се случи.

Домакинската дилема: кой ще чисти?

Всеки обект се нуждае от ресурси за да съществува, от които първа е паметта. Когато обектът вече не е необходим тези ресурси трябва да се освободят за повторно използване. В прости програмни ситуации въпросът колко съществува обекта не изглежда голямо предизвикателство: създавате обекта, той съществува колкото е необходимо и после трябва да се разрушси. Не е много трудно, обаче, да се намерят ситуации в които отговорът е много по-сложен.

Да предположим, например, че се проектира система за управление на трафика за летище. (Същият модел е за управление на палетите в складово стопанство, за видеоленти под наем и за боксовете за пансион на домашни любимци.) Отначало изглежда просто: прави се колекция да сложим аеропланите, после се прави обект и се вкарва в колекцията за всеки

самолет, който влиза в зоната на управление на трафика. За почистване просто се изтрива обекта за всеки аероплан, който напуска зоната.

Но може да има и друга система, която записва данни за самолетите; може би тя не изисква такава бърза намеса като прякото управление на полетите. Може да е запис на маршрутите на всички малки самолети, които напускат летището. Така че имаме втора колекция от малки самолети и винаги, когато се създава обект свързан с аероплан, той се вкарва и във втората колекция, ако самолетът е малък. После някакъв фонов процес върши необходимото в моменти на бездействие на инсталацията.

Сега проблемът е по-тежък: как бихте могли да знаете кога да се разрушат обектите? Когато главният процес е приключил с даден обект, някаква друга част от системата може да не е. Същия проблем може да възникне в множество ситуации и в програмни системи (като C++) в които обектът трябва явно да се унищожи, когато не е необходим, може да стане много сложен.⁶

В Java събирачът на боклук е проектиран да се грижи за освобождаването на паметта (въпреки че това не включва други аспекти от унищожаването на обектите). Събирачът „знае“ когато един обект вече не се използва и автоматично освобождава паметта на обекта. Това комбинирано с факта, че всички обекти са наследници на единствен клас **Object** и че може да се създават обекти по единствен начин: на хийпа, прави процеса на програмиране на Java много по-прост от този на C++. Имате много по-малко решения за вземане и препятствия за преодоляване.

Събирачите на боклук срещу ефективността и гъвкавостта

Ако всичко това е толкова добра идея, защо не са направили и C++ така? Разбира се, има цена за всичкото това програмистко удобство и тази цена е допълнителния разход на ресурси по време на изпълнение. Както се спомена преди, в C++ и в този случай те автоматично се почистват (но нямате гъвкавостта да създавате колкото ви трябват по време на изпълнение). Създаването на обекти на стека е най-ефективният начин за заемане и освобождаване на паметта. Създаването на обекти в хийпа може да бъде много по-скъпо. Наследяването винаги на базовия клас и правенето на всички извиквания на функции полиморфни също събира своя малък данък. Но събирачът на боклук е отделен проблем, защото никога не се знае кога ще се включи и колко ще работи. Това значи, че има неопределено време на реакцията на Java програма, така че не може да я използвате в някои ситуации, където времето на реакция е критично. (Такива ситуации се наричат изобщо *програми в реално време*, макар и не всички изисквания на програмирането в реално време да са толкова строги.)⁷

Проектантите на езика C++, опитвайки се да ухажват С програмистите (и най-успешно – докато го правеха), не искаха да добавят черти, които могат да засегнат скоростта или използването на C++ в каквото и да са ситуации, където С би могъл да бъде използван. Тази цел беше постигната, но на цената на по-голама сложност на програмирането на C++. Java е по-прост от C++, но недостатъкът е в ефективността и понякога в приложимостта. За значителна част от програмните задачи, обаче, Java често е най-добрият избор.

⁶ Note that this is true only for objects that are created on the heap, with **new**. However, the problem described, and indeed any general programming problem, requires objects to be created on the heap.

⁷ According to a technical reader for this book, one existing real-time Java implementation (www.newmonics.com) has guarantees on garbage collector performance.

Обработка на изключенията: работа с грешките

От началото на езечите за програмиране насам обработката на грешките е била винаги един от най-трудните моменти. Тъй като е толкова трудно да се направи добра схема за обработка на грешки, много езици просто игнорират този въпрос, предавайки проблема на разработчиците на библиотеки, които пък вземат половинчати мерки, въвршещи в много случаи работа, но които могат лесно да бъдат провалени, изобщо просто като бъдат игнорирани самите те. Главният проблем на повечето схеми за обработка на грешки е, че те се основават на бдителността на програмиста за спадване на приета схема, която не е наложена от самия език. Ако програмистът не е бдителен, каквито често биват те, когато бързат, схемата лесно може да бъде забравена.

Обработката на изключения връзва обработката на грешки направо в езика и даже операционната система. Изключението е обект, който е "изхвърлен" от частта на грешката и може да бъде "хванат" от подходящ обработчик на изключения проектиран да обработи съответния конкретен тип грешка. Сякаш обработката на изключения е друг, паралелен път, който пресмятанията могат да поемат, ако нещата се сбъркат. И понеже то използва друг път на изпълнение, не е необходимо да си пречи с вашия код, който не се занимава с грешки. Това прави кодирането по-просто, понеже не сте принудени постоянно да проверявате за грешки. В добавка изхвърленото изключение не прилича на върната стойност на грешката от функция или флаг сложен от функция с цел да отбележи наличието на грешка, те могат да бъдат игнорирани. Изключението не може да бъде игнорирано, така че ще му дойде времето в някой момент. Накрая, изключенията дават начин надеждно да се излезе от лошата ситуация. Наместо просто да излезете (да завърши програмата - б.пр.) често може нещата да се оправят и изпълнението да продължи, което довежда до много по-добри програми.

Обработката на изключения в Java изпъква сред останалите езици, понеже тук то е вградено от самото начало и потребителят на езика е принуден да го използва. Ако не си направите програмата да обработва правилно изключения, получавате грешки при компилиация. Тази гарантирана състоятелност прости обработката на грешки много по-лесна.

Нищо не пречи че обработката на изключения не е ОО черта, макар и в ООП езиците изключението да е представено обикновено с обект. Обработката на изключения съществува от преди ООП.

Многонишково изпълнение

Основна идея в компютърното програмиране е изпълняването на повече от една задача едновременно. Много програмни проблеми изискват възможността програмата да може да спре текущото изпълнение, да свърши нещо друго и да се върне към основната си задача. Много подходит е имало за решението. Първоначално програмисти с познаване на машината на ниско ниво писали програми за обслужване на прекъсванията и отклонението от главния процес се започвало с хардуерно прекъсване. Въпреки че работело добре, това било трудно и непреносимо, така че правело преместването на програмата на нов тип машина скъпо и бавно.

Понякога прекъсванията са важни за критични задачи, но в повечето случаи се иска просто задачата да се раздели на конкурентно работещи парчета, които съвместно да я правят по-диалогична. В рамките на програмата тези отделно работещи процеси се наричат **нишки** и основната концепция се нарича **многонишковост**. Общ пример за многонишковост е потребителският интерфейс. Чрез използването на многонишковост потребителят може да натисне бутон и да получи бърз отговор, наместо да чака завършването на цялата работа.

Както обикновено, нишките са само начин да се ангажира време от един процесор. Когато операционната система поддържа повече от един процесор и такива са налични, всяка нишка може да се изпълнява на отделен процесор и паралелно с другите. Едно от удобните неща на мулитрединга е, че програмистът на високо ниво не е необходимо да се грижи дали процесорите са много или само един. Програмата логически се разделя на нишки и ако има повече от един процесор, тя просто работи по-бързо.

Всичко това прави нишките да звучат доста прости. Има клопка: разделяемите ресурси. Ако има повече от една нишка, която се нуждае от даден ресурс, имате проблем. Например не може два процесора едновременно да подават информация към принтер. За да се реши проблемът ресурсите, които трябва да се споделят, като принтера, трябва да бъдат заключени по време на използването им. Така че нишката заключва ресурса, свършва си работата и го отключва, така че друга да може да го използва.

Нишковостта е вградена в Java, което прави сложният проблем много по-прост. Нишковостта се поддържа на обектно ниво, така че нишката се представя с обект. Java също осигурява заключване на ресурси с определени ограничения. Може да заключва паметта на всеки обект (която памет е, в края на краищата, разделяем ресурс) така че само една нишка да я използува едновременно. Това се постига с ключовата дума **synchronized**. Другите ресурси трябва да се заключват явно от програмиста, типично чрез създаване на обект, който представлява ключалката и трябва да бъде пробван преди ресурсът да се използва.

Упоритост

Като се създава обект, той съществува, колкото е необходим, но по никакъв начин след завършването на програмата. Докато на пръв поглед това има смисъл, има и ситуации, в които би било чудесно обектът да си запазва стойностите и когато програмата не работи. При следващо стартиране на програмата такъв обект ще съдържа същата информация, каквато при предишното е получил. Разбира се, това би могло да се постигне чрез писане на файлове, например бази данни, но в цялостния дух на ООП всичко да е обекти просто трябва да се направи обект, който е упорит (понякога - устойчив, б.пр.) и да се грижи за всички детайли вместо програмиста.

Java 1.1 поддържа "лека упоритост," което означава, че лесно може да запомняте и възстановявате обекти. Причината да е "лека" е, че все още трябва явно да се викат функции за запомнянето и възстановяването. В някоя бъдеща версия може да се появи по-пълно обслужване на "упоритост".

Java и Internet

Ако Java е на практика само още един програмен език, би могло да се запита защо е толкова важен и защо е бил прокламиран като революционна стъпка в компютърното програмиране. Отговорът не е непосредствено очевиден, ако се изходи от традиционната програмистка гледна точка. Въпреки че Java може да реши традиционните, самостоятелни програмни проблеми, причината да е толкова важен е, че ще реши и проблеми свързани с World Wide Web.

Какво е Мрежата?

Мрежата може да изглежда мъничко мистериозна на пръв поглед с всичките тези приказки за "сърфинг," "присъствие" и "home pages." Винаги е имало даже постоянно нарастваща реакция срещу "Internet-манията" на съмнение в ползата и доходите от такова помитащо движение. Полезно е да се върнем стъпка назад и да видим точно за какво става въпрос, но за да се направи това, трябва да се разберат клиент/сървър системите, друг аспект на компютърните технологии, изпълнен със смущаващи въпроси.

Клиент/сървър обработки

Основната идея на клиент/сървър системата е, че има централен склад за информация – някакъв вид данни, типично в база данни – която искате да разпределяте по заявки към някакво множество хора или машини. Ключът към клиент/сървър е че складът е централно разположен така че може да бъде променян и промените да бъдат предавани до потребителите навън. Взети заедно информацията, софтуерът който я предава навън и другият спомагателен софтуер заедно с машината (или машините) също представлява сървъра. Софтуерът който работи на отдалечената машина и комуникира със сървъра, приема информацията, обработка я и я изписва на дисплея на отдалечената машина се нарича клиент.

Основната идея на клиент/сървър обработките, значи, не е чак толкова сложна. Проблемите се появяват понеже има един сървър, който трябва да обслужва много клиенти наведнъж. Изобщо е налична и система за управление на база данни, така че проектантът "балансира" плана на данните в таблица за оптимална употреба. В добавка системите често позволяват клиентът да добавя данни към наличните. Това значи, че трябва да се осигури незастъпване на данните на различните клиенти, а също и че няма да се загубят в процеса на добавянето им. (Това се нарича *обработка на транзакции*.) Тъй като клиентският софтуер се променя и той трябва да бъде написан, тестван и внедрен, това излиза по-скъпо и трудно, отколкото може да се мисли на пръв поглед. Особено проблематично е поддържането на различни платформи. Накрая, въпросът с изпълнението: може да има стотици клиенти със заявки по едно и също време и даже малкото забавяне да е критично. За да се намали времето за отговор, програмистите работят здраво да облекчат задачите, често на клиентската машина но понякога на другите машини откъм сървърската страна използвайки т.н. *middleware*. (Мидълуерът също се използва за да се подобри управляемостта.)

Така простата идея за разпределяне на информация се оказва с толкова много нива на сложност, че като цяло проблемът може да изглежда безнадеждно енigmатичен. И следното също е критично: сметки за клиент/сървър обработки за грубо половината от всичката активност на машините. Те са отговорни за всичко: от приемането на поръчки до разпределението на всяка вид данни – борсови, научни, правителствени – каквато ги вие. Преди се правеше индивидуално решение на единичен проблем, всеки път наново. Беше трудно за произвеждане и трудно за използване и потребителят трябаше да учи всеки интерфейс отделно. Като цяло проблемът клиент/сървър се нуждае от генерално решение.

Мрежата е огромен сървър

Фактически Мрежата е една огромна система клиент/сървър. Мъничко по-зле е, понеже имате всичките сървъри и клиенти съвместно съществуващи на една връзка едновременно. Това обаче не ви интересува, понеже имате да се свържете и да обменяте данни само с един сървър в даден момент (макар че може да подскочате по целия свят търсейки подходящия сървър).

В началото беше прост еднопосочен процес. Правеше се заявка към сървъра и той изпращаше файл, който машинният броузър (т.е. клиентът) интерпретираше, форматирали го подходящо на локалната машина. Но на бърза ръка хората поискаха да правят нещо повече от просто вземане на файлове. Те искаха пълна клиент/сървър възможност така че да може да се връща информация към сървъра, например да се преглеждат базите данни на сървъра, да се добавя информация на сървъра, да се правят поръчки (което изисква повече сигурност отколкото първоначалната система предлагаше). Това са нещата които виждахме при развитието на Web.

Web броузера беше голяма стъпка напред: концепцията, че парче информация може да се изобразява на всяка вид компютър. Броузерите обаче бяха все още малко примитивни и бързо затънаха в нарастващите изисквания към тях. Те не бяха много интерактивни и имаха тенденция да спъват и Мрежата, и сървъра, понеже за всяко нещо, което изискваше програмиране трябаше да се обръщат към сървъра. Можеше да отнеме много секунди само за да се установи, че нещо е сбъркано при набирането на команда. Доколкото

броузера беше само за изобразяване на информация той не можеше да извършва никакви други обработки. (От друга страна така беше по-сигурно, понеже не можеше да се изпълни програма на локалната машина, която има грешки или вируси.)

Различни подходи се приемаха при решаването на този проблем. Колкото да се започне, графиката беше разширена с по-добра анимация и вградено видео. Останалата част от проблема може да бъде решена само с интегриране на възможност да се изпълняват програми на клиентската страна, чрез броузера. Това се нарича *програмиране на клиентската страна*.⁸

Програмиране на клиентската страна⁸

Първоначалният дизайн на Web броузерите включваше интерактивност, но тя изцяло се доставяше от сървърите. Сървърът произвеждаше статични страници за броузера, който просто ги извеждаше на екрана. Основата на HTML съдържа някои неща за събиране на данни: полета за въвеждане на текст, за избор, "радио" бутона, списъци и падащи списъци, къто и бутон, който бе програмиран само да изчисти данните или да ги "изпрати" на сървъра. Това изпращане минава през *Common Gateway Interface* (CGI) налично на всички Web сървъри. Текстът който се изпраща информира CGI какво да прави със самия него. Най-типичното нещо е стартиране на програма на сървъра, разположена най-често в директория "cgi-bin." (Ако следите адресния прозорец на вашия броузър когато натиснете бутон на Web страница можете понякога да видите "cgi-bin" сред всичкото, което минава там.) Тези програми могат да бъдат написани на повечето езици. Perl е общ избор, понеже е създаден за операции със стрингове и се интерпретира, така че може да бъде поставен на всеки сървър независимо от операционната система и типа на машината.

Много от мощните Web сайтове днес са изградени изцяло с CGI и практически нищо не може да се направи с тях. Проблемът е времето за отговор. Отговорът на CGI зависи от това колко данни се препращат и от натоварването както на сървъра, така и на Мрежата. (Отгоре на това стартирането на CGI програма има тенденция да бъде бавно.) Първоначалните проектанти на Web не предвиждаха колко бързо честотната лента ще се заеме от създадените от хората приложения. Например никакъв вид динамична графика не е смислено възможен, понеже GIF файл трябва да бъде създаден и изпратен за всяка междинна поза. Несъмнено и вие сте запознати с толкова простото нещо като валидиране на данни във форма. Натискате бутона за изпращане; данните отиват до сървъра; сървърът стартира CGI която открива грешка, произвежда HTML страница която информира за грешката и я изпраща обратно при вас; трябва тогава да извикате формата и да започнете пак. Това не само е бавно, то не е елегантно.

Решението е програмиране в клиентската страна. Много машини на които работят Web са мощни и способни да вършат обширна работа и с оригиналния статичен HTML подход те просто си седяха, чакайки сървърът да издуха поредната страница. Програмирането на клиентската страна значи, че Web броузерът е впрегнат да върши всяка работа която може и резултатът за потребителя е много по-бърза и приятна работа с неговия Web сайт.

Проблемът на дискусиите за сайтовете е, че той не е по-различен от дискусиите за програмирането изобщо. Парабетрите са почти същите, но платформата е друга: Web броузърът прилича на ограничена операционна система. Накрая, това си е програмиране и за негова сметка е замайващото поле от проблеми, свързани с програмирането на клиентската страна. Останалата част от тази секция дава общ поглед върху проблемите и подходите за решаването им що се отнася до програмирането на клиентската страна.

⁸ The material in this section is adapted from an article by the author that originally appeared on Mainspring, at www.mainspring.com. Used with permission.

Plug-ins

Една от най-съществените стъпки в развитието на програмирането на клиентската страна е развирането на plug-in. Това е начин за програмиста да включи нова функционалност в броузера чрез сваляна (при броузера - б.пр.) на парче код, което автоматично се включва само (от там и името - б.пр.) към определена точка от страницата. То казва на броузера "от сега нататък може да изпълняваш (и) нова дейност." (Необходимо е да се свали плуг-инът само веднъж.) Бързо и мощно поведение беше въведено в броузерите по този начин, но писането на плуг-инове не е тривиална задача и не е нещото, което бихте искали да правите, занимавайки се с определена страница. Ценното на плуг-иновете е, че експерт-програмист може да напише и внедри нов програмен език за броузера без разрешение от производителя на броузера. Така плуг-иновете дават задна врата за нови езици за програмиране на клиентската страна (макар че не всички езици са намерили приложение като плуг-инове).

Скриптове и езици за тях

Плуг-иновете докараха експлозия на скриптовете. Със скрипта вграждате директно в HTML страницата сорс код за програмиране на клиентската страна и плуг-инът автоматично се активира щом HTML страницата се визуализира. Скриптовите езици имат тенденция да бъдат доста прости за разбиране и понеже това е просто текст от HTML страницата се товарят бързо при простото доставяне на страницата. Недостатъкът е, че кодът е изложен пред всички да го гледат (и крадат) но изобщо казано не се правят вълнуващи сложни неща със скриптове така че това не е кой знае колко тежко.

Това идва да покаже, че със скриптовете се е възнамерявало да се реши определен клас проблеми, най-напред създаването на по-богати и интерактивни потребителски интерфейси (GUI-си). Скриптът обаче може да реши 80 от срещаните в програмирането на клиентската страна. Вашите проблеми може точно да се включват в тези 80 и доколкото скриптовете имат тенденция да са бързи и лесни за разработка, сигурно ще се спрете на скрипт преди да мислите за нещо като Java или ActiveX програмиране.

Най-общо обсъжданите скриптови езици са JavaScript (което няма нищо общо с Java; наречен е така за да грабне от пазарната инерция на Java), VBScript (който изглежда като Visual Basic) и Tcl/Tk, който идва от популярния междуплатформен език за постройка на потребителски интерфейси. Има и други извън нашия списък и без съмнение още повече се разработват.

JavaScript е вероятно най-широко поддържания. Той идва с всеки Netscape Navigator и Microsoft Internet Explorer (IE). В добавка, вероятно има повече книги за JavaScript отколкото за който и да е друг език и някои инструменти автоматично създават страници използвайки JavaScript. Ако обаче вече знаете добре Visual Basic или Tcl/Tk, би било по-продуктивно да се използват те, отколкото да се учи нов език. (Ще бъдете достатъчно заети с проблемите в Web в него момент.)

Java

Ако скриптът може да реши 80 от проблемите при програмирането на клиентската страна, какво става с останалите 20% – "наистина трудните?" Най-популярното решение днес е Java. Не само че е напълно развит програмен език построен за да бъде сигурен, междуплатформен и интернационален, но Java непрекъснато бива развиван за да предостави черти и библиотеки които елегантно се справят с проблемите, които са трудни в традиционните езици, като многонишковост, достъп до бази данни, мрежово програмиране и разпределени обработки. Java позволява програмиране на клиентската страна чрез *applet*.

Аплетът е малка програма, която работи само от Web броузер. Аплетът автоматично се сваля заедно със страницата Web (точно както например картинките биват сваляни (на клиентската машина -б.пр.)). Когато аплетът се активира той се изпълнява като програма. Това е част от

неговата красота – дава се възможност да се снабдяват потребителите със софтуер от сървъра когато е необходим и не по-рано. Потребителите получават последната версия без проблеми и уморителни преинсталации. Поради начина по който Java е проектиран програмистът трябва да произведе единствена програма и тази програма автоматично работи с всички компютри които имат вградени Java интерпретатори. (Това сигурно включва преобладаващата част от съществуващите машини.) Понеже Java е пълноценен език, може да се върши колкото си искате работа както преди, така и след контакти със сървъра. Например не е необходимо да пращате поръчка през Мрежата само за да откриете, че сте събрали данните или нещо друго не е наред и вашият клиентски софтуер може да си види данните вместо да чака сървърът да ги прегледа, да построи съответната страница и да я изпрати. Не само че има непосредствен резултат откъм скорост и интензивност на работата, но и мрежовия трафик към сървърите се разтоварва, предотвратявайки забавянето на цялата Мрежа.

Едно от предимствата които има Java аплетът пред скрипта е, че той е в компилирана форма, така че сорсът не е достъпен за потребителя. От друга страна, един Java аплет може да бъде декомпилиран без особени грижи и скриването на кода не е чак толкова важно обикновено. Два други фактора са важните. Както ще видите по-късно в тази книга един компилиран Java може да включва няколко модула и да изисква няколко "hits" (достъпа) до сървъра за да бъде разтоварен. (В Java 1.1 това е минимизирано чрез Java архивите, наречени JAR файлове, които позволяват всички необходими модули да бъдат пакетирани за разтоварване на един път.) Скриптова програма ще бъде вградена в Web страницата като част от нейния текст (и изобщо ще бъде по-малък и ще намали нитовете на сървъра). Това може да бъде важно за комуникативността на вашата страница. Друг фактор е във всяко отношение важната крива на научаването. Без значение какво сте слушали, Java не е тривиален за изучаване език. Ако сте Visual Basic програмист минаването към VBScript би било по-бързото решение и доколкото ще се решат повечето от най-типичните задачи за клиент/сървър изучаването на Java може да е много трудно за оправдаване. Ако сте опитен в скриптовете сигурно ще спечелите ако надникнете към JavaScript или VBScript преди да отидете към Java, понеже те може и да са достатъчни и вие ще бъдете по-продуктивен по-скоро.

ActiveX

До някаква степен съперник на Java е ActiveX на Microsoft, макар че подходът там е съвсем друг. ActiveX по начало е само за Windows, макар че сега е развит от независим консорциум като междуплатформен. Фактически ActiveX казва "ако вашата програма се свързва с програмното обкръжение точно еди как си, тя може да бъде в Web страница и да въвви на броузър който поддържа ActiveX." (IE директно поддържа ActiveX и Netscape го поддържа използвайки plug-in.) Така ActiveX не ви ограничава до определен език. Ако например вече сте опитен Windows с език като C++, Visual Basic, или Delphi на Borland, може да създадете ActiveX компоненти практически без промени във вашите програмистки знания. ActiveX също дава начин за използване на съществуващ от по-рано код в Web страници.

Сигурност

Автоматичното разпространение на програми по мрежата може да звучи като мечта на човек, правещ вируси. ActiveX специално разглежда трънливия въпрос за сигурността на клиентското програмиране. Ако щракнете на Web бихте могли да свалите автоматично много неща заедно с HTML страницата: GIF файлове, скрипт, компилиран Java код и ActiveX компоненти. Някои от тях са лесни; GIF не могат да причинят никаква вреда и скриптовете са обикновено ограничени, до това, което могат да вършат. Java също бе проектирана да пуска своите аплети извътре на "санитарна кутия" от сигурност, което не позволява да пише по диска или да има достъп до памет извън "кутията".

ActiveX е на другия край на спектъра. Програмирането с ActiveX е като програмиране на Windows – може да направите всичко, което поискате. Така че ако щракнете на страница, която съдържа ActiveX компонента, тази компонента може да причини повреждането на

файлове на вашия диск. Разбира се, програмите които пускате на вашия компютър и които не са ограничени в рамките на броузър могат да направят същото. Вирусите свалени от Bulletin-Board Systems (BBSи) отдавна са проблем, но скоростта на Мрежата усилва трудността.

Изглежда решението е "цифрови подписи," които сигурно показват кой е авторът. Това е базирано на идеята, че вирусът работи защото неговият автор може да остане анонимен, та като се махне анонимността индивидите ще могат да бъдат държани отговорни за действията си. Това изглежда добър план, понеже програмите могат да бъдат много по-функционални и аз подозирам, че би могло да премахне злонамерната пакостливост. Ако обаче програмата има неизвестен бъг, това може все пак да предизвика проблеми.

Подходът в Java е да се предпазим от възникването на такива проблеми чрез "кутията". Java интерпретаторът който живее във вашия локален Web броузър преглежда аплета за "опаки" инструкции докато аплетът току-що е свален. В частност аплетът не може да пише файлове по диска или да траси файлове (едни от главните места на стоеще на вирусите). Аплетите изобщо се предполага да бъдат сигурни и тъй като това е основно за всички надеждни клиент/сървър системи всички бъгове, където би могло да има вируси се оправят бързо. (Не е кой знае какво че броузерите фактически нарушават изискванията за сигурност и че някои позволяват различни нива на сигурност.)

Може би сте скептични относно тази драконовска мярка относно писането на файлове по вашия диск. Например може да искате да си направите локална база данни за използване офлайн. Първоначалното виждане беше всеки да върши всичко онлайн, но скоро се видя, че това не е практично (макар че ниските цени на трафика и устройствата може да задоволят нуждите на повечето потребители). Решението е "подписан аплет" който използва публичен ключ за да докаже, че идва наистина от там, от където се твърди. Подписаният аплет може после да продължи право напред и да ви строши диска, но теорията е, че сега можете да държите автора на аплета отговорен за такива злонравни работи. Java 1.1 дава рамка за дигиталните подписи така че може да предприемете стъпки извън кутията ако е необходимо.

Цифровите подписи са пропуснали важен момент и това е скоростта, с която хората се разхождат в Мрежата. Ако сте свалили лоша програма, колко ще трябва време за да откриете това? Може би дни и даже седмици. И от тогава нататък как ще откриете коя точно програма е направила белята (и какво ако я откриете чак тогава?).

Internet срещу Intranet

Web е най-общото решение на проблема клиент/сървър така че има смисъл да използвате същите прийоми за решаване на всеки подпроблем, в частност класическия клиент-сървър проблем в рамките на компанията. С традиционните клиент/сървър подходи се явява проблемът с различните типове на клиентските компютри, както и трудността на инсталирането на клиентския софтуер, като и двата добре се решават чрез Web броузърите и програмирането на клиентската страна. Когато Web технологията се използва в информационна мрежа в рамките на една компания това се нарича *Intranet*. Интранет дава много по-голяма сигурност от Мрежата, тъй като достъпът до сървърите във вашата компания може физически да се контролира (т.е. да се държат заключени - бел. прев.). В термините на обучението: научаването веднъж на броузерите и евентуалните неголеми промени очевидно предполага по-голяма производителност на обучението, отколкото в другата ситуация.

Проблемът със сигурността ни води към едно естествено разделение в света на клиент/сървър програмирането. Ако програмите ви ще се изпълняват в Мрежата, трябва да бъдете извънредно внимателни с тях, понеже не се знае на каква платформа ще се изпълняват. Нуждаете се от нещо многоплатформено и сигурно, като скрипт или Java.

В интранет ограниченията ще бъдат други. Не е необикновено всичките машини да са от Intel/Windows платформата. В интранет сте отговорни за ваш собствен код и бъговете могат да бъдат оправени когато се открият. В добавка може да имате наследен код в голямо количество от традиционния клиент/сървър подход в който трябва физически да инсталирате програмите при всяко осъвременяване. Времето за осъвременяване е най-непреодолимата

причина за минаване към броузъри, понеже с тях то става прозрачно и безболезнено. Ако имате такъв тип мрежа, най-подходящо е да минете към ActiveX вместо да прекодирате всичките си програми в нов език.

Изправени пред такова объркващо разнообразие от алтернативи в клиент/сървър програмирането, намираме, че най-добрят план за атака е анализът на разходите. Имайте пред вид ограниченията на вашата инсталация и търсете най-бързия път до решението. Тъй като програмирането на клиентската страна е програмиране, винаги е добра идеята да се предприеме най-бързият подход. Това е агресивна стойка при срещите с неизбежните усложнения при програмирането на клиентсървър системите.

Програмиране при сървъра

В цялата досегашна дискусия игнорирахме програмирането на сървъра. Какво става, когато му се поръча нещо? Повечето време поръчките са просто "изпрати ми този и този файл." Вашият броузер после интерпретира файла по подходящ начин: като HTML страница, графика, Java аплет, скрипт и т.н. Една по-обща поръчка към сървъра би могла да бъде транзакция с база данни. Типичния сценарий включва поръчка за сложно търсене в база данни, като резултатите сървърът формира в HTML и ги изпраща като резултат. (Разбира се, ако клиентът има повече интелигентност чрез Java или скрипт, необработените резултати могат да бъдат изпратени към клиента и там да се форматират, което би било по-бързо и с по-малко натоварване на сървъра.) Или бихте могли да искате да си впишете името в базата данни, което ще доведе до нейната промяна. Тези операции се извършват от някакъв код на сървърската страна, като създаването му изобщо се нарича *програмиране при сървъра*. Традиционно това програмиране се изпълнява на Perl и CGI скриптове, но се появиха и по-усложнени системи. Те включват Java-базирани Web сървъри, които позволяват цялото програмиране на сървърската страна да се извърши на Java чрез написването на това, което се нарича *servlets*.

Отделна арена: приложенията

Повечето от шумотевицата около Java е за аплетите. Java е език за програмиране в на-общия смисъл на думата и може да реши повечето програмистки проблеми, най-малкото на теория. И както беше посочено преди, може да има по-ефективни пътища за решаване на клиент/сървър проблемите. Когато излезете от арената на аплетите (като същевременно се освободите от ограниченията като това за писане по диска) влизате в света на общото програмиране с програми, които работят сами, без Web броузър, точно както всяка обикновена програма. Тук силата на Java е не само в преносимостта, но също така и в програмируемостта. Както ще видите през цялата тази книга, Java има много черти, които дават възможност да се страйт програмни проекти за по-късо време, отколкото с досега известните езици.

Трябва да сте предупредени, че това е двулична благословия (както повечето на този свят и така трябва да бъде -б.пр.). Плаща се за напредъка с по-бавно изпълнение (макар и да се върши значителна работа в тази посока). Като всеки език Java има присъщи ограничения, които го правят неподходящ за решаването на определен кръг от проблеми. Java е бързо развиващ се език, обаче, и с всяка нова версия става все по привлекателен за все по-разширяващо се множество от проблеми.

Анализ и проектиране

Парадигмата на обектно-ориентираното програмиране е нов начин на мислене за програмирането и много хора отначало се затрудняват как да подхождат към проектите си. Сега като знаете че всичко се предполага да бъде обекти вие можете да създадете "добър" дизайн, такъв който ще донесе всички предимства които се предполага да дава ООП.

Повечето от книгите посветени на ООП са пълни с дълги думи, тромава проза и важно звучащи декларации.⁹ Аз не споделям този начин мислейки че книгата би била по-добра като глава или най-много като много кратка книга и дразнейки се от схващането че този процес не може да се опише стегнато и директно. (Смущава ме, че хората, които претендират да управляват сложността се затрудняват да напишат приста книга.) Най-после, цялата работа с ООП е да се направи процесът на софтуерна разработка по-лесен и макар това да заплашва може би поминъка на хората, които консултират по въпросите на сложността, защо все пак нещата да не се направят по-прости? Така, надявайки се да съм възбудил здравословен скептицизъм у вас аз ще се стремя да ви предам моето виждане относно анализа и проектирането в колкото е възможно по-малко параграфи.

СТОИМ НА КУРСА

По време на процеса на разработка най-важното е: не се загубвай. Лесно е да се направи. Повечето от тези методологии са направени да решават най-големите проблеми. (Това има смисъл; тези са наистина много трудни проблеми които оправдават наемането на съответния автор като консултант и съответните му големи цени.) Запомнете че повечето проекти не попадат в тази категория, така че можете да имате успешен анализ и проектиране с малко подмножество от цялата методология в повечето случаи. Но винаги някакъв анализ и проектиране биха ви довели по-бързо до искания резултат, отколкото просто да седнете и да започнете да програмирате.

Ще рече, че ако вземете методология, която има огромно количество детайли и изиска съответното количество документация, трудно е все още да се каже къде да спрете. Помните какво се опитвате да откриете:

1. Какво са обектите? (Как разделяте проекта си на части?)
2. Какви са техните интерфейси? (Какви съобщения трябва да можете да изпращате на всеки обект?)

Ако се окаже че имате само обекти и интерфейси, може да започвате писането на програмата. Поради много причини може да ви трябват повече документи от споменатите, но трудно може да минете с по-малко.

Процесът може да се проведе на четири фази и фаза 0 е просто да се реши първоначално каква ще бъде общата структура.

Фаза 0: Да направим план

Първата стъпка е да решите какви стъпки ще има във вашия процес. Звучи просто (фактически всичко това звучи просто) и ето, често, хората даже не обикалят до фаза едно преди да започнат писането на програмите. Ако вашият план е "да скочим и да почнем кодирането," чудесно. (Понякога така е добре - когато имате добре разбран проблем.) Най-малкото съгласен съм, че това е план.

Бихте могли да решите също, че някаква допълнителна структурираност на проекта е необходима, но не прекалено велика. По достатъчно разбираме причини някои програмисти предпочита да работят в режим на "ваканция" където не се предполага никаква структура на процеса на тяхната работа: "Ще стане когато стане." Това може да бъде провлекателно за малка, но аз съм открил, че наличието на няколко километрични камъка по пътя помага да се фокусират и циментират усилията по посока на тях, отколкото единствено до целта "завършване на проекта." В добавка това разбива процеса на по-смилаеми парчета и го прави по-малко заплашителен.

⁹ The best introduction is still Grady Booch's *Object-Oriented Design with Applications*, 2nd edition, Wiley & Sons 1996. His insights are clear and his prose is straightforward, although his notations are needlessly complex for most designs. ([You can easily get by with a subset.](#))

Когато аз започнах да изучавам приказката структура (така че някой ден ще напиша новела) първоначално се съпротивлявах на идеята, чувствайки че когато пиша просто написаното се излива върху страниците от само себе си. Това, което открих бе, че когато пиша за компютри структурата е сравнително проста и не изисква специално внимание, но аз все пак структурирах моята работа все пак, само полуусъзнателно в моята глава. Така че даже и да мисбите, че плана ви е веднага да започнете кодирането, все пак минавате следващите фази задавайки си някои въпроси и отговаряйки им.

Фаза 1: Какво ще произвеждаме?

В предишното поколение програмно проектиране (процедурното проектиране), това би било наречено “създаване на анализ на изискванията и спецификация на системата.” Това, разбира се, бяха местата, където да се загубиш: заплашително наименовани документи които са способни да вземат голям проект в своя собственост. Те бяха писани с добри намерения, обаче. Анализът на изискванията казва “Направи списък на правилата, по които ще се водим за да свършим работата и задоволим потребителя.” Системната спецификация казва “Ето описанието какво програмата ще прави (не как) за да се задоволят изискванията.” Анализът на изискванията е в действителност договор между вас и потребителя (даже и потребителят да работи във вашата компания той е друг обект в системата). Системната спецификация е изследване на върхното ниво на проблема и в някакъв смисъл откриване как проектът да се направи и за колко време. Доколкото и за двете ще е необходим консенсус в персонала, мисля, че е добре да се държат колкото се може по-прости – в идеалния случай като списъци и диаграми – за да се спести време. Може да има и други изисквания, които да ви накарат да ги разпрострете в по-големи документи.

Необходимо е фокусът да остане върху това, което искате да направите в тази фаза: да определите какво ще се иска от системата да прави. Най-ценното в тази посока е колекция от т.н. “случай на използване.” Това са основно описателни отговори на въпроси които почват с “Какво прави системата ако ...” Например “Какво прави автоматичния (програмен -б.пр.) отговарящ ако клиентът е направил депозит в рамките на 24 и няма достатъчно пари в сметката за да му се изплати подаден чек?” Случаят на използване описва подробно какво ще се прави по-нататък.

Опитвате се да откриете пълната система на случаи на използване на програмата и когато веднъж го постигнете сте се сдобили със сърцевината на това, което системата (целевата -б.пр.) се предполага, че ще прави. Хубавото на събирането на случаите е, че те винаги ви връщат към основните неща и ви предпазват от отклонения към неща, които не са критично важни. Тоест ако имате пълната система случаи вие можете да опишите вашата система и да преминете към следващата фаза. Вероятно няма да я получите перфектно очертана в тази фаза, но това е нормално. Всичко ще се разбули от самосебе си с течение на процеса на разработка и ако вие искате перфектно описание още на този етап ще се си отворите излишна работа.

Както крачния стартер на мотоциклет помага в тази фаза, ако изложите нещата в няколко параграфа и проследите за глаголи и съществителни . Съществителните стават обекти и глаголите - методи в интерфейсите на тези обекти. Ще бъдете изненадани колко полезен инструмент може да бъде това; понякога то ще свърши лъвската част от работата вместо вас.

Въпреки че едва започваме, в тази точка никакво разпределение на времето вече е полезно. Вече имате поглед върху това, което се строи така че вероятно ще може да направите оценка колко дълго ще продължи. Много фактори важат тук: ако изчислите дълго време за реализация компанията може да не приеме проекта или управителят може вече да е решил колко време ще трае и да ви се намеси в разписанието. Но най-добре е да си направите честно разписанието и да вземете трудните решения рано. Много опити е имало да се излезе с акуратни разчети на времето (подобни на техниките на предвиждане на борсовия пазар), но може би е най-добре да се осланяте на опита и интуицията си. След като сте направили оценка, която смятате за добраудвоете я и добавете 10%. Вашето чувство вероятно

е точно; вие можете да получите нещо работещо за това време. "Удвояването" ще го направи прилично и десетте процента ще са за доизкусуряване и детайли на края. Обаче вие искате да го обясните и независимо от стоновете и манипулациите които ще се чуват, всичко ще стане за това време.

Фаза 2: Как ще го направим?

На края на тази фаза трябва да имате проект от който да личи как ще изглеждат обектите и как ще взаимодействват помежду си. Удобен продукт с дияграми е *Unified Modeling Language* (UML). Може да вземете спецификация за UML от www.rational.com. UML може да бъде също полезен като инструмент за описание по време на фаза 1 и някои от диаграмите които сте съставили там вероятно ще продължат модифицирани във фаза 2. Не е необходимо да използвате UML, но това може да бъде полезно, особено ако възнамерявате да окачите диаграми по стените за обсъждане от всички, което е добра идея. Алтернатива на UML е текстово описание на обектите и техните интерфейси (както описах в *Thinking in C++*), но това може да бъде ограничаващо.

Най-успешното консултиране което съм давал беше на тим, който никога преди това не беше правил ООП проект, като рисувах обекти на черната дъска. Обсъждахме как обектите ще взаимодействват помежду си, триехме някои и рисувахме нови. Тимът (те знаеха какво се очаква да прави проектът) фактически създаде дизайна; те "притежаваха" дизайна наместо той да им е спуснат отвън. Всичкото което аз правех беше да водя процеса чрез задаване на точните въпроси, правех предположения и имах обратната връзка с тима за корекцията им. Красотата на процеса беше в това, че тимът се научи да проектира ОО не с разглеждане на абстрактни примери, а чрез работа над проект, който беше най-интересният за тях към него момент: техния.

Ще знаете че сте свършили фаза 2 когато имате описането на обектите и техните интерфейси. Е, повечето от тях – има по няколко, които се промъкват и стават ясни във фаза 3. But that's OK. Трябва да откриете всички ваши обекти в края на краишата - това само ви засяга. Добре е да ги откриете рано, но ООП дава достатъчна база да ги включите и покъсно, ако ги откриете тогава.

Фаза 3: Да строим!

Ако четете тази книга вероатно сте програмисти, така че сега сме вече в частта, до която искате да стигнете. Чрез следване на план – без значение колко прост и кратък – и построяване на структурата на проекта преди кодирането ще откриете че двете неща се постигат заедно по-лесно, отколкото ако се гмурнете веднага в програмирането, и това дава голямо удовлетворение. Възнаграждаващо е да имаш желания код, който прави точно каквото се иска от него, даже като от друга, както става при определено излизашо от употреба поведение на някои програмисти. Моята практика обаче показва, че удовлетворението от елегантно решен проблем доставя удоволствие от съвсем друго ниво; то по-прилича на изкуство, отколкото на технология. И елегантността винаги се изплаща; тя не е лекомислено преследвана. Не само че дава програма, която е по-лесно да се построи и изтества, но също е по-лесна за управление и разбиране и именно от там идва финансовата ценност.

След като построите (компилирате и свържете - б.пр.) системата, важно е да се направи реалистичен тест и тук е мястото където системния анализ и спецификация влизат в работата пак. Минете по цялата програма и проверете дали всичко е наред и работи както се обавка по списъка на случайте. Сега сте готови. Дали сте готови?

Фаза 4: Итерация

Това е моментът в цикъла на програмното осигуряване, който традиционно се нарича "поддръжка," термин с всевъзможни значения който може да значи от "да се накара да върви по начина, който отначало се искаше от него" до "добавяне на черти, които потребителят забравил да спомене по-рано" и до по-традиционното "оправяне на грешки, които забавят" и "добавяне на нови черти както изискват появяващи се нужди." Толкова много смесени концепции са прилагани към термина "поддръжка" че той се взема малко за развалящ качеството, частично защото казва че сте направили завършена програма и че всичко, което трябва да правите е да сменяте части, да я смазвате и да я пазите от ръжда. Може би има по-добър термин за описание на това, което става.

Терминът е *итерация*. Тоест: "Не можахте да го направите точно от първия път, затова се върнете и си дайте труда да го проучите и подобрите." Може да се наложат много промени като навлизате по-дълбоко в проблема. Елегантността, която постигате при това ще се изплаща и в краткосрочен, и в дългосрочен план при това.

Какво значи "да се оправи" е - точно каквото програмата не прави точно според изискванията. Значи също, че вътрешната структура на програмата да има смисъл за вас и да създава чувството че всичко се сработва добре, без тромав синтаксис, прекалено големи обекти и несърчно изложени на показ парченца код. В добавка трябва да чувствате, че структурата ще оцелее след всички промени, които несъмнено ще има през време на живота на програмата, а и че такива промени ще стават лесно и евтино. Това не е малко постижение. Трябва не само да разбирате построеното от вас, но също и как програмата ще се развива (което аз наричам *вектор на промяната*). За щастие ООП езици са специално пригодни за този тип непрекъснати промени – границите създадени от обектите са това, което има тенденция да пази системата от сриване. Те също позволяват да правите промени, които изглеждат драстични при процедурното програмиране без да предизвикват земетресение. Фактически поддръжката на итерацията може би е на-печелившата част от ООП.

С итерацията може най-малкото да постигнете някакво приближение на желаното, след това спирате, сравнявате го с изискванията и набелязвате нови промени. После се връщате обратно и оправяте нещата с използване на парчета код.¹⁰ Може да имате в действителност да решавате проблем, или аспект на проблема, няколко пъти преди да ударите правилното решение. (Изучаването на *Design Patterns*, описани в глава 16 обикновено е полезно в това отношение.)

Също е итерация, когато построяте система, видите че удовлетворява изискванията и после откриете, че те не са били подходящи. Когато видите системата забелязвате, че сте искали да решите друг проблем. Ако смятате, че този вид итерация е на път да се съдне, ще гледате да направите първата версия колкото може по-набързо за изясняване след това какво всъщност се иска.

Итерацията е близка до *инкрементното разработване*. Инкрементно разработване е когато започнете с най-централната част на вашата система и после я използвате за среда за развитие, в която се разработва кодът парченце по парченце. После започвате да добавяте черти по цяла една наведнъж. Трикът е да се проектира среда, която лесно да получи всички черти, които ще искате после. (Вж. глава 16 за по-вътрешно разглеждане на това.) Предимството е, че когато веднъж подкарате началната система, всяка нова черта изглежда като малък проект, а не като част от голям проект. Също е и по-лесно да се провежда поддръжката. ООП поддръжка инкременталното разработване, понеже ако програмата ви е разработена добре, промените ще са нови обекти или групи обекти.

¹⁰ This is something like "rapid prototyping," where you were supposed to build a quick-and-dirty version so [that](#) you could learn about the system, and then throw away your prototype and build it right. The trouble with rapid prototyping is that people didn't throw away the prototype, but instead built upon it. Combined with the lack of structure in procedural programming, this often leads to messy, [expensive-to-maintain](#) systems [that are expensive to maintain](#).

Планирането се изплаща

Не може да се построи къща без множество грижливо начертани планове. Ако строите кучешка колибка вашите планове вероятно няма да са толкова подробни на сигурно ще започнете с нещо като скици, които да ви водят в работата. Софтуерната разработка е отишла към крайности. Дълго време хората нямаха структура на разработките си, но тогава големите проекти започнаха да се провалят. Като реакция се появиха методологии със сплашващо количество правила и данни. Те пък бяха тежки за използване - изглеждаше, че цялото време трябва да отиде за писане на документация, а не за програмиране. (Често тъкъв именно беше случаят.) Надявам се, че това което съм казал тук сочи средния път – една подвижна скала. Използвайте подход който отговаря на нуждите (и индивидуалността) ви. Без значение колко минимален, някакъв ще донесе голямо предимство пред липсата на всяка къв план. Помните че според някой оценки повече от 50% от проектите се провалят.

Java или C++?

Java доста прилича на C++ и естествено е да изглежда, че C++ ще бъде заместен от Java. Аз обаче се съмнявам в тази логика. От една страна C++ все още има някои черти, които Java няма и въпреки че имаше много обещания че Java някой ден ще бъде бърз колкото C++ или даже по-бърз, пробив още няма (много се е ускорил, но не се докосва до бързината на C++). Изглежда също че има наперен интерес към C++ в много области, така че не допускам, че C++ скоро ще отпадне. (Езиците изглежда са непредсказуеми. Говорейки на един "Intermediate/Advanced Java Seminars," Allen Holub твърдеше, че двата най-повсеместно използвани езика са Rexx и COBOL, в този ред.)

Започвам да мисля, че силата на Java е в малко по-различна област от тази на C++. C++ е език, който не се опитва да запълни определен калъп. Той е приспособен по множество начини да решава конкретен клас проблеми. Някои пакети комбинират библиотеки, инструменти за генерация на код за да постигнат произвеждане на код с прозоречен интерфейс (за Microsoft Windows). И какво ползват повечето разработчици на програми за Windows? Visual Basic (VB) на Майкрософт. Това е напук на факта че VB така произвежда кода, че програмата става неудобна за поддържане вече при няколко страници дължина (и синтаксис който може да бъде обект на митове). Колкото и успешен и популярен да е, VB е от гледната точка на проектирането камара кирки. Би било хубаво да имаме простотата и мощта на VB без резултантния неуправляем код. И там мисля че Java ще блесне: като "следващ VB." Може да трепвате или да не тръпнете като чувате това, но помислете си върху него: така много от Java е посветено на улесняването на програмиста в решаването на въпроси като мрежовото програмиране и UI, а все още той има структура която позволява да се продуцират големи количества добре управляем код. Добавете към това факта че Java има най-добрите проверки на типа и обработка на грешки които съм виждал в програмен език и имате предпоставките за голяма крачка напред в продуктивността на програмирането.

Да използвате ли Java вместо C++ за вашия проект? Освен Web аплетите има две неща да се вземат под внимание. Първо, ако искате да използвате множество съществуващи библиотеки (и сигурно ще получите много полза за продуктивността така) или ако имате съществуваща C или C++ маса код, Java може да снижи скоростта на проектирането вместо да я повиши. Ако започвате от нулата, простотата на Java спрямо C++ ще намали времето за разработка.

Най-големия въпрос е скоростта. Интерпретирания Java е бавен, даже 20 до 50 пъти по-бавен от C в оригиналните интерпретатори на Java. Това е подобрено малко с времето, но все още си остава важно число. Компютрите са за скорост; ако не беше важно нещо да става бързо, щяхте да го правите на ръка, а не с компютър. (Даже съм чувал да съветват да се започне с Java, за да се постигне късо време за разработка, после да се използва

енструмент и поддържащи библиотеки за да се мине на C++, ако е необходима по-голяма скорост на изпълнение.)

Ключът за направата на Java звероятен кандидат за повечето не-Web проекти е подобряване на скоростта чрез т.н. “just-in time” (JIT) компилатори и даже чрез компилатори на кода към машинен език (два вече съществуват към времето на написване на тази книга). Разбира се, компилаторите до естествен за машината код премахват желаната преносимост, но също и повишават скоростта до близка до тази на С и C++. И кроскомпилирането на Java трябва да е много по-лесно отколкото при С или C++. (На теория само прекомпилирате, но това обещание е давано по повод други езици.)

Може да намерите сравнения на Java и C++, обзори за Java реалностите и практичесността и правила за кодиране в приложенията.

2: Всичко е обект

Макар да е основан на C++, Java е "по-чист" обектно-ориентиран език за програмиране.

И C++ и Java са хибридни езици, но в Java проектантите не са чувствали хибридизацията така важна както в C++. Хибридния език допуска няколко стила на програмиране; причината C++ да е хибриден е че поддържа обратна съвместимост с C. Понеже C++ е надмножество на C езика, той включва много от нежеланите свойства на този език, което го прави в много случаи C++ натрупано сложен.

В Java се предполага, че ще се правят само обектно ориентирани програми. Това значи че преди да започнете трябва да превключите начина си на мислене към ОО свят (освен ако не сте го вече превключили). Печалбата от това предварително усилие е, че получавате възможност да работите с език, който е по-лесен за научаване и използване отколкото много ОО езици. В тази глава ще видим основните компоненти на една програма на Java и ще научим че всичко в Java е обект, даже самата Java програма.

Обектите манипулираме с манипулатори

Във всеки език се разбира различно нещо под "манипулиране на данните". Понякога програмистът трябва непрекъснато да внимава каква манипулация протича в момента. Да ли го манипулирате директно или индиректно чрез нещо, (указател в C или C++) което изисква специален синтаксис?

Всичко това е опростено в Java. Всичко се третира като обекти, затова има единен, винаги смислен синтаксис, който се използва навсякъде. Въпреки че всичко се третира като обект, идентификаторът с който работите е "манипулатор" на обект. (Може да го срещнете като *reference* или даже указател на други места, където се обсъжда Java.) Може да си представим сцената като телевизор (обектът) с дистанционно (манипулаторът). Докато държите манипулатора, можете и да управлявате обекта и когато някой каже "смени канала" или "намали звука," това, на което въздействате е манипулаторът, а той манипулира обекта. Ако искате да се разхождате из стаята, вземате със себе си дистанционното, а не телевизора.

Освен това дистанционното може да остане само, без телевизор. Тоест ако имате манипулатор това не значи, че непременно има обект, който той манипулира. Така ако искате да вместите дума или изречение създавате манипулатор на **String**:

```
| String s;
```

Създали сте само манипулаторът, не обект. Ако решите да отправите съобщение към **s** сега, ще получите грешка (по време на изпълнение) понеже **s** не е прикрепено към нищо (няма телевизор). Една по-сигурна практика е винаги да инициализирате, когато създавате:

```
| String s = "asdf";
```

Тук е използван специален случай: стринговете може да бъдат инициализирани с текст в кавички. Нормално се използват по-обобщени начини за инициализация на обекти.

Трябва вие да създадете всичките обекти

Когато се създава манипулятор искаме да го свържем с някакъв обект. Изобщо това се прави с ключовата дума **new**. **new** казва, "Направи ми нов обект от този вид." Така че горният пример може да бъде и:

```
| String s = new String("asdf");
```

Това не само значи "Направи ми **String**," но също дава информация как да се направи **String** чрез даването на началния стринг.

Разбира се **String** не е единственият съществуващ тип. Java идва преситен с готови типове. По-важното е, че може да създавате и свои типове. В това се състои основната активност в Java и за това ще учим в останалата част от книгата.

Къде живее паметта

Полезно е да се онагледи как стават някои неща докато се изпълнява програмата и в частност как се аранжира паметта. Има шест различни места за запомняне на данни:

1. **Регистри.** Това е най-бързата памет понеже е на различно място от обикновената: вътре в процесора. Регистрите са обаче жестоко ограничен ресурс и затова се запазват от компилатора за неговите нужди. Нямате директен контрол и програмите не виждат, че има регистри.
2. **Стекът.** В общата RAM (памет с произволен достъп) е, но директно се поддържа от процесора чрез неговия стеков указател. Указателят се увеличава за да освободи памет и намалява, за да я създаде. Това е извънредно бърз и ефективен начин да се създаде памет, отстъпващ само на регистрите. Java компилаторът трябва да знае докато компилира програмата точната дължина и време на живот на всичко, което е на стека, понеже трябва да създаде код, който да мърда указателя нагоре и надолу. Тези ограничения слагат граници за вашата програма така че макар и да има Java памет на стека – специално, манипулятори на обекти – Java обектите не се слагат на стека.
3. **Хийп.** Това е памет за обща употреба (също в RAM паметта) където всички Java обекти живеят. Хубавото е, че за разлика от случая със стека, компилаторът няма нужда да знае дължината на обектите и времето им на живот. Така има голяма гъвкавост при използването на хийпа. Щом ви потрябва обект, пишете код да го създаде чрез **new** и памет се алокира щом кодът се изпълни. Разбира се има цена за тази гъвкавост: повече време трябва за да се алокира памет в хийпа.
4. **Статична памет.** "Статична" е използвана тук в смисъл на "в определено място" (макар че е също в RAM). Статичната памет съдържа данни които са налични през цялото време на изпълнение на програмата. Може да използвате ключовата дума **static** за да посочите че даден елемент на обект ще е в тази памет, но самите Java никога не се слагат в статичната памет.
5. **Памет за константи.** Константите често се слагат в програмния код, където е по-сигурно, че никога няма да се променят. Понякога може да се сложат и в памет само за четене (ROM).

6. **Не-RAM памет.** Ако едни данни живеят напълно извън програмата, те могат да съществуват и когато програмата не е стартирана, без нейното управление. Два начални примера са *streamed objects*, където обектите са превърнати в поток от байтове, основно за да се изпратят към друга машина, и упоритите обекти (или устойчиви - б.пр.), в който случай обектите се запомнят на диска и остават и след като програмата е спряна. Трикът е и в двата случая да се използва енергонезависима памет, а после отново всичко да се прехвърли в обекновени RAM-базирани обекти когато е необходимо. Java 1.1 поддържа лека упоритост, а бъдещите версии на Java може би ще дават по-завършени решения на този въпрос.

Специален случай: първични типове

Има една група типове, които се третират специално; може да ги мислим като "първични" типове, които твърде често се използват при работа. Причината за специалното третиране е, че да се създаде нов тип с **new**, особено ако е малка, проста променлива, не е много ефективно, понеже **new** слага обектите на хийпа. За тези типове Java се връща на подхода използван в С и С++. Тоест, заместо да се създаде променлива чрез **new**, "автоматична" променлива се създава която не е манипулатор. Променливата съдържа стойността и се слага на стека, така че е много по-ефективна.

Java определя дължината на всеки първичен тип. Тези дължини не се променят при минаване от една машина на друга както в много други езици става. Тази инвариантност е една от причините Java програмите да са толкова преносими.

Първичен тип	Дължи на	Минимум	Максимум	Обхваща щ тип
<code>boolean</code>	1-bit	–	–	Boolean
<code>char</code>	16-bit	Unicode 0	Unicode $2^{16}-1$	Character
<code>byte</code>	8-bit	-128	+127	Byte ¹
<code>short</code>	16-bit	-2^{15}	$+2^{15}-1$	Short ¹
<code>int</code>	32-bit	-2^{31}	$+2^{31}-1$	Integer
<code>long</code>	64-bit	-2^{63}	$+2^{63}-1$	Long
<code>float</code>	32-bit	IEEE754	IEEE754	Float
<code>double</code>	64-bit	IEEE754	IEEE754	Double
<code>void</code>	–	–	–	Void ¹

Всички числови типове са със знак, така че не търсете беззнакови типове.

Примитивните типове данни също имат "обхващащи" ги класове. Това значи, че ако искате да създадете някой от примитивните типове в хийпа ще използвате асоциирания обхващащ тип. Например:

```
char c = 'x';
Character C = new Character(c);
```

или също може:

```
Character C = new Character('x');
```

Причините да се прави така ще се изяснят в друга глава.

¹ In Java version 1.1 only, not in 1.0.

Числа с висока точност

Java 1.1 добави два класа за аритметика с висока точност: **BigInteger** и **BigDecimal**. Макар и приблизително да подхождат за категорията “обхващащи” класове, никой от тях няма аналог-примитив.

И двата класа имат методи, които дават аналоги на операциите с примитивни типове. Тоест може да направите всичко с **BigInteger** or **BigDecimal** каквото можете с **int** или **float**, само дето трябва да използвате методи наместо оператори. Също, понеже повече неща се правят, операциите са по-бавни. Обменяте точност за скорост.

BigInteger поддържа цели (числа) с произволна точност. Това значи че може да представяте произвольни числа и да пресмятате без загуба на значещи цифри.

BigDecimal е за числа с фиксирана запетая с произволна точност; може например да го използвате за финансови изчисления които са напълно акуратни.

За детайли относно конструкторите и методите на тези класове вижте документацията.

Масиви в Java

Практически всички програмни езици поддържат масиви. Използването им в C и C++ е опасно, понеже повечето масиви са просто блокове памет. Ако програмата използва памет извън блока или го използва неинициализиран (чести програмни грешки) ще има непредсказуеми резултати.²

Една от основните цели на Java е сигурността, така че много от проблемите които вадят душата на програмистите в C и C++ липсват в Java. Масивът в Java се гарантира да бъде инициализиран и не може да бъде достъпен извън границите си. Проверката на индексите и т.н. по време на изпълнение довежда до малко повече разходи, но презумпцията е, че си струва заради сигурността.

Когато създавате масив от обекти фактически създавате масив от манипулатори и всеки от тях се инициализира със специална ключова дума `null` означаваща една специална стойност: **null**. Когато Java вижда **null** той разбира, че съответния манипулатор няма асоцииран обект. Трябва да асоциирате обект с всеки манипулатор преди да го използвате и ако се опитате да го направите докато стойността му е още **null** проблемът ще изскочи по време на изпълнение. По този начин типичните грешки са пресечени в Java.

Може също да се създаде масив от примитиви. Отново компилаторът гарантира същите неща, както и по-горе.

Масивите ще се разглеждат в детайли в следващите глави.

Никога не се налага да разрушавате обект

В повечето програмни езици въпросът за времето на живот на обектите (в широкия смисъл на думата - б.пр.) заема голяма част от общото програмистко усилие. Колко дълго ще трае дадена променлива? Ако трябва да се разрушь, кога да се направи това? Трудностите с времетраенето на променливите довеждат до множество грешки и тази секция показва как Java съществено опростява въпроса, почиствайки всичко вместо вас.

² [In C++ you should often use the safer containers in the Standard Template Library as an alternative to arrays.](#)

Обхват

Повечето процедурни езици имат концепцията за обхват. С нея се определят както видимостта, така и времето на живот на променливите (в дадения обхват - б.пр.). В C, C++ и Java обхватът се определя чрез поставяне на големи скоби {}. Така например:

```
{  
    int x = 12;  
    /* само x достъпно */  
    {  
        int q = 96;  
        /* x и q достъпни */  
    }  
    /* само x достъпно */  
    /* q "извън обхвата" */  
}
```

Променлива определена в даден обхват е достъпна само до края на същия обхват.

Индентацията прави Java кода по-лесен за четене. Тъй като Java е език със свободна форма, табулациите и краишата на редовете не променят програмата по същество.

Забележете че не може да се прави следното, макар и да би могло в C и C++:

```
{  
    int x = 12;  
    {  
        int x = 96; /* Неприемливо!! */  
    }  
}
```

Компилаторът ще каже, че **x** вече е било дефинирано. Така C и C++ възможността да се "скрие" променлива в по-голям обхват не е позволено, понеже проектантите на Java са считали, че това ще доведе до проблемни програми.

Обхват на обекти

Java нямат същото време на живот както примитивите. Като създадете Java обект с **new** той остава и след края на обхвата. Така при:

```
{  
    String s = new String("a string");  
} /* край на обхвата */
```

манипуляторът **s** изчезва на края на обхвата. **String** обектът към който сочеше **s** обаче още заема памет. В това парче код няма начин вече да се достигне обекта, понеже манипуляторът му е извън обхвата. В други глави ще научите как може да се управлява обектът през цялата програма.

Това извежда наяве че тъй като обектите създадени с **new** остават толкова, колкото ги искате, камарата програмистки проблеми просто изчезва в C++ и Java. Най-тежките проблеми изглежда да възникват в C++ понеже няма никаква помощ от езика да се определи дали обектите са още необходими. И по-важното, в C++ трябва да осигурите унищожаването на обектите когато сте си свършили работата с тях.

Това извиква интересен въпрос. Ако Java оставя обектите неразрушени, кое предпазва от препълването на паметта и спирането на програмата? Това е точно проблемът, който би възникнал в C++ (ко програмистът не се грижи достатъчно, вж. предния абзац - б.пр.). Малко магия се случва. Java има боклучар, който гледа всички обекти създадени с **new** и определя

към кои от тях вече няма обръщания от никъде. Тогава освобождава паметта на тези обекти, така че да се използва от други обекти. Това значи, че никога не се грижите за самата памет. Просто създавате обекти и когато те не са нужни вече, от само себе си изчезват. Това елиминира определен клас от програмистки проблеми: т.н. "изтичане на памет," където програмистът е забравил да освободи паметта.

Създаване на нови типове данни: class

Ако всичко е обект(и), как да определим поведението и изгледа на даден клас обекти? Другояче казано, как да определим типа на обект? Може би очаквате ключова дума "тип" и тя сигурно би имала смесъл. Исторически обаче повечето ООП езици са използвали думата **class** за да означат "Ще ви кажа на какво ще прилича новия обект." Ключовата дума **class** (която толкова много се използва навсякъде, че даже не е с уделени букви в книгата) се следва от името на новия тип. Например:

```
| class ATypename { /* тялото на класа ще е тук */ }
```

Това въвежда нов тип, така че може да създадете обект с **new**:

```
| ATypename a = new ATypename();
```

В **ATypename** тялото на класа е само един коментар (звездите и наклонените черти и това, което е между тях) така че не много може да се направи с него. Фактически не можете да му кажете да прави повече от нищо (тоест, не може да му изпратите никакво интересно съобщение) докато не му определите методи.

Полета и методи

Когато определяте клас (всичко което правите в Java е определяне на класове, правене на такива обекти и изпращане на съобщения до тях) може да сложите два типа елементи в класа: членове-данни (понякога наричани **полета**) и членове-функции (типовично наричани **методи**). Членъ-данни е обект (с когото комуникирате чрез манипулатора му) от какъвто и да е тип. Може да бъде и някой примитив (и тогава няма да има манипулатор). Ако е манипулатор на обект, трябва правилно да го инициализирате да сочи към обект, преди да го използвате (с **new**, както видяхме по-рано) в специална функция наречена **конструктор** (описана напълно в глава 4). Ако е първичен тип може да го инициализирате направо в точката на дефинирането му. (Както ще видите по-късно, манипулаторите също могат да се инициализират в точката на определянето им (другояче - декларацията им, б.пр.).)

Всеки обект има собствена памет за своите членове-данни; Те не се споделят от различните обекти. Ето пример за клас с членове данни:

```
class DataOnly {  
    int i;  
    float f;  
    boolean b;  
}
```

Този клас *не прави* нищо, но може да се създаде такъв обект:

```
| DataOnly d = new DataOnly();
```

Може да се присвояват стойности на членовете-данни, но най-напред трябва да научите как се означава член на обект. Това се прави с точка между името на обекта и члена (**objectHandle.member**). Например:

```
| d.i = 47;  
| d.f = 1.1f;  
| d.b = false;
```

Възможно е също обектът да съдържа други обекти, които искате да модифицирате. За целта продължавате с точките. Например:

```
| myPlane.leftTank.capacity = 100;
```

Класът **DataOnly** не може да прави повече от това да съдържа данни, понеже няма член-функции (методи). За да се разбере как те работят първо трябва да се разберат аргументите и връщаните стойности, които ще опишем накратко.

Стойности по подразбиране за примитивни членове

Когато примитивен даннов тип е член на клас гарантира се, че ще получи стойност по подразбиране, ако не го инициализирате:

Първичен тип	Подр. стойност
Boolean	false
Char	'\u0000' (null)
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0d

Внимателно отбележете, че стойностите по подразбиране се гарантират от Java когато променливата се използва като член на клас. Това осигурява членовете-примитиви да са винаги инициализирани (нещо което C++ не прави), намаляват се източниците на грешки.

Тази гаранция не се отнася за "локалните" променливи – онези, които не са полета в клас. Така е ако имате вътре в дефиниция на функция:

```
| int x;
```

Тогава **x** ще приеме каквато се случи стойност (както в С и C++); няма да бъде автоматично инициализирана с нула. Вие сте си отговорни за присвояването на подходяща стойност на **x** преди използването му. Ако забравите, Java определено изпреварва C++: имате грешка при компилация с предупреждение, че променлива може да не е инициализирана. (Много C++ компилатори биха ви предупредили за това, но в Java това е грешка, а не предупреждение.)

Методи, аргументи и връщани стойности

До сега терминът функция бе използван за означаване на именувана подпрограма. По-обичайния термин в Java е метод, както в "начин да се направи нещо." Ако искате, може да продължите да мислите в термините на функции. Разликата е само синтактична, но от сега нататък в книгата ще се идползва "метод" а не "функция."

Методите в Java определят съобщенията, които обектът може да приема. В тази секция ще научите колко просто е да се създават методи.

Основните части на метода са името, аргументите, връщаната стойност и тялото. Ето основната форма:

```
returnType methodName( /* списък аргументи */ ) {  
    /* Тяло на метода */  
}
```

Връщаният тип е типът на променливата, която изскача от метода след неговото извикване. Името, както се досещате, го идентифицира. Списъкът аргументи показва нещата, които се дават на метода и типа им.

В Java методи могат да се създават само като част от клас. Метод може да се вика само за обект³ и този метод трябва да може да осъществи извикването. Ако се опитате да извикате неправилен метод на обектще получите грешка при компилация. Методът се вика подобно както се означават член-променливите, както тук: **objectName.methodName(arg1, arg2, arg3)**. Да предположим например че имате метода **f()** който не приема аргументи и връща **int**. Тогава ако имате обект наречен **a** за който **f()** може да бъде извикан, това може да стане така:

```
int x = a.f();
```

Типът на връщане трябва да е съвместим с типа на **x**.

Актът на извикване на метод общоприето се означава като *изпращане на съобщение към обект*. В горния пример съобщението е **f()** и обектът е **a**. ОО програмиране често в резюме се изразява с “изпращане съобщения на обекти.”

Списъкът аргументи

Списъкът аргументи на метода задава каква информация може да му се дава при извикване. Както можете да познаете тази информация – както всичко друго в Java – има формата на обект. Такаче трябва да зададете вида на обектите и имената, които ще се използват. Както във всяка ситуация в Java с обекти фактически се предават манипулатори.⁴ Типът на манипулатора трябва да бъде точен, обаче. Ако аргументът се предполага да е **String** това, което давате трябва да е стринг.

Да кажем че един метод взима за аргумент стринг. Ето какво трябва да се сложи в дефиницията на класа, за да бъде компилирано:

```
int storage(String s) {  
    return s.length() * 2;  
}
```

Този метод казва колко байта трябват за да се вмести информацията от конкретен стринг **String**. (Всеки **char** в **String** е 16 бита, или два байта дълъг, за да поддържа Unicode знаци.) Аргументът е от тип **String** и е наречен **s**. Щом **s** е даден на метода, може да го третирате като всеки друг обект. (Може да му изпращате съобщения.) Тук **length()** метода се извиква, който е един от методите на **Strings**; той връща броя на знаците в стринга.

Може да видите също използвана ключовата дума **return** която прави две неща. Първо тя значи “излез от метода, свършихме.” Второ, ако методът произвежда стойност, тази стойност се слага точно след **return** оператора. В този случай връщаната стойност се дава с **s.length() * 2**.

³ **static** methods, which you'll learn about soon, can be called *for the class*, without an object.

⁴ With the usual exception of the aforementioned “special” data types **boolean**, **char**, **byte**, **short**, **int**, **long**, **float**_{ss} and **double**. In general, though, **you're passing you pass** objects, which **means you're really means you pass passing handles to objects**.

Може да се връща всякакъв тип, но ако не щете да се връща нищо, показвате го с декларация на метода като **void**. Ето примери:

```
boolean flag() { return true; }
float naturalLogBase() { return 2.718f; }
void nothing() { return; }
void nothing2() { }
```

Когато типът на връщане е **void**, ключовата дума **return** се използва само да се излезе от метода и затова не е необходима, ако стигнете края на метода. Може да излезете от всяка точка на метода, но ако сте дали тип на връщане не-**void** компилаторът ще осигури верния тип независимо къде излизате.

В този момент може да изглежда, че програмата е само обекти, чито методи вземат за аргументи други обекти и това е всичко. Това разбира се е повечето от работата, но в следващата глава ще научите как да вършите работата на ниско ниво, вземайки решения в методите. За тази глава изпращането на съобщения стига.

Построяване на Java програма

Има няколко други неща които трябва да разберете, преди да видите своята първа Java програма.

Видимост на имената

Управлението на имената е проблем във всеки програмен език. Ако използвате име в даден модул и друг програмист го използва в друг модул, как ще ги различните и предотвратите "сблъскването"? В С това особено е проблем, понеже често програмите са неуправляемо море от имена. C++ класовете (на които са базирани Java класовете) вместо тях са функциите в класовете ("манглинг" на имената - б.пр.) така че не може да се получи сблъсък с имената в друг клас. Обаче C++ позволява и глобални данни и глобални функции, така че то все още е възможно. За да реши проблема C++ въвежда *namespaces* използвайки допълнителни ключови думи.

Java може да избегне проблема чрез свеж подход. За да произведе недвусмислено име за библиотека, използвания спецификатор прилича на домейново име в Мрежата. Фактимески създателите на Java искат да използвате домейновото си име отзад напред, понеже тези имена са уникални. Щом имената са уникални, то и библиотеката **BruceEckel.com** може да има идентична структура като **com.bruceeckel.utility.foibles**. След обърнатата дамейново име точките представляват поддиректории.

В Java 1.0 и Java 1.1 домейновите разширения **com**, **edu**, **org**, **net** и т.н. са с главни букви по договаряне, така че имената ще са същите: **COM.bruceeckel.utility.foibles**. При разработването на Java 2 обаче открито че това създава проблеми и сега цялото име на библиотеката е с малки букви.

Този механизъм в Java означава, че всичките ви файлове имат отделни пространства на имената и всеки клас във файл има уникален идентификатор. (Имената на класове в един файл трябва да са уникални, разбира се.) Така че не се налага да учите специални черти на езика за да го осигурите - механизъмът го осигурява заради вас.

Използване на други компоненти

Щом поискате да използвате предварително дефиниран клас във вашата програма, компилаторът трябва да знае къде да го намери. Разбира се класът би могъл да съществува в същия сурс, откъдето се използва. В този случай просто го използвате, даже и да е

определен по-късно (от мястото на използване - б.пр.) във файла. Java премахва проблема "forward referencing" така че няма защо да мислите за него.

Как ще е ако класът е определен в друг файл? Мже да си помислите, че компилаторът е достатъчно умен да го потърси и намери, но има проблем. Да допуснем, че искате клас с определено име, но дефиниция за клас с такова име има в повече от един файл. Или още по-лошо, да допуснем че пишете програма и слагате в библиотека клас, чието име се дублира с името на клас, който вече е в библиотеката.

За да се реши въпросът трябва да се премахнат всички потенциални двусмислия. Това се прави чрез указване на Java точно кои класове искате чрез ключовата дума **import**. **import** казва на компилатора да вземе *package*, което е библиотека от класове. (В други езици би съдържала също и данни и функции, но помним, че в Java всичко е в рамките на класове.)

Повечето пъти ще използвате класове от стандартните библиотеки на Java които идват с компилатора. С тях няма нужда да се беспокоите с дълги, обърнати домейнови имена; просто пишете, например:

```
| import java.util.ArrayList;
```

за да кажете на компилатора че искате класа на Java **ArrayList**. Обаче **util** може да съдържа множество класове и вие да искате да ги използвате без да ги заявявате един по един. Това лесно се прави чрез използване на '*' за индикация:

```
| import java.util.*;
```

По-общоприето е да се импортира колекция от класове по този начин, отколкото отделни класове.

Ключовата дума **static**

Нормално като създавате клас описвате как обектите ще изглеждат и какво поведение ще имат. Фактически нищо не става, докато не създадете клас с **new** и чак след това са достъпни променливите и методите.

Има две ситуации, където този подход е недостатъчен. Едната е когато искате да има единствени данни за много обекти, без значение колко има създадени и даже ако няма. Другата е ако ви трябва метод, който не е асоцииран с никой клас. Тоест метод, който може да се вика ако няма създадени класове. Може да постигнете и двата ефекта с ключовата дума **static**. Когато напишете че нещо е **static** това значи че нещото не е свързано с никакъв конкретен екземпляр (обект - б.пр.) на този клас. Така че и да не сте създали обект може да викате **static** метод или да използвате **static** данни. С обикновените, не-**static** данни и методи трябва да създадете обект и да използвате именно неговите данни и методи, понеже за не-**static** данни и методи трябва да се знае с кой точно обект работят. Разбира се, тъй като **static** методите не изискват да е създаден обект за да бъдат използвани, те не могат *направо* да имат достъп до не-**static** членове и методи чрез просто извикване без споменаване на имената им (тъй като не-**static** членовете и методите трябва да са свързани с конкретен обект).

Някои ОО езици използват термините *class data* и *class methods*, което значи че данните и методите са на категорията на класа, а не на отделен обект. Понякога литературата за Java използва същите термини.

За да се направи член-променлива **static**, просто се слага ключовата дума пред дефиницията. Примерът прави **static** член и го инициализира:

```
| class StaticTest {  
|     static int i = 47;
```

```
| }
```

Сега и да направите два **StaticTest** обектаще има една и съща памет за **StaticTest.i**. За двата обекта **i** е общо. Нека:

```
| StaticTest st1 = new StaticTest();
| StaticTest st2 = new StaticTest();
```

В тази точка и **st1.i** и **st2.i** имат една и съща стойност 47 понеже я вземат от едно и също място в паметта.

Има два начина за споменаване на **static** променлива. Както е по-горе, може да е с името на обект, **st2.i**. Може също и направо чрез името на класа, което не може да се направи с не-статичните. (Това е предпочтитаният начин, доколкото подчертава статичната природа на **static** променливата.)

```
| StaticTest.i++;
```

Операторът **++** инкрементира променливата. В тази точка и **st1.i** и **st2.i** ще имат стойност 48.

Подобна логика се прилага към статичните методи. Може да се обръщате към тях чрез обекта, както и към всеки метод или да използвате допълнителния синтаксис **classname.method()**. Статичният метод се дефинира по подобен начин:

```
| class StaticFun {
|     static void incr() { StaticTest.i++; }
| }
```

Може да се види че **StaticFun** методът **incr()** инкрементира **static** данната **i**. Може да се извика **incr()** по типичния начин, чрез обект:

```
| StaticFun sf = new StaticFun();
| sf.incr();
```

Или, понеже **incr()** е статичен метод, чрез името на класа:

```
| StaticFun.incr();
```

Докато **static**, приложен към член-данни определено променя начина по който се създават данните (една за всички класове срещу една за всеки обект в не-**static** случая), приложена към методи не е толкова драматична промяната. Важно приложение на **static** за методи е да позволи да се използват методи без създаване на обект. Това е съществено, както ще видим, при създаването на **main()** метода, който е входната точка за стартиране на програмата.

Като всеки метод и **static** методът може да създава и използва обекти от неговия си тип, така че **static** методът често се използва за "овчар" на стадото от екземпляри от неговия си тип.

Вашата първа Java програма

Ето я накрая програмата.⁵ Тя извежда информация за системата на която работи използвайки различни методи на **System** обекта от стандартната библиотека на Java. Забележете, че допълнителен вид коментиране е въведено тук: ‘//’, което значи коментар до края на реда:

```
// Property.java
import java.util.*;

public class Property {
    public static void main(String[] args) {
        System.out.println(new Date());
        Properties p = System.getProperties();
        p.list(System.out);
        System.out.println("--- Memory Usage:");
        Runtime rt = Runtime.getRuntime();
        System.out.println("Total Memory = "
            + rt.totalMemory()
            + " Free Memory = "
            + rt.freeMemory());
    }
}
```

В началото на всеки програмен файл трябва да сложите **import** за да вземете всички допълнителни файлове освен този. Забележете “допълнителни.” Това е защото има библиотека класове която се взема винаги във всяка Java програма: **java.lang**. Пуснете си броузера и вижте документацията от Sun. (ако я нямате свалена от java.sun.com или по друг начин инсталлирана Java документация, сега му е времето да се сдобиете). Ако погледнете **packages.html** ще видите списък на всичките пакети които идват с Java. Изберете **java.lang**. Под “Class Index” ще видите списък на всичките класове, които са част от тази библиотека. Понеже **java.lang** неявно се включва във всеки Java сурс тези класове са автоматично достъпни. В списъка ще видите **System** и **Runtime**, които са използвани в **Property.java**. Няма **Date** клас в списъка **java.lang**, което значи че трябва да импортирате друга библиотека за да го използвате. Ако не знаете в коя библиотека е конкретният клас или искате да видите всичките класове можете да изберете “Class Hierarchy” в Java документацията. Доста време отнема това да стане в броузера, но пък можете да видите всеки клас, който идва с Java. Тогава може да използвате функцията “find” на броузера за да намерите **Date**. Като го направите ще видите че го намира като **java.util.Date**, което идва да каже че е в **util** библиотеката и трябва да напишете **import java.util.*** за да използвате **Date**.

Ако разглеждате документацията започвайки от **packages.html** файла (която аз съм сложил за начална страница по подразбиране на моя броузер), изберете **java.lang** и тогава **System**. Ще видите че **System** класа има няколко полета и ако изберете **out** ще откриете че е **static PrintStream** обект. Тъй като е **static** не се налага да създавате нищо. **out** е вече там и може да го използвате. Какво може да правите с този **out** обект е определено от неговия тип: **PrintStream**. Подходящо **PrintStream** е показан като хипервръзка, така че ако щракнете там

⁵ Some programming environments will flash programs up on the screen and close them before you've had a chance to see the results. You can put in the following bit of code at the end of **main()** to pause the output:

```
try {
    Thread.currentThread().sleep(5 * 1000);
} catch(InterruptedException e) {}
```

This will pause [the output](#) for **five** seconds. This code involves concepts that will not be introduced until much later in the book, so you won't understand it until then, but it will do the trick.

може да видите всичките методи на **PrintStream**. Има доста малко и ще се разгледат по-нататък. Засега интересното е **println()**, което ефективно е “изкарай на конзолата каквото ти давам и мини на нов ред.” Така в каква да е Java програма която пишете може да кажете **System.out.println("things")** винаги когато искате да пишете нещо на конзолата.

Името на класа е същото като на файла. Когато създавате самостоятелна програма като тази, един от класовете трябва да има същото име като файла. (Компилаторът се сърди ако не е така.) Въпросният клас трябва да има метод **main()** със следващата сигнатура:

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

public ключовата дума показва, че методът е достъпен за останалия свят (описано детайлно в глава 5). Аргументът на **main()** е масив от **String** обекти. **args** не се използва в тази програма, но е необходим, понеже там се съдържат аргументите (ключовете, квалификаторите - б.пр.) на командната линия.

Първия ред от програмата е доста интересен:

```
| System.out.println(new Date());
```

Гледаме аргумента: **Date** обект се създава само за да даде стойността си на **println()**. Щом редът се изпълни **Date** не е необходим вече и боклучарят ще дойде по някое време да го махне. Не е необходимо ние да се притесняваме за това.

Вторият ред извиква **System.getProperties()**. Ако прочете документацията със своя броузър ще видите, че **getProperties()** е **static** метод на класа **System**. Понеже е **static** няма нужда да създавате обекти за да викате метода; **static** методите винаги са достъпни без значение дали има или няма създадени обекти. Когато викате **getProperties()** се създават характеристиките на системата като обект **Properties**. Манипуляторът който се връща се запомня в манипулятора на **Properties** наречен **p**. В третия ред се вижда че **Properties** обектът има метод наречен **list()** който праща цялото това съдържание на **PrintStream** обекта който сте дали като аргумент.

Четвъртия и шестия ред в **main()** са типични оператори за извеждане. Забележете че за извеждане на няколко **String** стойности просто ги разделяме със знак ‘+’. Нещо странно става там, обаче. Знакът ‘+’ не означава събиране, когато е използван със **String** обекти. Нормално не бихте приписали никакво значение на ‘+’ когато мислите за стрингове. Java **String** класът е блажен (същевременно - проклет, б.пр.) с нещо, което се нарича “operator overloading.” Тоест ‘+’ знакът само като се използва със **String** обекти има различно поведение от където и да е другаде. За **Strings** то е “конкатенирай тези два стринга.”

Но това не е всичко. Ако погледнете оператора:

```
System.out.println("Total Memory = "
    + rt.totalMemory()
    + " Free Memory = "
    + rt.freeMemory());
```

totalMemory() и **freeMemory()** връщат числови стойности, а не **String** обекти. Какво става, когато “прибавите” числено значение към **String**? Компилаторът вижда проблема и магически вика подходящия метод, който превръща числата (**int**, **float**, etc.) в **String**, който после може да бъде “събиран” със знака плюс. Това автоматично превръщане на типовете също попада в категорията на операторния овърлоудинг.

Много от литературата за Java буйно твърди, че операторният овърлоудинг (черта на C++) е лошо нещо, и наистина то е! Тук обаче то е вързано за компилатора и е само за **String** обекти, та не може вие да го правите за никакви оператори като пишете програми.

Петия ред в `main()` създава **Runtime** обект чрез извикване на **static** методът `getRuntime()` за класа **Runtime**. Връща се манипулятор на **Runtime** обект; дали това е статичен обект или е създаден с `new` не е необходимо да ви засяга, тъй като можете да използвате обектите без да се грижите за почистването им. Както е показано **Runtime** може да даде информация за използването на паметта.

Коментари и вградена документация

Има два типа коментари в Java. Първият е традиционния С-стил коментар който беше наследен от C++. Този вид започва с `/*` и продължава, евентуално на много редове, до срещане на `*/`. Забележете, че много програмисти започват новите редове от продължаващия коментар с `*`, така че често може да видите:

```
/* Това е  
 * Коментар който продължава  
 * На повече от една линии  
 */
```

Забележете че всичко между `/*` и `*/` се игнорира така че е същото да кажем:

```
/* Това е коментар който  
продължава на повече от една линия */
```

Втората форма на коментара идва от C++. Това е коментар на един ред, който започва с `//` и продължава до края на реда. Този тип коментар е удобен и много използван, понеже е лесен. Няма нужда да ловувате по клавиатурата за да намерите `/` и после `*` (просто натискате един клавиши два пъти) и няма нужда да затваряте коментара. Така че често ще видите:

```
// това е коментар на един ред
```

Коментар на документацията

Едно от умните неща в езика Java че проектантите не считаха писането на код за единствено важна активност – те също помислиха за документирането. Възможно най-големия проблем с документацията е нейната поддръжка. Ако документацията и кода са отделно, голяма сума тоха става да се отразява всяка промяна. Решението изглежда просто: да се свърже кода към документацията. Най-лесният начин за това е всичко да се сложи в един единствен файл. За да се завърши картина обаче трябва специален синтаксис за документацията и инструмент, който да я извлича от общия файл. Това е направено в Java.

Инструментът за извлечение на коментарите е наречен *javadoc*. Той използва част от технологията на Java компилатора за да гледа за специалните белези, които вие пишете в кода си. Не само се извличат коментарите, маркирани по този начин, но също се дава и името на класа или метода, свързан с коментара. По този начин може да се разминете с минимален труд при създаването на прилична документация.

Изходът от *javadoc* е HTML файл който може да видите със своя броузър. Този инструмент ви позволява да поддържате единствен сурс файл и да извлечате от него полезна документация. Поради *javadoc* ние разполагаме със стандарт за създаване на документация и е достатъчно лесно да очакваме и даже да поръчаме документация за всичките Java библиотеки.

СИНТАКСИС

Всичките команди на явадок са в `/**` коментари. Коментарът завършва с `*/` както обикновено. Има два основни пътя да се използва javadoc: вграждане на HTML и използване на "doc tags." Doc tags са команди, които започват с '`@`' и се слагат в началото на команден ред. (Водещата '`*`' обаче се игнорира.)

Има три "типа" коментарна документация, които съответстват на елемента, който предхождат: клас, променлива или метод. Тоест, коментар на клас започва преди клас; коментар на променлива започва точно при дефиниране на променлива и коментар на метод започва точно при започването на дефиниция на метод. Като прост пример:

```
/** Коментар на клас */
public class docTest {
    /** Коментар на променлива */
    public int i;
    /** Коментар на метод */
    public void f() {}
}
```

Забележете че javadoc ще обработи коментарите само за **public** и **protected** членовете. Коментарите за **private** и "приятелски" (виж глава 5) се игнорират и няма да видите изход за тях. (Може да използвате **-private** флаг за да включите и **private** членовете.) Това има смисъл защото **public** и **protected** членовете са достъпни извън файла, което е перспективата на клиент-програмиста. Обаче всички **class** коментари се включват в изхода.

Изходът е HTML който има същия стандартен формат както останалата Java документация, така че потребителите ще се чувстват комфортно с вашия код и лесно ще го проучват. Струва си да се вмъкне гореспоменатия код, да се пусне през javadoc и да се види какъв HTML ще се получи, за да се видят резултатите.

Вграден HTML

Javadoc дава HTML команди чрез произведения HTML документ. Това позволява пълно използване на HTML; обаче основният мотив е да може да се форматира кода, както в:

```
/**
 * <pre>
 * System.out.println(new Date());
 * </pre>
 */
```

Може също да използвате HTML точно както бихте го правили в обикновени Web документи:

```
/**
 * You can <em>even</em> insert a list:
 * <ol>
 * <li> Текст едно
 * <li> Текст 2
 * <li> И три
 * </ol>
 */
```

Запомнете че в документен коментар звездите в началото на линията се изхвърлят от javadoc, както и водещите празни позиции (шпации - б.пр). Javadoc преформатира всичко така че да съответства на стандартния формат на документацията. Не използвайте заглавия като **<h1>** или **<hr>** за вграден HTML понеже javadoc слага свои и вашите ще си пречат с тях.

Всички типове от коментарна документация – клас, променлива и метод – поддържат вграден HTML.

@see: отнасяне към други класове

Всичките три типа документни коментари могат да съдържат **@see** директиви, които позволяват да се отнасяте към други класове. Javadoc ще произведе HTML със **@see** хиперсвързани към другата документация. Формите са:

```
| @see classname  
| @see fully-qualified-classname  
| @see fully-qualified-classname#method-name
```

Всяка добавя хиперсвързан “See Also” вход в генерираната документация. Javadoc няма да провери за валидност хипервръзките, които сте посочили.

Директива за документиране на клас

Заедно с вградения HTML и **@see** отнасянията документацията на клас може да включва и информация за версията и авторското име. Документацията за клас може също да се използва с **интерфейси** (описано е по-нататък в книгата).

@version

Има следната форма:

```
| @version version-information
```

където **version-information** е каквато искате да включите информация. Когато **-version** флаг се сложи на javadoc командния ред, информацията за версията ще се извика специално за HTML документацията.

@author

Има следната форма:

```
| @author author-information
```

където **author-information** е, предполага се, вашето име, но може да включва електронен адрес и друга подходяща информация. Когато **-author** флагът присъства в командната линия, информацията за автора се извиква специално в HTML документацията.

Може да имате няколко директиви за автор при списък от автори, но те трябва да са последователно разположени. Всичката авторска информация ще бъде събрана заедно в генерираната HTML документация.

Директиви за документиране на променлива

Може да включва само вграден HTML и **@see** отнасяния.

Директиви за документация на метод

Както и вградена документация и **@see** отнасяния, методите допускат директиви за параметри, връщани стойности и изключения.

@param

Формата е:

```
| @param parameter-name description
```

където **parameter-name** е идентификатор в списъка на параметрите, **description** е текст който може да заеме няколко линии. Описанието се счита завършено, когато се срещне нова директива за документацията. Може да имате всясасъв брой, предполагамо по едно за всеки параметър.

@return

Има следната форма:

```
| @return description
```

където **description** дава смисъла на връщаната стойност. Може да продължи на следващи линии.

@exception

Изключенията ще се описват в глава 9, но накъсо те са обекти, които могат да бъдат "изхвърлени" от метод, който се е провалил. Въпреки че само едно изключение може да възникне като извикате метод, конкретен обект би могъл да произведе неограничен брой типове изключения, всяко от които трябва да се опише. Така че формата на директивата е:

```
| @exception fully-qualified-class-name description
```

където **fully-qualified-class-name** дава недвусмислено име на клас, който е описан някъде, а **description** (може да заеме няколко линии)казва защо този конкретен случай на изключение може да възникне при викането на метода.

@deprecated

Това е ново в Java 1.1. То е за да се бележат черти, които са били заместени от по-нови такива. Белегът съветва да не се използва повече старата черта, понеже в бъдеще тя вероятно ще бъде махната. Методите които са маркирани с **@deprecated** карат компилатора да издаде предупреждение, ако се използват.

Пример за документация

Ето пак първата Java програма, този път с добавени коментари за документация:

```
//: c02:Property.java
import java.util.*;

/** The first Thinking in Java example program.
 * Lists system information on current machine.
 * @author Bruce Eckel
 * @author http://www.BruceEckel.com
 * @version 1.0
 */
public class Property {
    /** Sole entry point to class & application
     * @param args array of string arguments
     * @return No return value
     * @exception exceptions No exceptions thrown
    */
}
```

```
public static void main(String[] args) {
    System.out.println(new Date());
    Properties p = System.getProperties();
    p.list(System.out);
    System.out.println("--- Memory Usage:");
    Runtime rt = Runtime.getRuntime();
    System.out.println("Total Memory = "
        + rt.totalMemory()
        + " Free Memory = "
        + rt.freeMemory());
}
} ///:~
```

Първия ред:

```
//: Property.java
```

използва моя собствена техника за слагане на `:' като специален маркер за коментара, съдържащ името на файла. Последния ред също свършва с коментар, който показва края на вистинга на сорсовия код, което позволява да бъде автоматично извлечено от текста на книгата и да бъде дадено на компилатора. Това е описано детайлно в глава 17.

Стил на кодиране

Неофициален стандарт в Java е да се пише с главна буква първото име на класа. Ако името се състои от няколко думи, те се долепят (не се използва подчертаващо тире за съединяване на думите) и всяка дума е с главна буква като в:

```
class AllTheColorsOfTheRainbow { // ...
```

За почти всичко друго: методи, полета (член-променливи) и имена на манипулатори приетият стил е точно както за класовете освен че първата буква на идентификатора е малка. Например:

```
class AllTheColorsOfTheRainbow {
    int anIntegerRepresentingColors;
    void changeTheHueOfTheColor(int newHue) {
        // ...
    }
    // ...
}
```

Разбира се, ще помните, че потребителят ще трябва да пише всички тези дълги имена и да бъдете милостиви.

Резюме

В тази глава се запознахте достатъчно с Java програмирането за да можете да напишете приста програма, а също получихте общ поглед върху езика и някои от неговите основни идеи. Примерите до тук обаче бяха от вида "направи това, после направи нещо друго." Ако искате програмата да прави избор, както "ако резултатът е червено, прави това, ако не е, тогава прави нещо друго"? Поддръжката на Java за тези съществени неща ще видите в следващата глава.

Упражнения

1. Следвайки първия пример в тази глава създайте "Hello, World" която просто извежда това на екрана. Нужен е само един метод във вашия клас ("main", който се изпълнява когато програмата стартира). Не забравяйте да го направите **static** и да вмъкнете списък аргументи, макар и да не го използвате понататък. Компилирайте програмата с **javac** а и я стартирайте с **java**.
2. Напишете програма която да извежда трите аргумента на командния ред.
3. Намерете кода за втора версия на **Property.java**, който е прост пример за документационен коментар. Изпълнете **javadoc** с файла и разгледайте резултатите с броузера си.
4. Вземете програмата от упражнение 1 и добавете документация. Извлечете този коментар в HTML файл с **javadoc** и я разгледайте в броузера.

3: Управление хода на програмата

Както живо същество една програма трябва да променя своя свят и да взема решения по време на изпълнението си.

В Java обектите и данните се манипулират чрез оператори, а решения се вземат чрез оператори за управление на изпълнението. Java е наследен от C++, така че повечето от тези оператори ще са познати на С и C++ програмистите. Java е добавил също някои подобрения и опростявания.

Използваме Java оператори

Операторът взима един или повече аргументи и произвежда нова стойност. Аргументите са с друга форма от тази на единичното извикване на метод, но ефектът е същият. Ще се чувствате доста комфортно с операторите благодарение на вашия предишен опит с операторите. Събиране (+), изваждане и унарен минус (-), умножение (*), деление (/) и присвояване (=) — всичко работи както във всеки друг език.

Всички оператори произвеждат стойност от своите операнди. В добавка операторът може да промени стойността на operand. Това се нарича *страничен ефект*. Най-широко се употребяват оператори за получаване на страничен ефект, но ще помните че има и стойност достъпна за употреба точно както при операторите без страничен ефект.

Почти всички оператори работят с примитиви. Изключениета са '=' , '==' и '!=', които работят с всички обекти (и са приена за обръквания с обектите). В добавка, класа **String** поддържа '+' и '+='.

Приоритет

Приоритетът на операторите определя как трябва да се изпълняват, ако се срещнат няколко едновременно. Java има специфични правила които определят реда на изпълнението. Най-лесното за запомняне е че умножението и делението стават преди събирането и изваждането. Програмистите често забравят другите правила за приоритет, така че ще използват скоби за да направите реда на изпълнение явен. Например:

| A = X + Y - 2/2 + Z;

има друго значение от:

| A = X + (Y - 2)/(2 + Z);

Присвояване

Присвояването се изпълнява с оператора =. Той значи "вземи стойността от дясната страна (често наричано *rvalue*) и я копирай в лявата страна (често наричано *lvalue*). *rvalue* е някоя константа, променлива или израз която може да даде стойност, но *lvalue* трябва да бъде различна, именувана променлива. (Тоест, трябва да има физическо място за запомняне на

стойността.) Може да присвоявате константа на променлива (**A = 4;**) но не може да присвоявате нищо на константа – тя не може да бъде lvalue. (Не може **4 = A;.**)

Присвояването на примитивите е твърде праволинейно. Тъй като примитивът съдържа стойност, а не манипулатор на обект, когато присвоявате примитив копирате от едно място на друго. Например **A = B** за примитиви копира **B** в **A**. Ако продължите с модификация на **A, B** естествено не е засегната от това. Това е, което очаквате като програмист в повечето ситуации.

Обаче когато присвоявате обект нещата се променят. Щом манипулирате обект, манипулира се манипулаторът му, така чи когато присвоявате “от един обект на друг” фактически копирате манипулатора от едно място на друго. Това значи че ако се напише **C = D** свързвате с това, че **C** и **D** сочат към един и същ обект, този към който **D** сочише оригинално. Следващия пример ще демонстрира това.

Като странична реплика, първото нещо което виждате е **package** оператор за **package c03**, индициращ глава 3 на тази книга. Първата програма от всяка глава ще съдържа такъв оператор за да зададе номера на главата в останалите програми в нея. В глава 17 ще видите като резултат всички листинги от глава 3 (освен онези които имат различни имена на пакетите) ще се сложат автоматично в директория наречена **c03**, на глава 4 листингите ще бъдат в **c04** и така нататък. Всичко това се случва чрез **CodePackager.java** програмата показана в глава 17, а в глава 5 концепцията на пакетите ще бъде напълно обяснена. Това което трябва да знаете засега е, че за тази книга редовете с форма като на **package c03** се използват точно за посочване на директория за програмите към главата.

За да се пусне програмата, трябва да се осигури classpath да съдържа кореновата директория където сте разположили книгата. (От тази директория ще виждате поддиректории **c02, c03, c04**, и т.н..)

За по-късните версии на Java (1.1.4 and on) когато **main()** е във файл с **package** оператор ще трябва да дадете пълното име на пакета преди името на програмата за да я стартирате. В този случай командният ред е:

```
| java c03.Assignment
```

Помните това винаги, когато пускате програма която е в **package**.

Ето го и примера:

```
//: c03:Assignment.java
// Присвояването на обекти е малко сложно
package c03;

class Number {
    int i;
}

public class Assignment {
    public static void main(String[] args) {
        Number n1 = new Number();
        Number n2 = new Number();
        n1.i = 9;
        n2.i = 47;
        System.out.println("1: n1.i: " + n1.i +
            ", n2.i: " + n2.i);
        n1 = n2;
        System.out.println("2: n1.i: " + n1.i +
            ", n2.i: " + n2.i);
        n1.i = 27;
```

```

    System.out.println("3: n1.i: " + n1.i +
        ", n2.i: " + n2.i);
}
} //:~

```

Класът **Number** е простиčък и двата му екземпляра (**n1** и **n2**) се създават в **main()**. На **i** във всеки **Number** се дава различна стойност и после **n2** се присвоява на **n1**, а **n1** е променено. В много програмни езици бихте очаквали **n1** и **n2** да бъдат независими през цялото време, но понеже сте присвоили манипулатор, ще видите следния шзход:

```

1: n1.i: 9, n2.i: 47
2: n1.i: 47, n2.i: 47
3: n1.i: 27, n2.i: 27

```

Промяната на **n1** обекта се оказва че променя също и **n2** обекта! Това е защото и **n1** и **n2** съдържат един и същ манипулатор, който сочи към един обект. (Оригиналния манипулатор в **n1** който сочеше към обекта съдържащ 9 беше презаписан при присвояването и фактически загубен; неговият обект ще бъде изчистен от боклучаря.)

Този феномен е често наричан *aliasing* и е основен начин, по който Java работи с обекти. Но какво ще правим ако не искате да има псевдоними в този случай? Бихте могли преди присвояването да напишете:

```
n1.i = n2.i;
```

Така остават два отделни обекта наместо подхвърляне на един и насочване на **n1** и **n2** към същия обект, но скоро ще разберете, че манипулирането на полетата в един обект е объркана работа и в противоречие с принципите на ООП. Това не е тривиален въпрос и е описан за глава 12, която е посветена на псевдонимите. Междувременно ще помните, че присвояването на обекти може да донесе изненади.

Aliasing при извикване на методи

Псевдоними ще се получат също и ако предавате обект като параметър на метод:

```

//: c03:PassObject.java
// Даването на обекти на метод може да не е това,
// с което сте свикнали.

class Letter {
    char c;
}

public class PassObject {
    static void f(Letter y) {
        y.c = 'z';
    }
    public static void main(String[] args) {
        Letter x = new Letter();
        x.c = 'a';
        System.out.println("1: x.c: " + x.c);
        f(x);
        System.out.println("2: x.c: " + x.c);
    }
} //:~

```

В много езици методът **f()** щеше да направи копие на аргумента си **Letter y** вътре в обхвата на метода. Но пак манипулатор се изпраща, така че линията

```
y.c = 'z';
```

фактически променя обекта извън **f()**. Изходът показва това:

```
1: x.c: a  
2: x.c: z
```

Aliasing-ът и решението му са сложни въпроси; въпреки че трябва да чакате до глава 12 за всички отговори, трябва да сте предупредени от сега, за да не попадате във вълчи ями.

Математически оператори

Основните математически оператори са същите като тези в повечето езици за програмиране: събиране (+), изваждане (-), деление (/), умножение (*) и модуло (%), дава остатъка при целочислено деление (отрязва, а не закръгля резултата). Целочисленото деление отрязва, а не закръгля резултата.

Java използва също и съкратена нотация за получаване на присвояване и обработка на един път. Тя е оператор, следван от знак за равенство и работи за всички оператори на езика (навсякъде където има смисъл). Например за да добавим 4 към променливата **x** и присвоим резултата на **x** използваме: **x += 4;**

Този пример показва приложение на математическите оператори:

```
//: c03/MathOps.java  
// Демонстрира математическите оператори  
import java.util.*;  
  
public class MathOps {  
    // Create a shorthand to save typing:  
    static void prt(String s) {  
        System.out.println(s);  
    }  
    // shorthand to print a string and an int:  
    static void plnt(String s, int i) {  
        prt(s + " = " + i);  
    }  
    // shorthand to print a string and a float:  
    static void pFlt(String s, float f) {  
        prt(s + " = " + f);  
    }  
    public static void main(String[] args) {  
        // Create a random number generator,  
        // seeds with current time by default:  
        Random rand = new Random();  
        int i, j, k;  
        // '%' limits maximum value to 99:  
        j = rand.nextInt() % 100;  
        k = rand.nextInt() % 100;  
        plnt("j", j); plnt("k", k);  
        i = j + k; plnt("j + k", i);  
        i = j - k; plnt("j - k", i);  
        i = k / j; plnt("k / j", i);  
        i = k * j; plnt("k * j", i);  
        i = k % j; plnt("k % j", i);  
        j %= k; plnt("j % k", j);  
        // Тестове за числа с плаваща запетая:  
        float u,v,w; // applies to doubles, too  
        v = rand.nextFloat();
```

```

w = rand.nextFloat();
pFlt("v", v); pFlt("w", w);
u = v + w; pFlt("v + w", u);
u = v - w; pFlt("v - w", u);
u = v * w; pFlt("v * w", u);
u = v / w; pFlt("v / w", u);
// the following also works for
// char, byte, short, int, long,
// and double:
u += v; pFlt("u += v", u);
u -= v; pFlt("u -= v", u);
u *= v; pFlt("u *= v", u);
u /= v; pFlt("u /= v", u);
}
} //:~

```

Първото нещо, което ще видите са някои бързи начини за писане: `prt()` методът принтва `String`, `pInt()` печата `String` следван от `int` и `pFlt()` печата `String` следван от `float`. Разбира се, те всички използват `System.out.println()`.

За да генерира числа, програмата първо създава `Random` обект. Понеже не се дават аргументи, Java използва текущото време като начало за генератора на случайни числа. Програмата генерира множество типове случайни числа чрез `Random` обекта просто извиквайки различни методи: `nextInt()`, `nextLong()`, `nextFloat()` or `nextDouble()`.

Операторът модуло, използван със случайни числа, ограничава стойността до горната и граница минус едно (99 в този случай) (и я прави цяла - б.пр.).

Унарни минус и плюс оператори

Унарният минус (-) и унарният плюс (+) са същите като бинарните минус и плюс. Компилаторът разбира кой вид се иска от начина на записване. Например операторът

```
| x = -a;
```

има очевидно значение. Компилаторът е в състояние да разбере:

```
| x = a * -b;
```

но читателят може да бъде смутен, та е по-добре да се напише:

```
| x = a * (-b);
```

Унарният минус дава обратното на дадено число. Унарният плюс е за симетрия с минуса, макар и да не прави много.

Авто инкремент и декремент

Java, подобно на C, е пълен с кратки начини. Те могат да направят кодирането много по-лесно, а също да направят четенето по-трудно, а понякога и по-лесно.

Два от най-тънките кратки начини са инкремент и декремент (често споменавани като авто-инкремент и авто-декремент оператори). Декремент операторът е `--` и значи "намали с единица." Инкрементния оператор е `++` и значи "увеличи с единица." Ако `A` е `int`, например, изразът `++A` е еквивалентен на `(A = A + 1)`. Двата оператора дават като резултат стойност на променлива.

Всеки от двата има две версии, често наричани префиксна и постфиксна версия. Преинкремент значи че `++` се появява преди променлива или израз, а пост-инкремент значи че `++`

се появява след променлива или израз. Подобно, предекрементът значи че -- се появява преди променлива или израз и постдекремент — че -- оператор има след променлива или израз. За преинкремент и предекремент, (т.е. **++A** и **--A**) операцията се изпълнява и се произвежда стойност. За пост- динкремента и декремента (т.е. **A++** и **A--**) първо се произвежда стойност и после се изпълнява операцията. Като пример:

```
//: c03:Autolnc.java
// Демонстрира ++ и -- оператори

public class Autolnc {
    public static void main(String[] args) {
        int i = 1;
        prt("i : " + i);
        prt("++i : " + ++i); // Pre-increment
        prt("i++ : " + i++); // Post-increment
        prt("i : " + i);
        prt("--i : " + --i); // Pre-decrement
        prt("i-- : " + i--); // Post-decrement
        prt("i : " + i);
    }
    static void prt(String s) {
        System.out.println(s);
    }
} //:~
```

Тази програма извежда следното:

```
i : 1
++i : 2
i++ : 2
i : 3
--i : 2
i-- : 2
i : 1
```

Може да се види, че при префиксната форма се изпълнява операцията преди да се получи стойност и обратно за постфиксната форма. Тези са единствените оператори (освен онези които включват присвояване) които имат странични ефекти. (Променят operand, а не да му използват стойността.)

Инкрементният оператор е едно обяснение на името C++, а именно “една стъпка оттък С.” В отдавнашен Java разговор Bill Joy (един от създателите), каза че “Java=C++-” (С плюс плюс минус минус), имайки пред вид че Java е C++ с мањнати ненужно трудни части и по този начин много по-прост език. Като напредвате в тази книга ще виждате че много неща са по-прости, но все пак Java не е толкова по-лесен от C++.

Оператори за отношение

Операторите за отношение произвеждат **boolean** резултат. Те изчисляват отношението между стойностите на operandите. Израз за отношение произвежда **true** ако отношението го има и **false** ако го няма. Релационните оператори са по-малко (<), по-голямо (>), по-малко или равно (<=), по-голямо или равно (>=), еквивалентност (==) и нееквивалентни (!=). Еквивалентността и нееквивалентността работят с всички вградени типове, но други сравнения нама да работят с типа **boolean**.

Проверка на еквивалентност на обекти

Операторите за сравнение `==` и `!=` също работят с всички обекти, но работата им често смущава начинаещия Java програмист. Ето пример:

```
//: c03:Equivalence.java

public class Equivalence {
    public static void main(String[] args) {
        Integer n1 = new Integer(47);
        Integer n2 = new Integer(47);
        System.out.println(n1 == n2);
        System.out.println(n1 != n2);
    }
} ///:~
```

Изразът `System.out.println(n1 == n2)` ще изведе резултата от `boolean` сравнението в себе си. Сигурно изходът ще бъде `true` и после `false`, понеже `Integer` обектите са един и същ обект. Но докато съдържанието на обектите е едно и също, манипулаторите са различни и `==` и `!=` сравняват манипулатори. Така че резултатът е `false` и после `true`. Естествено, това изненадва хората отначало.

Ами ако искате да сравните съдържанието на обектите? Трябва да използвате специален метод `equals()` който съществува за всички обекти (непримитивни, те работят чудесно с `==` и `!=`). Ето как се използва:

```
//: c03:EqualsMethod.java

public class EqualsMethod {
    public static void main(String[] args) {
        Integer n1 = new Integer(47);
        Integer n2 = new Integer(47);
        System.out.println(n1.equals(n2));
    }
} ///:~
```

Резултатът ще бъде `true`, както може да се очаква. Е, това не е толкова просто както другото. Ако създавате собствен клас, както тук:

```
//: c03:EqualsMethod2.java

class Value {
    int i;
}

public class EqualsMethod2 {
    public static void main(String[] args) {
        Value v1 = new Value();
        Value v2 = new Value();
        v1.i = v2.i = 100;
        System.out.println(v1.equals(v2));
    }
} ///:~
```

отново сте на изходно положение: резултатът е `false`. Това е защото дефолтвото поведение на `equals()` е да сравнява манипулатори. Така че докато не подтиснете `equals()` във вашия нов клас няма да получите исканото поведение. За нещастие няма да учите за подтискането до

глава 7, но да сте предупредени за поведението на `equals()` може да ви спести малко скръб междувременно.

Повечето от библиотечните класове на Java прилагат такъв `equals()` че се сравнява съдържанието на обектите, а не манипуляторите им.

Логически оператори

Логическите оператори AND (`&&`), OR (`||`) и NOT (`!`) произвеждат **boolean** стойност **true** или **false** на основата на логическото отношение на аргументите. Този пример използва оператори за отношение и логически оператори:

```
//: c03:Bool.java
// Relational and logical operators
import java.util.*;

public class Bool {
    public static void main(String[] args) {
        Random rand = new Random();
        int i = rand.nextInt() % 100;
        int j = rand.nextInt() % 100;
        prt("i = " + i);
        prt("j = " + j);
        prt("i > j is " + (i > j));
        prt("i < j is " + (i < j));
        prt("i >= j is " + (i >= j));
        prt("i <= j is " + (i <= j));
        prt("i == j is " + (i == j));
        prt("i != j is " + (i != j));

        // Treating an int as a boolean is
        // not legal Java
        //! prt("i && j is " + (i && j));
        //! prt("i || j is " + (i || j));
        //! prt("i is " + !i);

        prt("(i < 10) && (j < 10) is "
            + ((i < 10) && (j < 10)) );
        prt("(i < 10) || (j < 10) is "
            + ((i < 10) || (j < 10)) );
    }
    static void prt(String s) {
        System.out.println(s);
    }
} //:~
```

Може да прилагате AND, OR и NOT за **boolean** стойности само. Не може да използвате не-**boolean** сякаш са **boolean** в логически израз както в C и C++. Може да се убедите сами като махнете коментарите започващи с `//!`. Следващите отношения, обаче, произвеждат **boolean** стойности чрез използване на сравнения, а после логически оператори с резултатите.

Изходът изглежда като това:

```
i = 85
j = 4
i > j is true
i < j is false
```

```
i >= j is true
i <= j is false
i == j is false
i != j is true
(i < 10) && (j < 10) is false
(i < 10) || (j < 10) is true
```

Забележете че **boolean** автоматично се превръща в подходящ текст там, където трябва да се очаква **String**.

Може да заместите определението за **int** в горната програма с който и да е примитивен тип освен **boolean**. Имайте предвид, обаче, че сравняването на числа с плаваща запетая е много точно. Число което и с най-малка част се отличава от друго е все още "неравно." И най-мъничко да е числото над нула то все още е ненулево.

Феноменът short-circuiting

Когато се работи с логически оператори се сблъскваме с феномен наречен "short circuiting." То означава, че логическият израз се изчислява само докато може еднозначно да се определи неговата истинност или неверност. Поради това може да не се изчисляват всички части на израза. Следващия пример демонстрира short-circuiting:

```
//: c03:ShortCircuit.java
// Demonstrates short-circuiting behavior
// with logical operators.

public class ShortCircuit {
    static boolean test1(int val) {
        System.out.println("test1(" + val + ")");
        System.out.println("result: " + (val < 1));
        return val < 1;
    }
    static boolean test2(int val) {
        System.out.println("test2(" + val + ")");
        System.out.println("result: " + (val < 2));
        return val < 2;
    }
    static boolean test3(int val) {
        System.out.println("test3(" + val + ")");
        System.out.println("result: " + (val < 3));
        return val < 3;
    }
    public static void main(String[] args) {
        if(test1(0) && test2(2) && test3(2))
            System.out.println("expression is true");
        else
            System.out.println("expression is false");
    }
} ///:~
```

Всеки тест изпълнява логически операции над аргументите и връща лъжа или истина. Също се извежда информация какво е било викано. Тестовете се използват в израза:

```
if(test1(0) && test2(2) && test3(2))
```

Бихте могли естествено да очаквате, че и трите теста ще се изпълнят, но изходът сочи друго:

```
test1(0)
result: true
```

```
test2(2)
result: false
expression is false
```

Първият тест дава **true** резултат, така че изчислението продължава. Вторият тест обаче дава **false** резултат. Тъй като това означава че целият израз ще има стойност **false**, защо да се изпълнява останалата част? Това би могло да бъде скъпо. Причината за short-circuiting-a, фактически, е точно това; Може да се получи подобрение на бързодействието, ако не се изчисляват всичките изрази.

Побитови оператори

Побитовите оператори дават възможност да се манипулират отделните битове в примитивен тип. Изпълнява се операция на булевата алгебра над съответните битове.

Побитовите оператори идват от ориентацията на C към работата на ниско ниво; често трябаше да се пипа директно по хардуера и да се работи с битове свързани с него. Java беше оригинално създадена за интеграция в TV устройства, така че тази ориентация към ниското ниво още имаше смисъл. Едва ли ще използвате побитовите оператори много, обаче.

Побитовия AND оператор (**&**) дава единица като изходен бит ако и двата входни са единица; иначе дава нула. Побитовия OR оператор (**|**) дава единица ако поне единият е единица и дава нула само ако и двата входни бита са нула. Побитовото ИЗКЛЮЧВАЩО ИЛИ, или XOR (**^**) дава единица само ако някой от входните битове е единица, но не и двата. Побитовото NOT (**~**, също наричано комплементиращ оператор) е унарен оператор; той приема само един аргумент. (Всичките други побитови оператори са двуместни.) Побитовото NOT дава обратния на входния бит – ако входният бит е бил нула, изходният е единица.

За побитовите оператори се използват същите знаци, както и за логическите, затова е полезно да имате евристика, която да напомня за значенията: тъй като битовете са “малки,” само един знак има при побитовите оператори.

Побитовите оператори могат да се комбинират със знака **=** за обединяване на операцията и присвояването: **&=**, **|=** and **^=** всичките са законни. (Тъй като **~** е унарен оператор той не може да се комбинира със знака **=**.)

Типът **boolean** се третира като еднобитова стойност така че е малко различен. Може да изпълни побитово AND, OR и XOR, но не и побитово NOT (вероятно за да се предотврати смесването с логическото NOT). За **boolean**-ите побитовите оператори имат същия ефект както логическите, само дето не правят short circuit. Побитовите оператори за **boolean** дават също XOR логически оператор, който не е включен в списъка на “логическите” оператори. Не се допуска използването на **booleans** в shift изрази, които са описани по-долу.

Shift оператори

Те също манипулират битове. Може да се използват само върху примитивни, цялостни типове. Операторът за изместване на ляво (**<<**) дава от операнда вляво на знака резултат, чийто битове са изместили толкова, колкото е числото вдясно от операнда (слагайки нули в най-малките значещи битове). Right-shift операторът със знак (**>>**) дава операнда вляво от знака си изместили битове колкото е числото след знака. Той (**>>**) използва разширение на знака: ако стойността е положителна, нули се слагат на мястото на най-значещите битове; ако стойността е отрицателна — единици. В Java също е добавен беззнаков right shift **>>>**, който използва разширение с нула: без значение какъв е знака нули се слагат на мястото на най-значещите битове. Този оператор не съществува в C и C++. Ако побитово измествите **char**, **byte**, или **short**, то ще бъде произведено в **int** преди да се направи отместването и резултатът ще бъде **int**. Само петте най-малко значещи бита на дясната страна ще се използват. Това предпазва от отмествания по-големи от броя на битовете в **int**. Ако оперирате с **long**, **long** ще

бъде резултатът. Само шестте най-малозначни бита на дясната страна ще се използват за да се предотвратят отмествания по-големи от броя на битовете в **long**. Обаче има проблем с беззнаковия right shift. Ако го използвате с **byte** или **short** може да не получите коректни резултати. (Това е прекъснато в Java 1.0 и Java 1.1.) Тези се произвеждат в **int** и се прави побитово отместване надясно, но разширение с нула не се прави, така че получавате **-1** в тези случаи. Следващия пример може да се използва за проверка на вашето приложение:

```
//: c03:URShift.java
// Тест на беззнаковия right shift

public class URShift {
    public static void main(String[] args) {
        int i = -1;
        i >>>= 10;
        System.out.println(i);
        long l = -1;
        l >>>= 10;
        System.out.println(l);
        short s = -1;
        s >>>= 10;
        System.out.println(s);
        byte b = -1;
        b >>>= 10;
        System.out.println(b);
    }
} //:~
```

Шифтовете могат да се комбинират със знака за равенство (**<<=** или **>>=** или **>>>=**). Lvalue-то се замества с lvalue-то отмествено колкото казва rvalue-то.

Ето пример който показва всичките оператори за битове:

```
//: c03:BitManipulation.java
// Using the bitwise operators
import java.util.*;

public class BitManipulation {
    public static void main(String[] args) {
        Random rand = new Random();
        int i = rand.nextInt();
        int j = rand.nextInt();
        pBinInt("-1", -1);
        pBinInt("+1", +1);
        int maxpos = 2147483647;
        pBinInt("maxpos", maxpos);
        int maxneg = -2147483648;
        pBinInt("maxneg", maxneg);
        pBinInt("i", i);
        pBinInt("~i", ~i);
        pBinInt("-i", -i);
        pBinInt("j", j);
        pBinInt("i & j", i & j);
        pBinInt("i | j", i | j);
        pBinInt("i ^ j", i ^ j);
        pBinInt("i << 5", i << 5);
        pBinInt("i >> 5", i >> 5);
        pBinInt("(~i) >> 5", (~i) >> 5);
        pBinInt("i >>> 5", i >>> 5);
    }
}
```

```

pBinInt("(~i) >>> 5", (~i) >>> 5);

long l = rand.nextLong();
long m = rand.nextLong();
pBinLong("-1L", -1L);
pBinLong("+1L", +1L);
long ll = 9223372036854775807L;
pBinLong("maxpos", ll);
long lln = -9223372036854775808L;
pBinLong("maxneg", lln);
pBinLong("l", l);
pBinLong("~l", ~l);
pBinLong("-l", -l);
pBinLong("m", m);
pBinLong("l & m", l & m);
pBinLong("l | m", l | m);
pBinLong("l ^ m", l ^ m);
pBinLong("l << 5", l << 5);
pBinLong("l >> 5", l >> 5);
pBinLong("(~l) >> 5", (~l) >> 5);
pBinLong("l >>> 5", l >>> 5);
pBinLong("(~l) >>> 5", (~l) >>> 5);
}

static void pBinInt(String s, int i) {
    System.out.println(
        s + ", int: " + i + ", binary: ");
    System.out.print("  ");
    for(int j = 31; j >=0; j--)
        if(((1 << j) & i) != 0)
            System.out.print("1");
        else
            System.out.print("0");
    System.out.println();
}

static void pBinLong(String s, long l) {
    System.out.println(
        s + ", long: " + l + ", binary: ");
    System.out.print("  ");
    for(int i = 63; i >=0; i--)
        if(((1L << i) & l) != 0)
            System.out.print("1");
        else
            System.out.print("0");
    System.out.println();
}

} ///:~

```

Двата метода накрая, **pBinInt()** и **pBinLong()** вземат **int** или **long**, респективно, и го извеждат в двоичен формат заедно с описателен стринг. Може да игнорирате имплементацията им засега.

Ще забележите използването на **System.out.print()** вместо **System.out.println()**. **print()** методът не започва нов ред на края, така че позволява да изведете реда на парчета.

Освен че демонстрира ефекта от всичките побитови оператори за **int** и **long**, този пример също показва мимикалната, максималната, +1 и -1 стойност за **int** и **long** така че може да

видите как изглеждат. Забележете че най-десният бит представя знака: 0 значи положително и 1 значи отрицателно. Изходат за **int** частта изглежда подобно на това:

```
-1, int: -1, binary:  
1111111111111111111111111111  
+1, int: 1, binary:  
00000000000000000000000000000001  
maxpos, int: 2147483647, binary:  
0111111111111111111111111111  
maxneg, int: -2147483648, binary:  
10000000000000000000000000000000  
i, int: 59081716, binary:  
0000001110000101100000111110100  
~i, int: -59081717, binary:  
111110001111010011110000001011  
-i, int: -59081716, binary:  
111110001111010011110000001100  
j, int: 198850956, binary:  
0000101110110100011100110001100  
i & j, int: 58720644, binary:  
0000001110000000000000110000100  
i | j, int: 199212028, binary:  
0000101110111101110111111100  
i ^ j, int: 140491384, binary:  
0000100001011111011101001111000  
i << 5, int: 1890614912, binary:  
0111000010110000011111010000000  
i >> 5, int: 1846303, binary:  
00000000000111000010110000011111  
(~i) >> 5, int: -1846304, binary:  
11111111100011110100111100000  
i >>> 5, int: 1846303, binary:  
00000000000111000010110000011111  
(~i) >>> 5, int: 132371424, binary:  
00000111110001110100111100000
```

Двоичното представяне на числата е **двоично комплементарно със знак**.

Триместен if-else оператор

Този оператор е необичаен понеже има три операнда. Наистина е оператор, защото произвежда стойност, за разлика от обикновения if-else който ще видите в края на главата.

Изразът има формата

boolean-exp ? value0 : value1

Ако *boolean-exp* даде **true**, *value0* се изчислява и получената стойност е тази, която се връща от оператора. Ако *boolean-exp* е **false**, *value1* се изчислява и стойността се връща от оператора.

Разбира се, бихте могли да използвате обикновен **if-else** оператор (описан по-късно), но триместният оператор е много по-сбит. Въпреки че С се гордее със сбитостта си и триместният оператор може би е въведен частично заради ефективността, ще внимавате като го използвате всекидневно – лесно е да се напише нечитаем код.

Операторът за условие може да бъде използван заради страничния му ефект или заради стойността, която произвежда, но обикновено искаме стойността му, понеже тя е това, което го прави различен от **if-else**-то. Ето пример:

```
static int ternary(int i {
```

```
    return i < 10 ? i * 100 : i * 10;
}
```

Може да видите че този код е по-компактен от това, което ще се напише без триместния оператор:

```
static int alternative(int i) {
    if (i < 10)
        return i * 100;
    return i * 10;
}
```

Втората форма е по-лесна за разбиране и не изисква много повече писане. Така че бъдете сигурни в причините когато избирате триместния оператор.

Операторът запетая

Запетаята се използва в C и C++ не само като разделител в списъците от аргументи на функции, но също като оператор за последователно изчисляване. Единственото място където запетаята-оператор се използва в Java е във **for** циклите, които ще се опишат по-късно в тази глава.

String операторът +

Един е операторът със специално използване в Java: **+** операторът може да се използва за конкатениране на стрингове, както вече знаете. Това изглежда естествено приложение на знака **+** въпреки че не се вмества в традиционния начин по който се използва **+**. Тази възможност изглеждаше добра идея в C++, така че **натоварването на оператори с много възможности** беше добавено в C++ за да позволи на C++ да добавя значения на почти всеки оператор. За нещастие, **operator overloading**-ът комбиниран с някои други ограничения в C++ стана твърде сложен за програмистите да го вграждат в своите класове. Въпреки че **operator overloading**-ът би бил много по-лесен за прилагане в Java отколкото беше в C++, тази черта беше счетена за твърде сложна, така че Java програмистите не могат да пренатоварват оператори както C++ програмистите могат.

Използването на **String**-овия **+** има нещо интересно в поведението си. Ако израз започва със **String** всички следващи оператори трябва да бъдат **String**-ове:

```
int x = 0, y = 1, z = 2;
String sString = "x, y, z ";
System.out.println(sString + x + y + z);
```

Тук Java компилаторът ще обърне **x**, **y**, и **z** в техните **String** представяния, наместо да ги събере първо. Ако обаче напишете:

```
System.out.println(x + sString);
```

По-раншните версии на Java ще сигнализират за грешка. (Новите версии, обаче, ще обърнат **x** в **String**.) Така че ако слагате заедно **String** (използвайки по-раншна версия на Java) със събиране, осигурете първия елемент да е **String** (или наниз от знаци, заграден в кавички, който компилаторът разпознава като **String**).

Обичайни капани при използването на операторите

Един от тях е опитът да се разминете без скоби когато има и най-малкото съмнение относно реда за използване на операторите. Това е в сила и в Java.

Изключително широко допускана грешка в С и С++ изглежда като това:

```
while(x = y) {  
    // ....  
}
```

Програмистът иска да провери за равенство (`==`) а не да прави присвояване. В С и С++ резултатът от присвояването винаги ще е **true** ако **y** не е нула и най-вероятно ще се получи безкраен цикъл. В Java резултатът оттози израз не е **boolean**, компилаторът очаква **boolean** и няма да превърне **int**, така че удобно ще даде грешка при компилирането, преди да се опитате да изпълнявате програмата. Така че капанът го няма в Java. (Единствения път когато няма да получите грешка по време на компилация е когато **x** и **y** са **boolean**, в който случай **x = y** е правилен израз и в горния случай — вероятно грешка.)

Подобен проблем в С и С++ има при използването на побитови AND и OR вместо логически. Побитовите AND и OR използват един от знаците (`&` или `|`) докато логическите AND и OR използват два (`&&` и `||`). Точно както с `=` и `==`, лесно е да се напише един знак вместо два. В Java компилаторът отново предотвратява това, понеже не позволява безцеремонно да използвате един знак където са нужни два.

Casting оператори

Думата *cast* е използвана в смисъл на “слагане в калъп.” Java автоматично ще промени типа на една променлива в друг щом е необходимо. Например ако присвоявате цяла стойност на променлива с плаваща запетая компилаторът автоматично ще превърне **int** във **float**. Casting-ът позволява подобни превръщания да се направят явно, а също и да се застави компилаторът да ги направи в някои случаи.

За да направите каст, сложете жерания даннов тип (заедно с всички модификатори) в скоби отляво на каквато и да е стойност. Ето пример:

```
void casts() {  
    int i = 200;  
    long l = (long)i;  
    long l2 = (long)200;  
}
```

Както се вижда може да се направи кастинг както на чисрова стойност, така и на променлива. И в двата показани случая, обаче, кастингът е излишен, защото компилаторът и автоматично би превърнал **int** стойността в **long** когато е необходимо. Все пак може да сложите каст за да подчертаете или пък за да стане кодът ви по-разбираем. В други случаи кастингът е нужен просто за да се компилира кода.

В С и С++ кастингът може да създаде главоболия. В Java кастингът е безопасен с изключение на случая когато се прави т. нар. стесняващо преобразуване (когато първоначалният тип може да съдържа повече информация от този, към който се преобразува) когато може да се загуби информация. В този случай компилаторът иска вие да направите кастинга, сякаш казвайки “това може да бъде опасно – ако искате да го направя, заявете го явно.” С разширяващото превръщане не е необходим явен каст, защото типът в който се превръща може да събере повече информация от първоначалния и информация не може да се загуби.

Java позволява кастинг на всякакви примитивни типове към всякакви примитивни типове, освен **boolean**, с който не са позволени кастове въобще. Класовите типове не позволяват кастинг. За да се превърне един в друг трябват специални методи. (**String** е специален случай и ще видите по-късно в книгата, че може да се прави кастинг на обекти в рамките на семейство типове; **Oak** може да се превърне в **Tree** и обратно, но не във външен клас като **Rock**.) (Дъб, Дърво и Скала са имената на класовете - б.пр.)

Литерали

Обикновено като сложите литерал в програмата си компилаторът знае точно какъв е типът му. Понякога, обаче, типът е двусмислен. В този случай трябва да кажете на компилатора чрез допълнителни знаци към литерала кой тип искате да е. Следващия код показва тези знаци:

```
//: c03:Literals.java

class Literals {
    char c = 0xffff; // max char hex value
    byte b = 0x7f; // max byte hex value
    short s = 0x7fff; // max short hex value
    int i1 = 0x2f; // Hexadecimal (lowercase)
    int i2 = 0X2F; // Hexadecimal (uppercase)
    int i3 = 0177; // Octal (leading zero)
    // Hex and Oct also work with long.
    long n1 = 200L; // long suffix
    long n2 = 200l; // long suffix
    long n3 = 200;
    //! long l6(200); // not allowed
    float f1 = 1;
    float f2 = 1F; // float suffix
    float f3 = 1f; // float suffix
    float f4 = 1e-45f; // 10 to the power
    float f5 = 1e+9f; // float suffix
    double d1 = 1d; // double suffix
    double d2 = 1D; // double suffix
    double d3 = 47e47d; // 10 to the power
} ///:~
```

Шестнадесетичен (основа 16), който работи с всички цели типове, се означава с водещи **0x** или **0X** следвани от 0–9 и a–f големи или малки. Ако се опитате да инициализирате променлива със стойност по-голяма отколкото тя може да съдържа (без значение в какво представяне е), компилаторът ще издаде съобщение за грешка. Забележете в горния пример максималните шестнадесетично записани стойности за **char**, **byte**, и **short**. Ако ги надминете, компилаторът автоматично ще направи **int** и ще ви каже, че трябва стесняващо превръщане за присвояването. Ще научите, че сте престъпили линията.

Осмичен (основа 8) се означава с водеща нула и цифрите 0–7. Няма двоични литерали в C, C++ и Java.

Знак зад литерала показва типа. Голямо или малко **L** значи **long**, голямо или малко **F** значи **float** и малко или голямо **D** значи **double**.

Експонентите се означават по начин, който винаги съм намирал за малко слизващ: **1.39 e-47f**. В науката и инженерството, ‘e’ означава основата на натуралните логаритми, приблизително 2.718. (По-точна **double** стойност е достъпна в Java като **Math.E**.) Използва се в експонентни изрази като $1.39 \times e^{-47}$, което значи 1.39×2.718^{-47} . Когато FORTRAN обаче беше измислен реши се, че **e** естествено ще значи “десет на степен,” което е лош избор понеже FORTRAN беше проектиран за научни и инженерни цели и човек би трявало да очаква, че проектантите ще са чувствителни към такъв род двусмислия.¹ Както и да е, този обичай беше следван и в C,

¹ John Kirkham writes:² “I started computing in 1962 using FORTRAN II on an IBM 1620. At that time³ and throughout the 1960s and into the 1970s, FORTRAN was an all uppercase language. This probably started because many of the early input devices were old teletype units [which had](#) used 5 bit Baudot code⁴, which had no lowercase capability. The ‘E’ in the exponential notation was also always upper case and was never confused with the natural logarithm base ‘e’⁵, which is always lower case. The ‘E’ simply stood for exponential⁶, which was for the base of the number system used – usually 10. At the time octal was also widely used by programmers. Although I never saw it used, if I had seen an octal number in

C++ и сега в Java. Така че ако сте свикнали да мислите в термините на **e** като основа на натуралните логаритми, ще трябва да направите душевна трансляция когато срещнете **1.39e-47f** в Java; то значи 1.39×10^{-47} .

Забележете че знакът на края не е необходим, когато компилаторът може да разбере тчния тип. С

```
| long n3 = 200;
```

няма двусмислие, така че **L** след 200 би било излишно. Обаче с

```
| float f4 = 1e-47f; // 10 to the power
```

компилаторът нормално взема експоненциалните числа като двойна точност, така че без опашното **f** ще ви даде грешка,казвайки че трябва кастиг на **double** към **float**.

Разширяване

Ще откриете, че ако изпълнявате аритметични или побитови операции над някакви типове по-малки от **int** (тоест: **char**, **byte** или **short**), стойностите ще бъдат разширени до **int** преди изпълнение на операциите и резултантната стойност ще бъде от тип **int**. Така че ако искате да присвоите обратно на изходния тип трябва да направите кастиг. (И, ако присвоявате на по-малък тип, би могла да се загуби информация.) Изобщо, най-големият тип който участва в израз определя типа на резултата от този израз; ако умножите **float** и **double**, резултатът ще е **double**; ако съберете **int** с **long** резултатът ще бъде **long**.

Java НЯМА “**sizeof**”

В С и C++ операторът **sizeof()** удовлетворява специфична нужда: дава колко байта се алокират за даден даннов тип. Най-непреодолимо **sizeof()** трява в С и C++ заради преносимостта. Различните типоме данни могат да бъдат с различни дължини на различните машини, така че програмистът трябва да узнае в някои случаи колко са дълги. Например един компютър може да запомня целите в 32 бита, докато друг в 16. Програмите могат да запомнят по-големи стойности на цялото в първата машина. Както може да си представите, преносимостта е огромно главоболие за С и C++ програмистите.

Java не се нуждае от **sizeof()** оператора за тази цел, понеже всичките типове данни са еднакви на всички машини. Не е необходимо да се мисли за преносимостта на това ниво — тя е впроектирана в езика.

Приоритетът по нов начин

Във връзка с оплакването ми от трудното запомняне на приоритета на един от моите семинари един студент предложи мнемоника, която същевременно е коментар: "Ulcer Addicts Really Like C A lot."

Mnemonic	Operator type	Operators
Ulcer	Unary	+ - ++ - ((rest...))
Addicts	Arithmetic (and shift)	* / % + - << >>
Really	Relational	> < >= <= == !=
Like	Logical (and)	&& & ^

exponential notation I would have considered it to be base 8. The first time I remember seeing an exponential using a lower case 'e' was in the late 1970's and I also found it confusing. The problem arose as lowercase crept into FORTRAN, not at its beginning. We actually had functions to use if you really wanted to use the natural logarithm base, but they were all uppercase."

	bitwise)	
C	Conditional (ternary)	A > B ? X : Y
A Lot	Assignment	= (and compound assignment like *=)

Разбира се, с шифт и битовите оператори разхвърляни по таблицата тя не е перфектна, но за не-битовите оператори работи.

Резюме на операторите

Следващият пример показва кои примитивни типове данни могат да се използват с даден оператор. Основно това е един и същ пример повтарян пак и пак, но с различни типове данни. Файльтът ще се компилира без грешка, понеже редовете които биха предизвикали грешка са изкоментирани с `//!`.

```
//: c03:AllOps.java
// Tests all the operators on all the
// primitive data types to show which
// ones are accepted by the Java compiler.

class AllOps {
    // To accept the results of a boolean test:
    void f(boolean b) {}
    void boolTest(boolean x, boolean y) {
        // Arithmetic operators:
        //! x = x * y;
        //! x = x / y;
        //! x = x % y;
        //! x = x + y;
        //! x = x - y;
        //! x++;
        //! x--;
        //! x = +y;
        //! x = -y;
        // Relational and logical:
        //! f(x > y);
        //! f(x >= y);
        //! f(x < y);
        //! f(x <= y);
        f(x == y);
        f(x != y);
        f(!y);
        x = x && y;
        x = x || y;
        // Bitwise operators:
        //! x = ~y;
        x = x & y;
        x = x | y;
        x = x ^ y;
        //! x = x << 1;
        //! x = x >> 1;
        //! x = x >>> 1;
        // Compound assignment:
        //! x += y;
        //! x -= y;
        //! x *= y;
```

```

//! x /= y;
//! x %= y;
//! x <= 1;
//! x >= 1;
//! x >>= 1;
x &= y;
x ^= y;
x |= y;
// Casting:
//! char c = (char)x;
//! byte B = (byte)x;
//! short s = (short)x;
//! int i = (int)x;
//! long l = (long)x;
//! float f = (float)x;
//! double d = (double)x;
}

void charTest(char x, char y) {
    // Arithmetic operators:
    x = (char)(x * y);
    x = (char)(x / y);
    x = (char)(x % y);
    x = (char)(x + y);
    x = (char)(x - y);
    x++;
    x--;
    x = (char)+y;
    x = (char)-y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Bitwise operators:
    x= (char)~y;
    x = (char)(x & y);
    x = (char)(x | y);
    x = (char)(x ^ y);
    x = (char)(x << 1);
    x = (char)(x >> 1);
    x = (char)(x >>> 1);
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;

```

```

x |= y;
// Casting:
//! boolean b = (boolean)x;
byte B = (byte)x;
short s = (short)x;
int i = (int)x;
long l = (long)x;
float f = (float)x;
double d = (double)x;
}

void byteTest(byte x, byte y) {
    // Arithmetic operators:
    x = (byte)(x * y);
    x = (byte)(x / y);
    x = (byte)(x % y);
    x = (byte)(x + y);
    x = (byte)(x - y);
    x++;
    x--;
    x = (byte)+ y;
    x = (byte)- y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Bitwise operators:
    x = (byte)~y;
    x = (byte)(x & y);
    x = (byte)(x | y);
    x = (byte)(x ^ y);
    x = (byte)(x << 1);
    x = (byte)(x >> 1);
    x = (byte)(x >>> 1);
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <= 1;
    x >= 1;
    x >>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! boolean b = (boolean)x;
    char c = (char)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
}

```

```

float f = (float)x;
double d = (double)x;
}
void shortTest(short x, short y) {
    // Arithmetic operators:
    x = (short)(x * y);
    x = (short)(x / y);
    x = (short)(x % y);
    x = (short)(x + y);
    x = (short)(x - y);
    x++;
    x--;
    x = (short)+y;
    x = (short)-y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Bitwise operators:
    x = (short)~y;
    x = (short)(x & y);
    x = (short)(x | y);
    x = (short)(x ^ y);
    x = (short)(x << 1);
    x = (short)(x >> 1);
    x = (short)(x >>> 1);
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <= 1;
    x >= 1;
    x >>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! boolean b = (boolean)x;
    char c = (char)x;
    byte B = (byte)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
}
void intTest(int x, int y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;

```

```

x = x % y;
x = x + y;
x = x - y;
x++;
x--;
x = +y;
x = -y;
// Relational and logical:
f(x > y);
f(x >= y);
f(x < y);
f(x <= y);
f(x == y);
f(x != y);
//! f(!x);
//! f(x && y);
//! f(x || y);
// Bitwise operators:
x = ~y;
x = x & y;
x = x | y;
x = x ^ y;
x = x << 1;
x = x >> 1;
x = x >>> 1;
// Compound assignment:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <=> 1;
x >=> 1;
x >>>= 1;
x &= y;
x ^= y;
x |= y;
// Casting:
//! boolean b = (boolean)x;
char c = (char)x;
byte B = (byte)x;
short s = (short)x;
long l = (long)x;
float f = (float)x;
double d = (double)x;
}
void longTest(long x, long y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
}

```

```

// Relational and logical:
f(x > y);
f(x >= y);
f(x < y);
f(x <= y);
f(x == y);
f(x != y);
//! f(!x);
//! f(x && y);
//! f(x || y);
// Bitwise operators:
x = ~y;
x = x & y;
x = x | y;
x = x ^ y;
x = x << 1;
x = x >> 1;
x = x >>> 1;
// Compound assignment:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <=> 1;
x >=> 1;
x >>>= 1;
x &= y;
x ^= y;
x |= y;
// Casting:
//! boolean b = (boolean)x;
char c = (char)x;
byte B = (byte)x;
short s = (short)x;
int i = (int)x;
float f = (float)x;
double d = (double)x;
}

void floatTest(float x, float y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
}

```

```

//! f(!x);
//! f(x && y);
//! f(x || y);
// Bitwise operators:
//! x = ~y;
//! x = x & y;
//! x = x | y;
//! x = x ^ y;
//! x = x << 1;
//! x = x >> 1;
//! x = x >>> 1;
// Compound assignment:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
//! x <= 1;
//! x >= 1;
//! x >>>= 1;
//! x &= y;
//! x ^= y;
//! x |= y;
// Casting:
//! boolean b = (boolean)x;
char c = (char)x;
byte B = (byte)x;
short s = (short)x;
int i = (int)x;
long l = (long)x;
double d = (double)x;
}
void doubleTest(double x, double y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Bitwise operators:
    //! x = ~y;
    //! x = x & y;
    //! x = x | y;

```

```

//! x = x ^ y;
//! x = x << 1;
//! x = x >> 1;
//! x = x >>> 1;
// Compound assignment:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
//! x <= 1;
//! x >= 1;
//! x >>= 1;
//! x &= y;
//! x ^= y;
//! x |= y;
// Casting:
//! boolean b = (boolean)x;
char c = (char)x;
byte B = (byte)x;
short s = (short)x;
int i = (int)x;
long l = (long)x;
float f = (float)x;
}
} //:~

```

Забележете че **boolean** е твърде ограничен. Може да му присвоявате стойностите **true** и **false** и може да проверявате за истинност, но не може да събирате и да правите каквито и да е аритметични операции изобщо.

В **char**, **byte** и **short** може да видите разширяването при аримт. операции. Всяка аритметична операция върху тях завършва с **int** резултат, който трябва явно да бъде превърнат в оригиналния тип (стесняваща конверсия която може да доведе до загуба на информация) за да може да се присвои на същия тип. С **int** стойностите, обаче, няма нужда от кастинг, понеже всичко вече е **int**. Не бъдете упоявани от мисълта че всичко е безопасно, все пак. Ако умножите две достатъчно големи **int** ще препълните резултата. Следващия пример демонстрира това:

```

//: c03:Overflow.java
// Surprise! Java lets you overflow.

public class Overflow {
    public static void main(String[] args) {
        int big = 0x7fffffff; // max int value
        prt("big = " + big);
        int bigger = big * 4;
        prt("bigger = " + bigger);
    }
    static void prt(String s) {
        System.out.println(s);
    }
} //:~

```

Извежда се това:

```

big = 2147483647
bigger = -4

```

и не получавате грешки или предупреждения от компилатора, и никакви изключения по време на изпълнение. Java е добър, но не толкова добър.

Съставните присвоявания **не** изискват кастинг за **char**, **byte** и **short**, защо и да изпълняват разширение както за аритметичните операции. От друга страна, липсата на каст операторът оправдява кода.

Може да видите, че с изключение на **boolean** всеки примитивен тип може да бъде превърнат във всеки друг примитивен тип. Напомням, че трябва да внимавате за стесняващи превръщания, при които може да се загуби информация.

Управление на изпълнението

Java използва всичките оператори за управление на хода на програмата известни от С, така че ако сте програмирали на С или C++ повечето от това, което ще видите, ще е същото. Повечето процедурни езици имат същите управляващи оператори и често се припокриват едни с други в това отношение. В Java ключовите думи включват **if-else**, **while**, **do-while**, **for** и **switch** за избор. Java не поддържа, обаче, много злеставяното **goto** (който оператор продължава да бъде най-бързият начин за решаване на някои проблеми). Все още може да се правят подобни на **goto скокове**, но са по-ограничени в сравнение с него.

true и false

Всичките оператори за условен преход използват лъжливостта или истинността за определяне пътя по който ще мине изпълнението. Пример за условен израз е **A == B**. Той използва условния оператор **==** за да види дали стойността на **A** е равна на стойността на **B**. Изразът връща **true** или **false**. Всеки от операторите за отношенията които видяхте в тази глава може да бъде използван за определяне хода на изпълнението. Забележете че Java не позволява използването на число за **boolean**, нищо че това е позволено в С и C++ (където истината е *неравно на нула* и лъжата е *равно на нула*). Ако искате да използвате **не-boolean** в **boolean** проверка като **if(a)**, трябва първо да превърнете в **boolean** стойност използвайки условен израз като **if(a != 0)**.

if-else

if-else е може би най-основният начин да се управлява хода на програмата. **else**-то не е задължително, така че може да използвате **if** в две форми:

if(Булев израз)
 оператор

или

if(Булев израз)
 оператор
else
 оператор

Условието трябва да дава Boolean резултат. Оператор значи или прост оператор след който има точка и запетая или съставен, който е група от прости оператори затворени в скоби. Винаги когато се използва думата "оператор" се предполага, че операторът може да е прост или съставен.

Като пример за **if-else** ето **test()** метод, който ще познае дали познатото число е по-малко, равно или по-голямо от истинското:

```

static int test(int testval) {
    int result = 0;
    if(testval > target)
        result = -1;
    else if(testval < target)
        result = +1;
    else
        result = 0; // match
    return result;
}

```

Прието е да се прави отместване на тялото на условния израз за да може читателят да вижда къде той започва и свършва.

return

Ключовата дума **return** има две предназначения: определя каква стойност методът ще връща (ако типът на връщане не е **void**) и незабавно връща въпросната стойност. **test()** методът по-горе може да бъде пренаписан за да се ползва от предимствата:

```

static int test2(int testval) {
    if(testval > target)
        return -1;
    if(testval < target)
        return +1;
    return 0; // match
}

```

Няма нужда от **else** понеже методът няма да продължи след изпълнението на **return**.

Итерация

while, **do-while** и **for** управляват цикленето и понякога се наричат *оператори за итерация*. Оператор **re** повтаря докато управляващия Булев-израз не получи стойност лъжа. Формата на **while** цикъл е

while(Булев-израз)

оператор

Булев-израз се изчислява преди първото изпълнение и при всяка итерация на оператор.

Ето пример който генерира случаини числа докато се удовлетвори конкретно условие:

```

//: c03:WhileTest.java
// Demonstrates the while loop

public class WhileTest {
    public static void main(String[] args) {
        double r = 0;
        while(r < 0.99d) {
            r = Math.random();
            System.out.println(r);
        }
    }
} ///:~

```

Използва се **static** методът **random()** в **Math** библиотеката, който генерира **double** стойност между 0 и 1. (включва 0, но не 1.) Условният израз за **while** казва "продължавай този цикъл

докато стойността стане 0.99 или по-голяма.” Всеки път когато стартирате тази програма ще получавате списък стойности който има различна дължина.

do-while

Формата за **do-while** е

```
do
  statement
  while(Boolean-expression);
```

Единствената разлика между **while** и **do-while** че операторът за **do-while** винаги се използва най-малко един път, даже ако изразът е “лъжа” от първия път. За **while**, ако условието е лъжа от първия път операторът никога не се изпълнява. На практика **do-while** по-малко се използва от **while**.

for

for прави инициализация преди първата итерация. След това проверява условието и, в края на всяка итерация, някаква форма на “стъпки.” Формата за **for** цикъл е:

```
for(initialization; Boolean-expression; step)
  statement
```

Кой да е от изразите *initialization*, *Boolean-expression* или *step* може да е празен (т.е. да го няма - б.пр.). Изразът се проверява преди всяка итерация и щом стане **false** изпълнението продължава със следващия **for** оператор. В края на всеки цикъл се изпълнява *step*.

for циклите обикновено се използват за “броящи” задачи:

```
//: c03>ListCharacters.java
// Demonstrates "for" loop by listing
// all the ASCII characters.

public class ListCharacters {
    public static void main(String[] args) {
        for( char c = 0; c < 128; c++)
            if (c != 26) // ANSI Clear screen
                System.out.println(
                    "value: " + (int)c +
                    " character: " + c);
    }
} //:~
```

Забележете че променливата **c** е определена в точката, където е използвана, въtre в управляващия израз на **for** цикъла, а не в началото на блока означен с отваряща кръгла скоба. Обхватът на **c** е изразът управляван от **for**.

Традиционните процедурни езици като С изискват всички променливи да бъдат деклариирани в началото на блок така че когато компилаторът генерира блок да може да алокира място за променливите. В Java и C++ може да правите декларациите си навсякъде в блока, дефинирайки променливите когато ви потрябват. Това позволява по-естествен стил на кодиране и прави кода по-лесен за разбиране.

Може да определите няколко променливи във **for** оператор, но те трябва да бъдат от един и същ тип:

```
| for(int i = 0, j = 1;
```

```
i < 10 && j != 11;  
i++, j++)  
/* тяло на for цикъла */;
```

Дефиницията на **int** във **for** цикъла е за **i** и **j**. Възможността да се дефинират променливи в управляващ израз е ограничена до **for** цикъла. Този подход не може да се използва с никакъв друг управляващ оператор.

Операторът запетая

По-рано в тази глава казах, че **операторът запетая** (не разделител, който се използва за разделяне на аргументите в списъка им) има само едно приложение в Java: в управляващия израз на **for** цикъл. И в инициализационния, и в стъпковия израз може да имате няколко оператора, разделени със запетая и те ще бъдат изчислявани последователно. Предишното парче код използва тази възможност. Ето друг пример:

```
//: c03:CommaOperator.java  
  
public class CommaOperator {  
    public static void main(String[] args) {  
        for(int i = 1, j = i + 10; i < 5;  
            i++, j = i * 2) {  
            System.out.println("i= " + i + " j= " + j);  
        }  
    }  
} ///:~
```

Ето какво се извежда:

```
i= 1 j= 11  
i= 2 j= 4  
i= 3 j= 6  
i= 4 j= 8
```

Може да се види, че и в инициализационната и в стъпковата част операторите се изпълняват последователно. Също инициализационната част може да има всякакъв брой декларации от един тип.

break и continue

Вътре в тялото на всеки итерационен блок може да се управлява изпълнението чрез **break** и **continue**. **break** спира цикъла без да изпълнява останалите оператори в него. **continue** спира изпълнението на текущата итерация и отива в началото за започване на нова итерация.

Тази програма показва пример на **break** и **continue** вътре във **for** и **while** цикли:

```
//: c03:BreakAndContinue.java  
// Demonstrates break and continue keywords  
  
public class BreakAndContinue {  
    public static void main(String[] args) {  
        for(int i = 0; i < 100; i++) {  
            if(i == 74) break; // Out of for loop  
            if(i % 9 != 0) continue; // Next iteration  
            System.out.println(i);  
        }  
        int i = 0;  
        // An "infinite loop":
```

```

while(true) {
    i++;
    int j = i * 27;
    if(j == 1269) break; // Out of loop
    if(i % 10 != 0) continue; // Top of loop
    System.out.println(i);
}
} ///:~

```

Във **for** стойността на **i** никога не става 100 поради това че **break** прекъсва цикъла когато **i** е 74. Нормално **break** се използва по този начин когато не се знае кога ще се случи условието за спиране на цикъла. **continue** операторът предава управлението в началото на цикъла за започване на нова итерация (инкрементирачки по този начин **i**) винаги щом **i** не се дели точно на 9. Когато се дели, извежда се променливата.

Втората част показва “безкраен цикъл” който на теория продължава вечно. Вътре в цикъла обаче има **break** оператор който ще прекъсне цикъла. В добавка ще видите, че **continue** отива обратно в началото без да изпълни останалото. (Това може би става когато **i** се дели на 9.) Изходът е:

```

0
9
18
27
36
45
54
63
72
10
20
30
40

```

Стойността 0 се извежда защото 0 % 9 дава 0.

Втората форма на безкраен цикъл е **for(;;)**. Компилаторът третира и **while(true)** и **for(;;)** по еднакъв начин така че кой ще използвате е въпрос на програмистки вкус.

БЕЗСЛАВНОТО “**goto**”

Ключовата дума **goto** съществува в програмните езици от самото начало. Разбира се **goto** произхожда от управлението на хода на програмата в асемблерния език: “if условие A, then скочи тук, иначе скочи там.” Ако четете асемблерският код генериран от практически всеки компилатор ще видите че управлението на програмата съдържа много скокове (преходи -б.пр.). Обаче **goto** преходите са на ниво сурс и това е което им развали репутацията. Ако програмата винаги ще скача от една точка в друга, няма ли начин така да се организира програмата, че да не скача? **goto** изпадна в истинка немилост след известната публикация “*Goto considered harmful*” на Edsger Dijkstra и оттогава премахването на **goto** е популярен спорт.

Както е типично в подобни ситуации, средата е най-доброто нещо. Проблемът не е използването на **goto** ами прекаленото използване на **goto** и в редки ситуации **goto** е най-добрият начин за структуриране на програмата.

Въпреки че **goto** е запазена дума в Java тя не се използва в езика; Java няма **goto**. Има обаче нещо което изглежда малко като преход свързано с ключовите думи **break** и **continue**. Това не

е скок а начин за излизане от цикъла. Причината често да се споменава в дискусиите за **goto** е че използва същия механизъм: етикет.

Етикетът е идентификатор следван от двоеточие както тук:

```
| label1:
```

Единственото място където етикетът е полезен в Java е точно преди итерационен оператор. Именно точно преди – нищо хубаво не донася поставянето на друг оператор между етикета и цикъла. И единстветата примина да се слага етикет е ако смятате да слагате друга итерация или условен преход в първата. **break** и **continue** нормално ще прекъснат само текущия цикъл, но когато са използвани с етикет те ще прекъснат всичко до етикета:

```
label1:  
outer-iteration {  
    inner-iteration {  
        //...  
        break; // 1  
        //...  
        continue; // 2  
        //...  
        continue label1; // 3  
        //...  
        break label1; // 4  
    }  
}
```

В случай 1 **break**-ът прекъсва вътрешната итерация и отивате във външната. В случай 2 **continue** довежда обратно в началото на вътрешната итерация. Но в случай 3 **continue label1** прекъсва вътрешната и външната итерация, винаги до **label1**. След това фактически итерациите продължават, но започвайки от външната итерация. В случай 4 **break label1** също всичко до **label1**, но не започва пак итерация. Фактически излиза и от двете итерации.

Ето пример за **for** цикли:

```
//: c03:LabeledFor.java  
// Java's "labeled for loop"  
  
public class LabeledFor {  
    public static void main(String[] args) {  
        int i = 0;  
        outer: // Can't have statements here  
        for(; true;) { // infinite loop  
            inner: // Can't have statements here  
            for(; i < 10; i++) {  
                System.out.println("i = " + i);  
                if(i == 2) {  
                    System.out.println("continue");  
                    continue;  
                }  
                if(i == 3) {  
                    System.out.println("break");  
                    i++; // Otherwise i never  
                          // gets incremented.  
                    break;  
                }  
                if(i == 7) {  
                    System.out.println("continue outer");  
                }  
            }  
        }  
    }  
}
```

```

    i++; // Otherwise i never
          // gets incremented.
    continue outer;
}
if(i == 8) {
    prt("break outer");
    break outer;
}
for(int k = 0; k < 5; k++) {
    if(k == 3) {
        prt("continue inner");
        continue inner;
    }
}
// Can't break or continue
// to labels here
}
static void prt(String s) {
    System.out.println(s);
}
} //:~

```

Използва се **prt()** който беше определен в други примери.

Забележете че **break** извежда от **for** цикъла и няма инкрементиращ израз до края на **for** цикъла. Понеже **break** пропуска инкрементирация израз инкрементирането се изпълнява директно за **i == 3**. Операторът **continue outer** в случая на **i == 7** също скочи в началото на цикъла и пропуска инкрементирането, така че тук отново се инкрементира директно.

Ето изхода:

```

i = 0
continue inner
i = 1
continue inner
i = 2
continue
i = 3
break
i = 4
continue inner
i = 5
continue inner
i = 6
continue inner
i = 7
continue outer
i = 8
break outer

```

Ако липсваше **break outer** операторът нямаше да има начин да се излезе от външния цикъл извънре на вътрешния, понеже **break** (без етикет - б.пр.) може само да прекъсне най-вътрешния цикъл. (Същото е вярно за **continue**.)

Разбира се, в случаите когато прекъсването на цикъла ще завърши и работата на метода може да се използва просто **return**.

Ето демонстрация на **break** и **continue** с етикети в **while** цикли:

```
//: c03:LabeledWhile.java
// Java's "labeled while" loop

public class LabeledWhile {
    public static void main(String[] args) {
        int i = 0;
        outer:
        while(true) {
            prt("Outer while loop");
            while(true) {
                i++;
                prt("i = " + i);
                if(i == 1) {
                    prt("continue");
                    continue;
                }
                if(i == 3) {
                    prt("continue outer");
                    continue outer;
                }
                if(i == 5) {
                    prt("break");
                    break;
                }
                if(i == 7) {
                    prt("break outer");
                    break outer;
                }
            }
        }
    }
    static void prt(String s) {
        System.out.println(s);
    }
} ///:~
```

Същите правила се прилагат за **while**:

1. Без етикет **continue** скача в началото на най-вътрешния цикъл и продължава.
2. С етикет **continue** скача на етикета и започва цикъла след този етикет.
3. **break** “скача на дъното” на цикъл.
4. С етикет **break** скача на дъното на цикъла с етикета.

Изходът на този метод прави нещата по-ясни:

```
Outer while loop
i = 1
continue
i = 2
i = 3
continue outer
Outer while loop
i = 4
i = 5
```

```
break
Outer while loop
i = 6
i = 7
break outer
```

Важно е да се запомни че единствената причина да се използват етикети в Java е когато имате вместени един в друг цикли и искате да **break** или **continue** през повече от едно ниво на вместване.

В “*goto considered harmful*” на Dijkstra той протестира фактически срещу етикетите, не срещу *goto*. Той забелязва, че броят на грешките изглежда расте с броя на етикетите в програмата. Етикетите и *goto* правят програмата мъчна за статичен анализ, понеже се въвеждат цикли в изпълнението. Забележете че в Java етикетите не страдат от този проблем, тъй като са ограничени и не могат да предават управлението по *ad hoc* маниер. Интересно е да се отбележи също, че това е случай в който една черта на езика е направена по-полезна с намаляване на мощта ⁹.

switch

switch понякога се определя като *оператор за избор*. **switch** операторът избира измежду различни парчета кон на основата на цял израз. Формата му е:

```
switch(integral-selector) {
    case integral-value1 : statement; break;
    case integral-value2 : statement; break;
    case integral-value3 : statement; break;
    case integral-value4 : statement; break;
    case integral-value5 : statement; break;
    ...
    default: statement;
}
```

Integral-selector е израз, който дава цяла стойност. **switch** сравнява резултата от *integral-selector* с всяка от *integral-value*. Ако намери съвпадение съответният *statement* (прост или съставен) се изпълнява. Ако не се намери съвпадение **default statement** се изпълнява.

Ще забележите че всеки **case** свършва с **break**, което предизвиква скок след края на тялото на **switch**. Това е конвенционалния метод за конструиране на **switch** оператора, но **break** е незадължителен. Ако е изпуснат се изпълнява кода на следващия по ред **break**. Въпреки че обикновено не се предпочита този род поведение, той може да бъде много полезен за напредналия програмист. Забележете че последния оператор, **default**, няма **break** понеже идпълнението продължава точно там, където би го продължил и **break**. Не бихте сложили **break** след **default** оператор ако считате стила за важно нещо.

Операторът **switch** е добър начин да се направи многопътна селекция (т.е избиране измежду много възможни пътища на изпълнение), но изисква селектор който дава цяла стойност като **int** или **char**. Ако искате, например, да използвате низ или плаваща запетая, това няма да работи със **switch** оператора. За нецели стойности трябва да използвате серия **if** оператори.

Ето пример в който се създават букви и се определя дали са гласни или съгласни:

```
//: c03:VowelsAndConsonants.java
// Demonstrates the switch statement

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

```

for(int i = 0; i < 100; i++) {
    char c = (char)(Math.random() * 26 + 'a');
    System.out.print(c + ": ");
    switch(c) {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
            System.out.println("vowel");
            break;
        case 'y':
        case 'w':
            System.out.println(
                "Sometimes a vowel");
            break;
        default:
            System.out.println("consonant");
    }
}
}
}
} //:~

```

Тъй като **Math.random()** генерира числа между 0 и 1 необходимо е само да се умножи по горната граница на интервала числа, който искате да получите (26 за буквите на английската азбука) и да се добави отместване за да се получи долната граница.

Въпреки че изглежда че превключването става по знаци, **switch** операторът фактически използва цялата стойност на знака. Значите с една кавичка в **case** операторите също дават цяла стойност която се използва за сравнение.

Забележете как **case**та могат да бъдат "стекирани" едно върху друго за да се получи избор на една част от кода за няколко стойности. Ще знаете също че е важно да се сложи **break** оператор във всяко отделно сравнение, иначе управлението ще продължи надолу с кода на следващото **case**.

Детайли на изчислението

Операторът:

```
| char c = (char)(Math.random() * 26 + 'a');
```

заслужава поглед по-отблизо. **Math.random()** дава **double**, така че стойността 26 се превръща в **double** за да се изпълни умножението, което също дава **double**. Това значи че '**a**' трябва да бъде превърнато в **double** за да се изпълни събирането. **double** резултатът се превръща обратно в **char** с кастинг.

Първо, какво прави касингът към **char**? Тоест, ако имате стойност 29.7 и го превръщате в **char**, 30 или 29 е стойността? Отговорът може да бъде видян в този пример:

```

//: c03:CastingNumbers.java
// What happens when you cast a float or double
// to an integral value?

public class CastingNumbers {
    public static void main(String[] args) {
        double
        above = 0.7,

```

```

below = 0.4;
System.out.println("above: " + above);
System.out.println("below: " + below);
System.out.println(
    "(int)above: " + (int)above);
System.out.println(
    "(int)below: " + (int)below);
System.out.println(
    "(char)('a' + above): " +
    (char)('a' + above));
System.out.println(
    "(char)('a' + below): " +
    (char)('a' + below));
}
} //:/~
```

Изходът е:

```

above: 0.7
below: 0.4
(int)above: 0
(int)below: 0
(char)('a' + above): a
(char)('a' + below): a
```

Така че отговорът е: кастингът от **float** или **double** към цели стойности винаги реже.

Следващият въпрос е за **Math.random()**. Дава ли той стойност от нула до единица включително '1'? На математически език $(0,1]$, или $(0,1)$, или $[0,1)$ или $[0,1]$? (Квадратната скоба значи "включва" докато кръглата значи "не включва.") Отново тестова програма дава отговора:

```

//: c03:RandomBounds.java
// Does Math.random() produce 0.0 and 1.0?

public class RandomBounds {
    static void usage() {
        System.err.println("Usage: \n\t" +
            "RandomBounds lower\n\t" +
            "RandomBounds upper");
        System.exit(1);
    }
    public static void main(String[] args) {
        if(args.length != 1) usage();
        if(args[0].equals("lower")) {
            while(Math.random() != 0.0)
                ; // Keep trying
            System.out.println("Produced 0.0!");
        }
        else if(args[0].equals("upper")) {
            while(Math.random() != 1.0)
                ; // Keep trying
            System.out.println("Produced 1.0!");
        }
        else
            usage();
    }
} //:/~
```

За да стартирате програмата пишете:

| java RandomBounds lower

или

| java RandomBounds upper

В двета случая трябва да прекъснете програмата ръчно, така че ще изглежда че **Math.random()** никога не дава 0.0 и 1.0. Експериментът тук обаче може да заблуждава. Ако считаме че има 2^{128} различни мантиси с плаваща запетая между 0 и 1, вероятността да се уцели точно може да е такава, че съответното време да е по-дълго от времето на живот на всеки компютър, а и на експериментатора. Излиза че 0.0 се включва във възможния изход на **Math.random()**. Или на математически език (0,1).

Резюме

Тази глава завършва изучаването на основните черти на повечето програмни езици: изчисления, приоритет на операторите, превръщане на типовете и селекция и итерация. Сега сте готови за стъпки които ще ви приближат към ОО програмиране. Следващата глава ще разгледа важния въпрос за инициализацията и почистването на обекти, а по-следващата фундаменталната концепция за скриване на код.

Упражнения

1. Напишете програма която извежда стойности от едно до 100.
2. Променете упражнение 1 така че програмата да завърши чрез **break** оператор при стойност 47. Опитайте с **return** след това.
3. Създайте **switch** оператор който извежда съобщение с всеки **case** и сложете **switch** вътре във **for** цикъл който опитва всеки **case**. Сложете **break** след всеки **case** и го изprobвайте, после maxнете **break**овете и вижте какво ще стане.

4: Инициализация и почистване

С напредването на компютърната революция “небезопасното” програмиране е станало главния обвиняем по скъпотията на програмирането.

Два от въпросите на безопасността са *инициализацията* и *очистването*. Много С бъгове стават поради забравянето от програмиста да инициализира променлива. Така е особено с библиотеките, където програмистът не знае как да инициализира променлива или пък че въобще трябва да го прави. Очистването пък е още по-голям проблем, защото човек рядко се сеща за някакъв елемент, който вече не му трябва. Така ресурсите използване от въпросния елемент остават заети и лесно се стига до липса на ресурси (обикновено памет).

C++ въведе концепцията за конструктор, специален метод, който се вика когато се създава обект. Java също възприе конструктора и в добавка има събирач на ресурси, който освобождава ресурсите когато обектът вече не е нужен. Тази глава разглежда въпросите на инициализацията и очистването и поддръжката им в Java.

Гарантирана инициализация с конструктора

Може да си въобразим създаване на метод наречен **initialize()** за всеки клас който създаваме. Името подсказва че методът се вика преди да се създаде обекта. За нещастие потребителят трябва да помни, че трябва да извика метода. В Java проектантът на класа може да гарантира инициализацията със специален метод, наречен **constructor**. Ако класа има конструктор, Java автоматично го вика при създаването на обект преди потребителите да могат да го пипнат с пръст. Така инициализацията е гарантирана.

Следващото предизвикателство е как да наречем този метод. Има две неща. Първото е че каквото и име да изберете то може да е в противоречие с името, което бихте използвали за някой член. Другото е, че понеже компилаторът е отговорен за викането на конструктора, трябва винаги да му е известно кой именно метод да извика. Решението в C++ изглежда най-лесно и логично и се използва също и в Java: Името на конструктора е същото като името на класа. Смислено е такъв член да се вика именно при инициализацията.

Ето простичък клас с конструктор: (Вижте стр. 63 ако имате проблем с пускането на тази програма.)

```
//: c04:SimpleConstructor.java
// Demonstration of a simple constructor
package c04;

class Rock {
    Rock() { // This is the constructor
        System.out.println("Creating Rock");
```

```
    }
}

public class SimpleConstructor {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++)
            new Rock();
    }
} //:/~
```

Сега, когато обектът е създаден:

```
| new Rock();
```

се алокира памет и се вика конструктора. Гарантирано е, че всичко ще бъде правилно инициализирано преди всянакъв друг достъп до обекта.

Забележете, че правилото всички имена на методи да започват с малка буква не се прилага за конструкторите, понеже името на конструктора трябва да съвпада с името на класа точно.

Като всеки метод конструкторът може да има аргументи които да показват как се създава обектът. Горният пример може лесно да бъде променен така, че конструкторът да има аргумент:

```
class Rock {
    Rock(int i) {
        System.out.println(
            "Creating Rock number " + i);
    }
}

public class SimpleConstructor {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++)
            new Rock(i);
    }
}
```

Аргументите на конструктора дават възможност да се управлява инициализацията. Например ако класът **Tree** има единствен числен аргумент показващ височината на дървото, обектът от тип **Tree** ще се създава така:

```
| Tree t = new Tree(12); // 12-foot tree
```

Ако **Tree(int)** е единственият конструктор, компилаторът няма да позволява създаването на **Tree** обект по никакъв друг начин.

Конструкторите премахват голям клас проблеми и правят кода лесен за четене. В предишния кодов фрагмент, например, не се вижда никакво явно викане на **initialize()** метод, който е концептуално отделен от дефиницията. В Java дефинирането и инициализацията са обединен процес – не може едното без другото.

Конструкторът е необичаен метод понеже не връща стойност. Това е различно от **void** връщана стойност в това, че методът не връща нищо но все още имате възможност да се направи да връща нещо друго. Конструкторът не връща нищо и толкова – нямаете избор. Ако имаше връщана стойност или ако можехте да изберете такава щеше да трябва компилаторът да знае какво да прави с нея.

Пренатоварване на методи

Една важна черта на всеки програмен език е използването на имената. Когато се създава обект се дава име на област от паметта. Методът е име на действие. Чрез използване на имената за описание на системата вие създавате програма която е лесна за разбиране от хората и за променяне. Много прилика на писане на проза – целта е комуникацията с читателите.

Отнасяме се към всички обекти и методи използвайки имена. Добре подбраните имена подволяват на вас и на другите да четат по-лесно кода.

Проблемът изниква когато се проектира концепцията за нюанса на човешкия език върху програмния език. Често една и съща дума се използва в различни значения – тя се пренатоварва (английската дума е "претоварва", но до сега избягвах точния превод за да не изглежда, че нещо се претоварва и ще се счупи, примерно. По-нататък ще използвам и двата превода - бел.пр.). Това е полезно особено когато се отнася за тривиални разлики. Казваме "измий лицето," "измий колата," "измий кучето." Би било тъпко да се налага да се казва "faceWash лицето," "carWash колата," и "dogWash кучето" само за да може слушателят да направи разликата. Повечето човешки езици са с излишък и ако се изпуснат няколко думи смисълът остава разбираем. Не се нуждаем от уникатни идентификатори – значението може да се изведе от контекста.

Повечето програмни езици (С в частност) изискват да има уникатен идентификатор за всяка функция. Не може да има една функция наречена **print()** за печатане на цели числа и друга наречена **print()** за такива с плаваща запетая – всяка функция изиска уникатно име.

Друг фактор налага претоварването на имената в Java: конструкторът. Понеже името на конструктора е същото като на класа може да има само едно име на конструктор. Ами ако искаме да създаваме обекта по повече начини? Да кажем искаме да създадем обект който се инициализира по стандартен начин или взема данните от файл. Трябват два конструктора, единият няма аргументи (конструкторът по подразбиране) и един с аргумент **String** който е името на файла от който ще се вземат данните за инициализацията. И двата са конструктори, така че трябва да имат едно и също име – името на класа. Така претоварването на методите е необходимо за да позволи едно име да се използва с различни типове аргументи. И освен че претоварването на методите е неизбежно при конструкторите, то е много удобно и се използва за всякакви методи.

Ето пример показващ претоварването на конструктори и обикновени методи:

```
//: c04:Overloading.java
// Demonstration of both constructor
// and ordinary method overloading.
import java.util.*;

class Tree {
    int height;
    Tree() {
        prt("Planting a seedling");
        height = 0;
    }
    Tree(int i) {
        prt("Creating new Tree that is "
            + i + " feet tall");
        height = i;
    }
    void info() {
        prt("Tree is " + height
```

```

        + " feet tall");
    }
    void info(String s) {
        prt(s + ": Tree is "
            + height + " feet tall");
    }
    static void prt(String s) {
        System.out.println(s);
    }
}

public class Overloading {
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++) {
            Tree t = new Tree(i);
            t.info();
            t.info("overloaded method");
        }
        // Overloaded constructor:
        new Tree();
    }
} //:~

```

Tree обект може да се създаде като разсад, без аргументи, или като растение от разсадник, със съществуваща височина. За да се осъществи това има два конструктора, единият без аргументи (такива ги наричаме *конструктори по подразбиране*¹) и един който приема съществуващата височина.

Може също да искате да викате **info()** методът по повече от един начин. Например със **String** аргумент може да искате извеждане на допълнително съобщение и без нищо ако нямате какво да кажете. Би било странно ако се налага да се дават различни имена на неща от една концепция. За щастие претоварването на методите позволява едно име и за двета случаи.

Различаване на претоварените методи

Ако методите имат едно име, как да определи Java кой метод имате пред вид? Правилото е просто: Всеки претоварен метод трябва да има уникален списък аргументи.

Ако помислите за секунда ще видите, че следното има смисъл: как по друг начин, освен този с аргументите, програмистът би могъл да зададе разликата между методите?

Даже промяната на реда на аргументите може да послужи за определяща разлика: (Макар че този подход обикновено се избягва, защото води до труден за поддържане код.)

```

//: c04:OverloadingOrder.java
// Overloading based on the order of
// the arguments.

public class OverloadingOrder {
    static void print(String s, int i) {
        System.out.println(
            "String: " + s +
            ", int: " + i);
    }
}

```

¹ In some of the Java literature from Sun they instead refer to these with the clumsy but descriptive name “no-arg constructors.” The term “default constructor” has been in use for many years and so I shall use that.

```

static void print(int i, String s) {
    System.out.println(
        "int: " + i +
        ", String: " + s);
}
public static void main(String[] args) {
    print("String first", 11);
    print(99, "Int first");
}
} //:~

```

Двета `print()` имат идентични аргументи, но редът им е различен и това е, което прави методите различими.

Претоварването и примитивите

Примитивните типове могат да бъдат разширявани при операции с тях и това е смущаващо във връзка с претоварването. Следващият пример показва какво става, когато примитив се даде на претоварен метод:

```

//: c04:PrimitiveOverloading.java
// Promotion of primitives and overloading

public class PrimitiveOverloading {
    // boolean can't be automatically converted
    static void prt(String s) {
        System.out.println(s);
    }

    void f1(char x) { prt("f1(char)"); }
    void f1(byte x) { prt("f1(byte)"); }
    void f1(short x) { prt("f1(short)"); }
    void f1(int x) { prt("f1(int)"); }
    void f1(long x) { prt("f1(long)"); }
    void f1(float x) { prt("f1(float)"); }
    void f1(double x) { prt("f1(double)"); }

    void f2(byte x) { prt("f2(byte)"); }
    void f2(short x) { prt("f2(short)"); }
    void f2(int x) { prt("f2(int)"); }
    void f2(long x) { prt("f2(long)"); }
    void f2(float x) { prt("f2(float)"); }
    void f2(double x) { prt("f2(double)"); }

    void f3(short x) { prt("f3(short)"); }
    void f3(int x) { prt("f3(int)"); }
    void f3(long x) { prt("f3(long)"); }
    void f3(float x) { prt("f3(float)"); }
    void f3(double x) { prt("f3(double)"); }

    void f4(int x) { prt("f4(int)"); }
    void f4(long x) { prt("f4(long)"); }
    void f4(float x) { prt("f4(float)"); }
    void f4(double x) { prt("f4(double)"); }

    void f5(long x) { prt("f5(long)"); }
    void f5(float x) { prt("f5(float)"); }
}

```

```

void f5(double x) { prt("f5(double)"); }

void f6(float x) { prt("f6(float)"); }
void f6(double x) { prt("f6(double)"); }

void f7(double x) { prt("f7(double)"); }

void testConstVal() {
    prt("Testing with 5");
    f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5);
}

void testChar() {
    char x = 'x';
    prt("char argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}

void testByte() {
    byte x = 0;
    prt("byte argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}

void testShort() {
    short x = 0;
    prt("short argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}

void testInt() {
    int x = 0;
    prt("int argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}

void testLong() {
    long x = 0;
    prt("long argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}

void testFloat() {
    float x = 0;
    prt("float argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}

void testDouble() {
    double x = 0;
    prt("double argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}

public static void main(String[] args) {
    PrimitiveOverloading p =
        new PrimitiveOverloading();
    p.testConstVal();
    p.testChar();
    p.testByte();
    p.testShort();
    p.testInt();
    p.testLong();
    p.testFloat();
    p.testDouble();
}

```

```
    }
} //:/~
```

Ако разгледате изхода от тази програма ще видите, че константата 5 се третира като **int**, така че ако преторавен метод има за аргумент **int** той се използва. Във всички други случаи, ако имате даннов тип по-малък от този в метода, той се разширява. **char** дава малко по-различен ефект, понеже ако не се намери точно **char** съвпадение той се разширява до **int**.

Какво става, ако вашият аргументи е *по-голям* от очаквания от претоварен метод? Отговорът се дава от модификация на предишната програма:

```
//: c04:Demotion.java
// Demotion of primitives and overloading

public class Demotion {
    static void prt(String s) {
        System.out.println(s);
    }

    void f1(char x) { prt("f1(char)"); }
    void f1(byte x) { prt("f1(byte)"); }
    void f1(short x) { prt("f1(short)"); }
    void f1(int x) { prt("f1(int)"); }
    void f1(long x) { prt("f1(long)"); }
    void f1(float x) { prt("f1(float)"); }
    void f1(double x) { prt("f1(double)"); }

    void f2(char x) { prt("f2(char)"); }
    void f2(byte x) { prt("f2(byte)"); }
    void f2(short x) { prt("f2(short)"); }
    void f2(int x) { prt("f2(int)"); }
    void f2(long x) { prt("f2(long)"); }
    void f2(float x) { prt("f2(float)"); }

    void f3(char x) { prt("f3(char)"); }
    void f3(byte x) { prt("f3(byte)"); }
    void f3(short x) { prt("f3(short)"); }
    void f3(int x) { prt("f3(int)"); }
    void f3(long x) { prt("f3(long)"); }

    void f4(char x) { prt("f4(char)"); }
    void f4(byte x) { prt("f4(byte)"); }
    void f4(short x) { prt("f4(short)"); }
    void f4(int x) { prt("f4(int)"); }

    void f5(char x) { prt("f5(char)"); }
    void f5(byte x) { prt("f5(byte)"); }
    void f5(short x) { prt("f5(short)"); }

    void f6(char x) { prt("f6(char)"); }
    void f6(byte x) { prt("f6(byte)"); }

    void f7(char x) { prt("f7(char)"); }

    void testDouble() {
        double x = 0;
        prt("double argument:");
    }
}
```

```

f1(x);f2((float)x);f3((long)x);f4((int)x);
f5((short)x);f6((byte)x);f7((char)x);
}
public static void main(String[] args) {
    Demotion p = new Demotion();
    p.testDouble();
}
} //:~

```

Тук методите получават по-малки типове. Ако вашият аргумент е по-голям тип трябва да го *cast*-нете към необходимия тип използвайки името в скоби. Ако не направите това, компилаторът ще издаде съобщение за грешка.

Трябва да сте предупредени че това е **стесняващо преобразуване**, което значи че може да се загуби информация при кастинга. Това е причината компилаторът да ви кара вие да го правите – за да отбележи стесняващата конверсия (и накара програмиста да вземе решението - б.пр.).

Претоварване на връщаните стойности

Често се чудят “Защо само имената на класовете и списъците на аргументите? Защо да не различаваме методите по връщаните стойности?” Например два метода които имат еднакви имена и аргументи могат да бъдат ясно отличени:

```

void f0 {}
int f0 {}

```

Това работи добре докато компилаторът може да определи каквото му трябва от контекста, като **int x = f()**. Може обаче да се извика метод и да се игнорира връщаната стойност; това често се нарича *извикване на метод заради страничния му ефект* тъй като въобще не ви трябва връщаната стойност а се интересувате само от страничния ефект. Например викати метода по следния начин:

```
f0;
```

Как би могъл компилаторът да определи кое **f()** ще се вика? Как би могъл да определи това и кой да е читател? Поради този сорт проблеми не може да се различават в езика методи по връщаната стойност.

Конструктори по подразбиране

Както беше споменато преди, конструкторът по подразбиране е този без аргументи. Ако създадете клас без конструктори компилаторът ще създаде автоматично конструктор по подразбиране заради вас. Например:

```

//: c04:DefaultConstructor.java

class Bird {
    int i;
}

public class DefaultConstructor {
    public static void main(String[] args) {
        Bird nc = new Bird(); // default!
    }
} //:~

```

Редът

```
| new Bird();
```

създава нов обект и вика конструктор по подразбиране, въпреки че такъв не е явно деклариран. Без него нямаше да имаме начин да построим нашия обект. Ако обаче дефинирате никакви конструктори (с или без аргументи), компилаторът няма да синтезира конструктор по подразбиране вместо вас:

```
class Bush {  
    Bush(int i) {}  
    Bush(double d) {}  
}
```

Ако сега напишем:

```
| new Bush();
```

компилаторът ще се оплаква, че не може да намери подходящ конструктор. Все едно че ако не запишете никакви конструктори компилаторът казва "Трябва да използвате **НЯКАКЪВ** конструктор, така че нека да направя един вместо вас." Но ако напишете конструктор компилаторът казва "Написали сте конструктор значи знаете какво правите; ако не сте сложили конструктор по подразбиране това ще е защото не искате да има."

Ключовата дума **this**

Ако имате два обекта от един и същ тип наречени **a** и **b** може да се чудите как ли ще се вика **f()** за тези два обекта:

```
class Banana { void f(int i) { /* ... */ } }  
Banana a = new Banana(), b = new Banana();  
a.f(1);  
b.f(2);
```

Ако има само един метод наречен **f()**, как той знае дали е викан от **a** или **b**?

За да се позволи да се пише код по удобен ОО синтаксис в който се "изпраща съобщение на обект," компилаторът върши скрито работа заради вас. Има тайна в първия аргумент на **f()** и тя е, че този аргумент е манипулятор на обекта, с който работите. Така че двете викания на метода стават нещо като:

```
Banana.f(a,1);  
Banana.f(b,2);
```

Това става вътрешно и не може да се напишат изразите и компилаторът да ги възприеме, но се дава идея за нещата.

Да предположим че сме вътре в обект и искачме да вземем манипулятора му. Тъй като той се дава скрито от компилатора, няма идентификатор за него. Има обаче ключова дума за тази цел: **this**. Ключовата дума **this** – която може да се използва само вътре в метод – дава манипулятора на обекта, от който методът е извикан. Това е манипулятор и може да се третира като всеки друг. Помнете, че ако викате метод от ваш клас отвътре на друг метод на ваш клас не е необходимо да използвате **this**; просто викате метода. Текущия **this** манипулятор автоматично се използва за другия метод. Така може да се напише:

```
class Apricot {  
    void pick() { /* ... */ }  
    void pit() { pick(); /* ... */ }  
}
```

В **pit()** може да се напише **this.pick()** но не е необходимо. Компилаторът го прави автоматично. Ключовата дума **this** се използва само в онези специални случаи, когато трябва

явно да се използва манипуляторът на текущия обект. Например често се използва в **return** операторите когато се иска да се върне манипулятор към текущия обект:

```
//: c04:Leaf.java
// Simple use of the "this" keyword

public class Leaf {
    private int i = 0;
    Leaf increment() {
        i++;
        return this;
    }
    void print() {
        System.out.println("i = " + i);
    }
    public static void main(String[] args) {
        Leaf x = new Leaf();
        x.increment().increment().print();
    }
} //:~
```

Понеже **increment()** връща манипулятор към текущия обект чрез ключовата дума **this** многократно може да се изпълнят оператори върху същия обект.

Викане на конструктори от конструктори

Когато се пишат няколко конструктора за един клас понякога е удобно да се вика конструктор от конструктор, за да се избегне дублирането на код. Това може да се направи с ключовата дума **this**.

Нормално кагато се напише **this** то е в смисъл на "този обект" или "текущия обект" и произвежда манипулятор към текущия обект. В конструктор **this** има друг смисъл ако дадете списък от аргументи: тя прави явно викане на конструктор с този списък аргументи. Така имате праволинеен начин да викате конструктор от конструктор:

```
//: c04:Flower.java
// Calling constructors with "this"

public class Flower {
    private int petalCount = 0;
    private String s = new String("null");
    Flower(int petals) {
        petalCount = petals;
        System.out.println(
            "Constructor w/ int arg only, petalCount= "
            + petalCount);
    }
    Flower(String ss) {
        System.out.println(
            "Constructor w/ String arg only, s=" + ss);
        s = ss;
    }
    Flower(String s, int petals) {
        this(petals);
    }
    //! this(s); // Can't call two!
    this.s = s; // Another use of "this"
    System.out.println("String & int args");
}
```

```

Flower() {
    this("hi", 47);
    System.out.println(
        "default constructor (no args)");
}
void print() {
//! this(11); // Not inside non-constructor!
    System.out.println(
        "petalCount = " + petalCount + " s = " + s);
}
public static void main(String[] args) {
    Flower x = new Flower();
    x.print();
}
} ///:~

```

Конструкторът **Flower(String s, int petals)** показва, че докато можете да викате един конструктор чрез **this**, не може да викате два. В добавка викането на конструктора трябва да е първото нещо, което се прави, иначе се получава съобщение за грешка.

Този пример също показва друг начин за използване на **this**. Тъй като името на аргумента **s** и името на члена-данни **s** е същото, има двусмислие. То може да се разреши като се напише **this.s** за члена данни. Често ще видите тази форма в Java код и тя е използвана множество пъти в тази книга.

В **print()** може да видите, че компилаторът не ще ви позволи да викате конструктор от друг метод освен от конструктор.

Значението на **static**

С ключовата дума **this** на ум може по-пълно да се разбере значението на **static**. То е че няма **this** за конкретния метод. Не може да се викат **не-static** извънре на **static** методи² (макар че обратното е възможно) и може да се вика **static** за самия клас, без никакъв обект. Фактически най-вече за това са и **static** методите. Това е като да се прави еквивалент на глобална функция (в C). Глобалните функции не са позволени в Java и писането на **static** вътре в клас позволява да са достъпни други **static** методи и **static** полета.

Някои хора спорят че **static** методите не са ОО понеже имат семантиката на глобалните функции; със **static** метод не изпращате съобщение на обект, понеже няма **this**. Това е аргумент и ако установите че използвате много статични методи вероятно ще е добре да си преосмислите стратегията. Обаче **static** нещата са прагматични и понякога истински се нуждаете от тях, така че и да са и да не са "правилно ОО" въпроса ще оставим на теоретиците. Разбира се, даже Smalltalk има еквиваленти с неговите "class methods."

Почестване: финализация и събиране на боклука

Програмистите знаят важността на инициализацията, но често забравят важността на начина на приключване. Най-после, кой има нужда да чисти променлива **int?** С библиотеките, обаче "зарязването" на обект когато сте свършили с него не е винаги безопасно. Разбира се, Java има боклучар който да освободи паметта на обектите, когато вече не са нужни. Сега да

² The one case [wherein which](#) this is possible occurs if you pass a handle to an object into the **static** method. Then, via the handle (which is now effectively **this**), you can call non-**static** methods and access non-**static** fields. But typically if you want to do something like this you'll just make an ordinary, non-**static** method.

предположим много специален и необичаен случай. Да кажем че обект заема "специална" памет без използване на **new**. Боклучарят знае само как се освобождава памет заета с **new**, така че не знае как се освобождава "специална" памет. За такива случаи Java предлага метод наречен **finalize()** който може да дефинирате за ваш клас. Ето как се предполага да работи този метод. Когато боклучарят е готов да освободи паметта на обекта ви, той първо вика **finalize()** и чак на следващото си минаване освобождава паметта. Така че ако речете да използвате **finalize()** той ви дава възможност да изпълните важни действия по времето на събирането на боклука.

Това съдържа потенциални неприятности, понеже някои програмисти, специално C++ програмисти, биха могли в началото да събркат **finalize()** с деструкторите в C++, които са функции които се викат винаги когато се разрушава обект. Важно е да се прави разлика между C++ и Java в този случай, понеже в C++ обектите винаги биват разрушавани (в програма без бъгове), докато в Java обектите не винаги минават през боклучаря. Или, с други думи:

Събирането на боклука не е разрушаване.

Ако запомните това, няма да имате тревоги. Това значи че трябва да предприемете някои действия сами преди събирането на боклука, когато вече не се нуждаете от даден обект. Java няма деструктори или подобна концепция, така че трябва вие да създадете подходящ метод за тази дейност. Например да предположим че в процеса на създаване вашият обект се изобразява на екрана. Ако явно не изтриете образа на екрана той би могъл никога да не бъде почищен. Ако сложите подходящите неща за изтриването във **finalize()**, тогава ако обектът мине през боклучаря образът ще се махне, но ако не — ще остане. Така че второто нещо за запомняне е:

Обектите ви може и да не бъдат почиствани от боклучаря.

Може да установите че в програмата ви никога не се освобождава памет понеже тя никога не заема твърде много памет. Ако вашата програма завърши (нормално - б.пр.) и боклучарят не е освобождавал въобще памет, цялата на веднъж заемана от програмата памет ще бъде върната на операционната система при изхода от програмата. Това е хубаво нещо, защото събирането на боклука отнема допълнително време и няма защо да се прави, ако не е необходимо.

За какво е **finalize()**?

Може да помислите в този момент, че никога няма да използвате **finalize()** като метод за обща употреба в почистването. Какво хубаво дава той?

Третото нещо за запомняне е:

Събирането на боклука е само за паметта.

Тоест боклучарят съществува само за да освободи паметта, която вашата програма вече не използва. Така че всяка активност асоциирана с боклучаря, най-вече **finalize()** метода, трябва да бъде само за паметта и нейното освобождаване.

Значи ли това че ако вашият обект съдържа други обекти **finalize()** трябва явно да ги освобождава? Ами, не – събирачът на боклук се грижи за освобождаването на обектите без значение как са създадени. Това показва, че нуждата от **finalize()** е ограничена до специални случаи, в които обект може да заеме памет по начин различен от създаването на обект. Но, може да забележите вие, всичко в Java е обект и как би станало това?

Би могло да се каже, че **finalize()** съществува, защото може да направите нещо като C-подобните начини за алокиране на памет Java. Това може да се случи най-вече чрез *native*

methods, които са начин да се вика не-Java код от Java. (Те се обсъждат в приложение A.) С и C++ са единствените езици поддържани в момента от собствените (т.е. на езика на конкретната машина, на която се изпълняват - б.пр.) методи, но тъй като те могат да викат подпрограми на всякакви езици, фактически може да се вика всичко. Вътреш в не-Java кода, фамилията **malloc()** функции на C може да бъде викана за заемане на памет докато не извикате **free()** тази памет няма да бъде освободена, предизвиквайки съответните проблеми. Разбира се, **free()** е C и C++ функция, така че ще я викате в нативен метод във **finalize()**.

След прочитането на това може би ще си помислите, че няма да използвате **finalize()** много. Прави сте, това не е мястото за обикновеното почистване. Така че къде ще го правим нормално?

Вие трябва да почистите

За да се почисти обект, потребителят му трябва да предприеме съответните действия, когато е необходимо. Това звучи много праволинейно, но прилича на концепцията в C++ за деструкторите. В C++ всички обекти се разрушават. Или по-скоро трябва да се разрушават. В C++ обектът се създава като локален, т.е на стека (невъзможно в Java), после деструкторът се изпълнява при затварящата фигурана ("голяма") скоба където свършва обхватът на обекта. Ако обектът е бил създаден с **new** (както в Java) деструкторът се вика когато програмистът извика C++ операторът **delete** (който не съществува в Java). Ако програмистът забрави, деструкторът никога не се вика и стават "изтичания" на памет, плюс че другите части на обекта никога не се почистват.

В контраст Java не позволява да се създават локални обекти – винаги трябва да използвате **new**. Но в Java няма "delete" за освобождаване на обекти понеже има събирач на боклук. Така че от опростенческа гледна точка би могло да се каже, че понеже има събирач на боклук Java няма деструктори. С напредването на четенето на тази книга, обаче, ще се уверите, че боклучарят не премахва нуждата и полезността на деструкторите. (И никога няма да викате **finalize()** директно, така че това не е подходящото решение.) Ако искате някакви други операции преди излизане на обекта от обхвата все още трабва да извикате метод в Java, който е еквивалентът на C++ деструктора без удобството му.

Едно от нещата за които **finalize()** може да бъде полезен е наблюдаването на процеса на събиране на боклuka. Следващият пример проследява какво става и резюмира досегашното изложение за боклучаря:

```
//: c04:Garbage.java
// Demonstration of the garbage
// collector and finalization

class Chair {
    static boolean gcrun = false;
    static boolean f = false;
    static int created = 0;
    static int finalized = 0;
    int i;
    Chair() {
        i = ++created;
        if(created == 47)
            System.out.println("Created 47");
    }
    protected void finalize() {
        if(!gcrun) {
            gcrun = true;
            System.out.println(
                "Beginning to finalize after " +
```

```

        created + " Chairs have been created");
    }
    if(i == 47) {
        System.out.println(
            "Finalizing Chair #47, " +
            "Setting flag to stop Chair creation");
        f = true;
    }
    finalized++;
    if(finalized >= created)
        System.out.println(
            "All " + finalized + " finalized");
}
}

public class Garbage {
    public static void main(String[] args) {
        if(args.length == 0) {
            System.err.println("Usage: \n" +
                "java Garbage before\n or:\n" +
                "java Garbage after");
            return;
        }
        while(!Chair.f) {
            new Chair();
            new String("To take up space");
        }
        System.out.println(
            "After all Chairs have been created:\n" +
            "total created = " + Chair.created +
            ", total finalized = " + Chair.finalized);
        if(args[0].equals("before")) {
            System.out.println("gc():");
            System.gc();
            System.out.println("runFinalization():");
            System.runFinalization();
        }
        System.out.println("bye!");
        if(args[0].equals("after"))
            System.runFinalizersOnExit(true);
    }
} //:~

```

Горната програма създава много **Chair** обекти и когато по някое време боклучарят се включи тя спира да ги създава **Chairs**. Доколкото боклучарят може да се включи по кое да е време не може да се каже точно кога това ще стане, така че има флаг наречен **gcrun** за индикация кога събирането е започнало. Втори флаг **f** е начинът за **Chair** да каже на **main()** цикъла да спре да прави обекти. И двата флага са сложени във **finalize()**, който се вика при събирането на боклука.

Две други **static** променливи, **created** и **finalized**, пазят броя на създадените **обекти** и тези, които са финализирани от събираща на боклук. Накрая, всеки **Chair** има собствен (не-**static**) **int i** така че знае кой номер е. Когато **Chair** номер 47 се финализира флагът се слага **true** за да накара да спре процесът на създаване на **Chair**.

Всичко това става в **main()**, в цикъла

```
while(!Chair.f) {
    new Chair();
    new String("To take up space");
}
```

Бихте могли да се чудите как този цикъл въобще ще свърши, понеже няма нищо вътре което да мени стойността на `Chair.f`. Обаче `finalize()` ще го промени, накрая, като се стигне до обект 47.

Създаването на `String` обекти е просто създаване на повече боклук, така че да се включи боклучарят и да ги изрита, което той и ще направи когато се почувства нервен по въпроса за достатъчността на незаетата още памет.

Когато стартирате програмата давате аргумент на командния ред "преди" или "след." "before" ще идвика `System.gc()` метода (за да стартира събирането на боклука) заедно със `System.runFinalization()` за да стартира финализаторите. Тези методи бяха достъпни в Java 1.0, но `runFinalizersOnExit()` методът който се вика чрез "след" аргумента го има само в Java 1.1³ и по-нататък. (Забележете че може да викате този метод когато и да е през време на изпълнението на програмата и изпълнението на финализаторите не зависи от събирането на боклука при това положение).

Предишната програма показва че в Java 1.1 обещанието че финализаторите ще се стартират винаги е изпълнено, но само ако изрично ги посочите да се изпълнят. Ако използвате аргумент който не е "преди" или "след" (като "никакъв"), няма да има финализационен процес и ще се получи изход като този:

```
Created 47
Beginning to finalize after 8694 Chairs have been created
Finalizing Chair #47, Setting flag to stop Chair creation
After all Chairs have been created:
total created = 9834, total finalized = 108
bye!
```

Така не всички финализатори ще се стартират до свършването на програмата.⁴ За да се извикат с необходимост може да се извика `System.gc()` следван от `System.runFinalization()`. Това ще разруши всички обекти които не са в употреба вече в този момент. Лошото в това е че викате `gc()` преди `runFinalization()`, което изглежда да противоречи на документацията на Sun която твърди, че финализаторите се стартират първи, а паметта се освобождава после. Обаче ако извикате `runFinalization()` първо, а после `gc()`, финализаторите няма да се изпълнят.

Една причина Java 1.1 може би да пропуска по предположение финализаторите е, че изпълнението им е скъпо. Който и от двата подхода да използвате може да забележите по-дълги задръжки, отколкото ако се пусне боклучарят без финализаторите.

Как работи събирачът на боклук

Ако сте работили с език където алокирането на памет за обекти на хийпа е скъпо естествено е да предполагате че схемата в Java за алокиране на всичко (освен привитиви) на хийпа е скъпа. Оказва се обаче, че събирачът на боклуха може да окаже значително облекчаващо влияние върху процеса на създаване на обектите. Това може да звучи малко тъло отначало –

³ Unfortunately, the implementations of the garbage collector in Java 1.0 would never call `finalize()` correctly. As a result, `finalize()` methods that were essential (such as those to close a file) often didn't get called. The documentation claimed that all finalizers would be called at the exit of a program, even if the garbage collector hadn't been run on those objects by the time the program terminated. This wasn't true, so as a result you couldn't reliably expect `finalize()` to be called for all objects. Effectively, `finalize()` was useless in Java 1.0.

⁴ By the time you read this, some Java Virtual Machines may show different behavior.

че освобождането на памет може да засегне алокирането – но това е начинът на работа на някои JVMашини и означава, че създаването на обекти в нийпа в Java може да бъде сравнимо по бързина със създаването на обекти на стека в други езици.

Наприме може да се мисли че хийпът в C++ е нещо като двор, в който всеки обект си заделя негово си местенце. Този недвижим имот може да бъде изоставен по-късно и повторно използван. В някои JVMашини хийпът на Java е доста различен; той по-прилича на конвейерна лента, която се придвижва напред с всяко създаване на обект. Това значи че алокирането на памет е забележително бързо. „Хийп указателят“ просто се мести напред върху девствена територия, така че ефективно е същото като алокирането в стека в C++. (Разбира се, има малко редици за счетоводството, но не и търсене в паметта. (И, разбира се, самата виртуална машина не бива да се забравя при сравняването с C++ - б.пр.))

Може да забележите сега, че хийпът не е конвейерна лента и ако се третира по този начин ще се получи голямо прехвърляне на страници (странициранието е голям пробив в скоростта) и недостиг на страници. Трикът е, че събирачът на боклук влиза в ролята си и мести „указателя на хийпа“ по-близо до началото на конвейера и по-далеч от издънването на програмата вследствие page fault и невъзможност да се излезе от него. Боклучарят реорганизира нещата и проводи възможно прилагането на модела на безкрайния свободен хийп при алокирането на памет.

За да разберете как става това, трябва да имате по-добра представа как различните боклучарски (GC) схеми работят. Проста но бавна GC техника е броене на позоваванията. При нея всеки обект има брояч на позоваванията и всеки път когато се присъедини манипулатор към обекта броячът се увеличава. Всеки път когато манипулатор излезе от обхвата или стане **null** броячът се намалява. Така управлението на броячите довежда до малки допълнителни разходи за следене какво става през живота на програмата. Боклучарят преглежда целия списък на обектите и когато намери обект с брояч сочещ нула освобождава паметта му. Единия недостатък е, че понеже обектите може да се позовават един на друг и сами на себе си може да се случи броячът да не е нула, а обектът да си е за изчистване. Улавянето на такова самопозоваване изисква значителна допълнителна работа от боклучаря. Броенето на позоваванията често се използва за обясняване на събирането на боклука но не изглежда да се използва в много реализации на JVM.

При по-бързи схеми събирането не е основано на броене на позоваванията. Вместо това се използва идеята че от всеки жив (продължаващ да бъде използван) обект може да се проследи път до валиден манипулатор или на стека или в статична памет. Веригата може да върви през няколко слоя обекти. Така ако проследите всички манипулатори които са на стека и в статичната памет ще стигнете и до всички живи обекти. За всеки манипулатор трябва да се проследи пътя до обекта и след това да се видят манипулаторите в обекта, които ще доведат до други обекти, и т.н., докато се обхване цялата паяжина започваща от споменатия манипулатор на стека или в статичната памет. Всеки срещнат по пътищата обект сигурно е жив. Забележете, че няма проблем със самозатворени групи — те просто не се броят въобще и биват почиствани целите.

При описания подход JVM използва адаптина схема за събиране на боклука и това което се прави с живите обекти зависи от конкретната текуща конфигурация. Един от вариантите е **спри-и-копирай**. Тоест по причини които по-късно ще станат ясни, програмата първо се спира (тази не е фонова схема за събиране на боклука). После всеки намерен жив обект се копира в друг хийп, а старата памет се освобождава цялата. В добавка, понеже се копират на ново място, обектите се разполагат един до друг, т.е. новата памет е компактно заета (и старата памет изцяло, т.е. бързо, се освобождава, както вече се спомена).

Разбира се, когато обект се мести на друго място всички манипулатори които го сочат трябва да се променят. Лесно се променя първоначалният манипулатор, но има и други манипулатори които могат да сочат към обекта и те да се открият по време на „разходката.“ Те се променят когато бъдат намерени (можем да си представим една таблица която изобразява старите адреси в новите).

Има две неща които правят тези тъй наречени "copy collectors" (копиращи боклучари, във възприетата в превода терминология - б.пр.) неефективни. Първото е, че има два хийпа и обектите се преточват от единия в другия, на практика използвайки два пъти повече памет. Някои JVM машини се оправят с това като алокират памет на порции и работят с тях, а алокират нови само ако е необходимо.

Второто е копирането. Веднъж като се стабилизира програмата може да генерира малко или никакъв боклук. Напук на това копи боклучарят ще продължи да копира всичко от една място на друго, което е прахосничество. За да предотвратят това някои JVM машини детектират когато не се прави нов боклук и превключват на друга схема (това е "адаптивната" част). Тази друга схема се нарича *mark and sweep* и тя е която се използваше през цялото време в ранните версии на виртуалната машина на Sun. За обща употреба схемата "маркирай и измети" е доста бавна, но когато не се генерира много или въобще боклук тя е много добра.

"Маркирай и измети" следва същата логика на проследяване на обектите както преди. Обаче при намиране на жив обект в него се установява флаг, с което той се отбелязва, но още не се прави почистването. То става чак като завърши процесът на маркирането. По време на метенето мъртвите обекти се освобождават. Не се прави и компактинг, обаче, така че ако боклучарят иска да упълни паметта ще го направи в отделен пас.

Името "спри и копирай" е свързано с идеята че събирането на боклука *не* е фонов процес; програмата е спряна пре време на GC. В литературата на Sun може да се намерят много споменавания на събирането на боклука като фонов процес с нисък приоритет, но изглежда че това е бил теоретичен експеримент без реализация, най-малкото не в ранните версии на JVM на Sun. Вместо това боклучарят на Sun се задействаше при недостиг на памет. В добавка, "маркирай и измети" изисква спиране на програмата.

Както се спомена по-рано, във JVM описвана тук паметта се заема на големи блокове. Ако се алокира голям обект той взема свой собствен блок. Стриктното спри-и-копирай изисква копирането на всеки жив обект на новото място преди да може да се освободи старото, което резултира в много памет. С блоковете GC типично може да използува мъртвите блокове за копиране в трях в процеса на работа. Всеки блок има брояч на поколението за да се знае дали е жив. В нормалния случай само блоковете създадени след последния GC се състяват; всички останали отбелязват с брояча си ако на тях има позоваване от някъде. Това обслужва нормалния случай на множество късоживеещи обекти. Периодично се прави пълно измитане – големите обекти още не са копирани и блоковете къито съдържат малки обекти са копирани и състени. JVM следи ефективността на GC и ако то се окаже загуба на време защото повечето обекти са дългоживеещи, превключва се на маркирай-и-измети. Подобно JVM следи доколко е ефективно маркирай-и-измети и ако хийпът се фрагментира се превключва обратно на спри-и-копирай. Това е мястото където работи "адаптивната част", така че завършваме на един дъх с фразата: "адаптивна генерационна спри-и-копирай маркирай-и-измети."

Има и други допълнителни възможности за ускоряване в JVM-та. Една особено важна включва лоудера и Just-In-Time (JIT) компилатора. Когато трябва да се натовари клас (типично: когато за пръв път ще се създава обект от този клас), **.class** файлът се намира и кодът се качва в паметта. В тази точка еденият подход е просто да се JIT всичкия код, но това има два недостатъка: заема малко повече време, което по продължение на програмата може да се натрупа; и увеличава дължината на изпълнимия модул (байт кодовете са значително по-компактни от разширения JIT код) и така може да се предизвика пейджинг, което определено забавя програмата. Алтернативен подход е **мързеливият**, което значи че кодът не се JIT компилира докато това не стане необходимо. Така кодът който никога не се изпълни може и никога да не се JIT компилира.

Инициализация на членове

Java излиза от пътя си за да гарантира че променливите са инициализирани преди да бъдат използвани. В случая на променливи които са определени локално в метод тази гаранция идва във формата на грешка при компилация. Така ако напишем:

```
void f() {  
    int i;  
    i++;  
}
```

Ще получим съобщение за грешка в което се казва че **i** може да не е било инициализирано. Разбира се, компилаторът би могъл да даде на **i** стойност по подразбиране, но е повороятно това да е програмистка грешка, която ще се открие по тази начин. Заставянето на програмиста да даде стойността най-вероятно ще хване бъг.

Ако примитивът е член-данни на клас, обаче, нещата са по-различни. Тъй като всеки метод би могъл да инициализира данните, не е практично да се кара програмистът да го прави. Не е безопасно и да се оставя каквато се случи стойност, така че всеки член се гарантира да има определена начална стойност. Тези стойности могат да се видят тук:

```
//: c04:InitialValues.java  
// Shows default initial values  
  
class Measurement {  
    boolean t;  
    char c;  
    byte b;  
    short s;  
    int i;  
    long l;  
    float f;  
    double d;  
    void print() {  
        System.out.println(  
            "Data type    Initial value\n" +  
            "boolean      " + t + "\n" +  
            "char         " + c + "\n" +  
            "byte         " + b + "\n" +  
            "short        " + s + "\n" +  
            "int          " + i + "\n" +  
            "long          " + l + "\n" +  
            "float         " + f + "\n" +  
            "double        " + d);  
    }  
}  
  
public class InitialValues {  
    public static void main(String[] args) {  
        Measurement d = new Measurement();  
        d.print();  
        /* In this case you could also say:  
        new Measurement().print();  
        */  
    }  
} //:/~
```

Програмата извежда следното:

Data type	Initial value
boolean	false
char	
byte	0
short	0
int	0
long	0
float	0.0
double	0.0

char стойността е нула (нулева стойност, не цифрата нула - б.пр.) и затова не се вижда.

Ще видите по-нататък че при декларация на манипулятор в метод той се инициализира с нула по начало.

Може да се види, че макар и да не се дават стойности, променливите се инициализират. Така че най-малкото го няма наказанието работа с неинициализирани променливи.

Задаване на инициализация

Какво става ако искаме да зададем начална стойност? Един прям начин е просто да присвоим стойността в мястото, където се обявява променливата в класа. (Забележете че това не може да се направи в C++, макар че C++ новаците винаги се опитват.) Ето дефинициите на полета за класа **Measurement** променени за да получават начални стойности:

```
class Measurement {  
    boolean b = true;  
    char c = 'x';  
    byte B = 47;  
    short s = 0xff;  
    int i = 999;  
    long l = 1;  
    float f = 3.14f;  
    double d = 3.14159;  
    // ...
```

Също може да се инициализират и не-примитиви по този начин. Ако **Depth** е клас може да вмъкнете променлива и да я инициализирате така:

```
class Measurement {  
    Depth o = new Depth();  
    boolean b = true;  
    // ...
```

Ако не сте дали на **o** начална стойност и продължите напред и се опитате така да го използвате, ще получите грешка по време на изпълнение от тип **изключение** (разгледан в глава 9).

Може даже да извикате метод за даване на начална стойност:

```
class CInit {  
    int i = f();  
    // ...  
}
```

Този метод може да има аргументи, разбира се, но те не могат да бъдат неинициализирани още променливи. Това е допустимо:

```
class CInit {  
    int i = f();  
    int j = g(i);  
    //...  
}
```

Но това не е:

```
class CInit {  
    int j = g(i);  
    int i = f();  
    //...  
}
```

Тук компилаторът, съвсем на място, се оплаква от позоваване напред, понеже то се отнася за реда на инициализация, а не за компиляцията.

Този подход към инициализацията е прост и праволинеен. Той има ограничението че *всеки* обект от тип **Measurement** ще получи същите начални стойности. Понякога това е точно което искаме, но в други случаи трябва повече гъвкавост.

ИНИЦИАЛИЗАЦИЯ В КОНСТРУКТОРИТЕ

Конструкторът може да се използва за задаване на начални стойности и това е по-гъвкав начин, понеже по време на изпълнение може да се викат методи и да се определят стойностите. Трябва да се помни едно нещо, обаче: не се предотвратява автоматичната инициализация, която става преди викането на метода. Например ако напишем:

```
class Counter {  
    int i;  
    Counter() { i = 7; }  
    // ...
```

`i` отначало ще стане нула, а после 7. Това се отнася за всички примитивни типове и манипулаторите на обекти, включително онези, които получават явно стойности при декларацията им. По тази причина компилаторът не се опитва да ви заставя да инициализирате елементи никъде в конструктора или преди да се използват – те са вече инициализирани.⁵

Ред на инициализация

Вътре в клас редът се определя от реда на деклариране на променливите. Ако и променливите да са размесени с дефиниции на методи, всички променливи се инициализират преди да се извика кой да е метод – даже конструктора. Например:

```
//: c04:OrderOfInitialization.java  
// Demonstrates initialization order.  
  
// When the constructor is called, to create a  
// Tag object, you'll see a message:  
class Tag {  
    Tag(int marker) {  
        System.out.println("Tag(" + marker + ")");  
    }  
}
```

⁵ In contrast, C++ has the *constructor initializer list* that causes initialization to occur before entering the constructor body, and is enforced for objects. See *Thinking in C++*.

```

class Card {
    Tag t1 = new Tag(1); // Before constructor
    Card() {
        // Indicate we're in the constructor:
        System.out.println("Card()");
        t3 = new Tag(33); // Re-initialize t3
    }
    Tag t2 = new Tag(2); // After constructor
    void f() {
        System.out.println("f()");
    }
    Tag t3 = new Tag(3); // At end
}

public class OrderOfInitialization {
    public static void main(String[] args) {
        Card t = new Card();
        t.f(); // Shows that construction is done
    }
} //:~

```

В **Card** декларациите на **Tag** нарочно са разхърдяни за да се види, че инициализацията протича най-напред. В добавка **t3** се преинициализира в конструктора. Изходът е:

```

Tag(1)
Tag(2)
Tag(3)
Card()
Tag(33)
f()

```

Така **t3** манипуляторът бива инициализиран два пъти, един преди и един по време на извикването на конструктора. (Първият обект се бракува, така че може да бъде почищен покъсно.) На пръв поглед това може да изглежда неефективно, но то гарантира правилна инициализация – какво би станало ако бил определен претоварен конструктор който *не* инициализира **t3** и нямаше “дефолтна” инициализация за **t3** в декларацията му?

ИНИЦИАЛИЗАЦИЯ НА СТАТИЧНИ ДАННИ

Когато данните са **static** е същото; ако е примитив и не го инициализирате получава стойност по подразбиране. Ако е манипулятор остава инициализирано с нула докато не създадете обект и свържете манипулятора с него.

Ако искате да сложите инициализацията в точката на декларацията, прави се също като за не-статични. Тъй като обаче има само едно място в паметта за **static** независимо от броя на създадените обекти възниква въпросът кога става инициализацията. Примерът прави въпроса по-ясен:

```

//: c04:StaticInitialization.java
// Specifying initial values in a
// class definition.

class Bowl {
    Bowl(int marker) {
        System.out.println("Bowl(" + marker + ")");
    }
    void f(int marker) {
        System.out.println("f(" + marker + ")");
    }
}

```

```

        }

class Table {
    static Bowl b1 = new Bowl(1);
    Table() {
        System.out.println("Table()");
        b2.f(1);
    }
    void f2(int marker) {
        System.out.println("f2(" + marker + ")");
    }
    static Bowl b2 = new Bowl(2);
}

class Cupboard {
    Bowl b3 = new Bowl(3);
    static Bowl b4 = new Bowl(4);
    Cupboard() {
        System.out.println("Cupboard()");
        b4.f(2);
    }
    void f3(int marker) {
        System.out.println("f3(" + marker + ")");
    }
    static Bowl b5 = new Bowl(5);
}

public class StaticInitialization {
    public static void main(String[] args) {
        System.out.println(
            "Creating new Cupboard() in main");
        new Cupboard();
        System.out.println(
            "Creating new Cupboard() in main");
        new Cupboard();
        t2.f2(1);
        t3.f3(1);
    }
    static Table t2 = new Table();
    static Cupboard t3 = new Cupboard();
} //:~

```

Bowl подволява да се види създаването на клас и **Table** и **Cupboard** създават **static** членове на **Bowl** разпръснати в техните декларации на класовете. Забележете че **Cupboard** създава не-**static Bowl b3** преди **static** декларациите. Изходът показва какво се е случило:

```

Bowl(1)
Bowl(2)
Table()
f(1)
Bowl(4)
Bowl(5)
Bowl(3)
Cupboard()
f(2)
Creating new Cupboard() in main

```

```
Bowl(3)
Cupboard()
f(2)
Creating new Cupboard() in main
Bowl(3)
Cupboard()
f(2)
f2(1)
f3(1)
```

Инициализацията на **static** се прави само ако е необходимо. Ако не създадете **Table** и никога не се отнесете към **Table.b1** или **Table.b2**, **static Bowl b1** и **b2** никога няма да се създадат. Обаче те се създават само когато първия **Table** обект се създава (или първия достъп до **static** се върши). След това **static** обект не се реинициализира.

Редът на инициализацията е вички **static** първо, ако не са били инициализирани по време на предишното създаване на обект и тогава **не-static** обектите. Може да видите доказателство за това в изхода.

От помощ ще е да резюмираме процеса на създаване на обект. Нека класът се казва **Dog**:

1. Първият път когато се създава обект от тип **Dog** или когато за пръв път до **static** метод или **static** поле от клас **Dog** се извършва достъп Java интерпретаторът трябва да намери **Dog.class**, което той прави чрез търсене по "пътя за класовете".
2. Като е натоварен **Dog.class** (което създава **Class** обект, както ще научите по-нататък), всички негови **static** инициализатори се пускат да работят. Така **static** инициализацията става само веднъж, когато **Class** обектът се натоварва за пръв път.
3. Когато се прави **new Dog()**, процесът на конструиране на **Dog** обекта първо алокира достатъчно памет от хийпа за **Dog** обект.
4. Тази памет се нулира, автоматично задавайки стойностите по подразбиране на всички примитиви в **Dog** (нула за числата и еквивалент за **boolean** и **char**).
5. Всички инициализации които са при декларациите се изпълняват.
6. Конструкторите се изпълняват. Както ще видите в глава 6, това би могло да предизвика значителна активност, особено когато има наследяване.

Явна инициализация на static

Java позволява да се групират друг вид инициализации на **static** в специална "static конструкционна клауза" (понякога наричана *static block*) в класа. Това изглежда така:

```
class Spoon {
    static int i;
    static {
        i = 47;
    }
    // ...
```

Изглежда като метод, но е точно ключовата дума **static** следвана от тялото на метода. Този код, както другата **static** инициализация, се изпълнява само веднъж, първия път, когато се създава обект от този клас или се прави достъп до **static** член от него клас (даже и да не е създаден обект от този клас). Например:

```
//: c04:ExplicitStatic.java
// Explicit static initialization
```

```
// with the "static" clause.

class Cup {
    Cup(int marker) {
        System.out.println("Cup(" + marker + ")");
    }
    void f(int marker) {
        System.out.println("f(" + marker + ")");
    }
}

class Cups {
    static Cup c1;
    static Cup c2;
    static {
        c1 = new Cup(1);
        c2 = new Cup(2);
    }
    Cups() {
        System.out.println("Cups()");
    }
}

public class ExplicitStatic {
    public static void main(String[] args) {
        System.out.println("Inside main()");
        Cups.c1.f(99); // (1)
    }
    static Cups x = new Cups(); // (2)
    static Cups y = new Cups(); // (2)
} //:~
```

static инициализаторите за **Cups** ще се пуснат или като се прави достъп до **static** обекта **c1** на реда отбелязан с (1), или редът (1) се постави на коментар а коментарът на (2) се мањне. Ако и (1) и (2) се поставят на коментар, **static** инициализация за **Cups** не се прави въобще.

ИНИЦИАЛИЗАЦИЯ НА НЕСТАТИЧНИ ЕКЗЕМПЛЯРИ

Java 1.1 има подобен синтаксис за инициализация на нестатични променливи за всеки обект. Ето един пример:

```
//: c04:Mugs.java
// Java 1.1 "Instance Initialization"

class Mug {
    Mug(int marker) {
        System.out.println("Mug(" + marker + ")");
    }
    void f(int marker) {
        System.out.println("f(" + marker + ")");
    }
}

public class Mugs {
    Mug c1;
    Mug c2;
}
```

```

c1 = new Mug(1);
c2 = new Mug(2);
System.out.println("c1 & c2 initialized");
}
Mugs() {
    System.out.println("Mugs()");
}
public static void main(String[] args) {
    System.out.println("Inside main()");
    Mugs x = new Mugs();
}
} //:~

```

Може да видите че клаузата за инициализация на екземпляра:

```

{
    c1 = new Mug(1);
    c2 = new Mug(2);
    System.out.println("c1 & c2 initialized");
}

```

изглежда точно като клаузата за статичните, освен че я няма ключовата дума **static**. Този синтаксис е необходим за поддържане на инициализацията на *анонимни вътрешни класове* (виж глава 7).

ИНИЦИАЛИЗАЦИЯ НА МАСИВИ

Инициализирането на масиви в С често е съпроводено с грешки и е досадно. C++ използва *инициализация на агрегати* за много по-бързо.⁶ Java няма “агрегати” като C++, понеже всичко е обекти в Java. Има масиви и те се инициализират чрез “инициализация на масиви”.

Масивът е просто последователност от или примитиви или обекти, всички от един тип и опаковани заедно под едно име. Масивите се декларират и използват с оператор от квадратни скоби наречен индексиращ оператор (**1**). За да се дефинира масив просто слагате след типа празни квадратни скоби:

```
| int() a1;
```

Може да ги сложите след името със същия резултат:

```
| int a1();
```

Това отговаря на очакванията на С и C++ програмистите. Първия пример като чели има по-смислен синтаксис, понеже говори за “масив от **int**.” Този стил ще се използва в тази книга.

Компилаторът не дава да му казвате колко голям е масивът. Това ни връща пак към въпроса за “манипуляторите.” Всичко което до тук имате е манипулятор към масив и не е алокирана памет за масива. За да се вземе памет за масива трябва да напишете израз за даване на начални стойности. За масивите инициализацията може да се появи навсякъде в програмата, но също може да използвате специален синтаксис в точката, където се декларира масивът. Тази специална инициализация е множество начални стойности, разделени със запетай и оградени с големи скоби. Алокирането на памет (еквивалентът на използване на **new**) е обект на грижите на компилатора в този случай. Например:

```
| int() a1 = { 1, 2, 3, 4, 5 };
```

Защо въобще ще дефинираме манипулятор на масив без масива?

⁶ See *Thinking in C++* for a complete description of aggregate initialization.

```
| int() a2;
```

Ами възможно е да се присвояват един масив на друг в Java, така че да се напише:

```
| a2 = a1;
```

Фактически се копира манипулатор, както е демонстрирано тук:

```
//: c04:Array.java
// Arrays of primitives.

public class Arrays {
    public static void main(String[] args) {
        int[] a1 = { 1, 2, 3, 4, 5 };
        int[] a2;
        a2 = a1;
        for(int i = 0; i < a2.length; i++)
            a2[i]++;
        for(int i = 0; i < a1.length; i++)
            prt("a1(" + i + ") = " + a1[i]);
    }
    static void prt(String s) {
        System.out.println(s);
    }
} ///:~
```

Може да видите че **a1** му се дава начална стойност докато на **a2** не се дава; **a2** му се приравнява по-късно – в този случай към друг масив.

Има нещо ново тук: всички масиви имат вграден член (без значение дали са масиви от примитиви или обекти) който може да четете – но не да променяте – за да видите колко елемента има масивът. Този член е **length**. Тъй като масивите в Java, подобно на С и C++, започват от елемент нула, най-големият номер на елемент който може да стои като индекс е **length - 1**. Ако излезете от масива С и C++ тихично приемат това позволявайки ви да шарите из цялата памет, което е източник на много безславни бъгове. Java обаче ви предпазва от такива неща чрез грешка по време на изпълнение (изключение, тема на глава 9) ако пристъпите извън масива. Разбира се, проверките при всеки достъп до масива струват време и код и не може да се изключат, което значи че достъпът до масивите може да стане източник на неефективност за вашата програма, ако става в критичен момент. За сигурност в Internet и продуктивност на програмиста проектантите на Java са счели че това си струва цената.

Ако не знаете колко елемента ще има масивът докато пишете програмата? Просто използвате **new** за създаване на елементите в масива. Тук **new** работи въпреки че се създава масив от примитиви (**new** не би създал примитив извън масив):

```
//: c04:ArrayNew.java
// Creating arrays with new.
import java.util.*;

public class ArrayNew {
    static Random rand = new Random();
    static int pRand(int mod) {
        return Math.abs(rand.nextInt()) % mod + 1;
    }
    public static void main(String[] args) {
        int[] a;
        a = new int(pRand(20));
        prt("length of a = " + a.length);
        for(int i = 0; i < a.length; i++)
```

```

    prt("a(" + i + ") = " + a(i));
}
static void prt(String s) {
    System.out.println(s);
}
} //:~

```

Понеже дължината на масива е случайно избрана (с използване на **pRand()** метода срещан и преди), ясно е че наистина създаването на масива става по време на изпълнение. В добавка се вижда от извежданото от програмата, че елементите се инициализират с "празни" стойности. (За числата това е нула, за **char** е **null**, за **boolean** е **false**.)

Разбира се, масивът може да се декларира и инициализира на едно място:

```
int() a = new int(pRand(20));
```

Ако имате работа с масив от непримитивни обекти, винаги трябва да използвате **new**. Тук въпросът с манипуляторите се проявява отново понеже това, което фактически се създава е масив от манипулятори. Да вземем обхващащия/заместващия клас **Integer**, който е клас и не е прimitив:

```

//: c04:ArrayClassObj.java
// Creating an array of non-primitive objects.
import java.util.*;

public class ArrayClassObj {
    static Random rand = new Random();
    static int pRand(int mod) {
        return Math.abs(rand.nextInt()) % mod + 1;
    }
    public static void main(String() args) {
        Integer() a = new Integer(pRand(20));
        prt("length of a = " + a.length());
        for(int i = 0; i < a.length(); i++) {
            a(i) = new Integer(pRand(500));
            prt("a(" + i + ") = " + a(i));
        }
    }
    static void prt(String s) {
        System.out.println(s);
    }
} //:~

```

Тука даже след като е извикан **new** да създаде масива:

```
Integer() a = new Integer(pRand(20));
```

Това е само масив от манипулятори и докато не се инициализира самия манипулятор чрез създаване на нов **Integer** обект инициализацията не е завършена:

```
a(i) = new Integer(pRand(500));
```

Ако забравите да създадете обекта ще получите грешка по време на изпълнение, когато се опитате да четете празното място на масива (по-точно несъществуващото място на масива, понеже манипуляторът му ще остане инициализиран с "нула", което за манипулятори е "никъде" - бел.прев.).

Хвърлете поглед на формирането на **String** обект вътре в операторите за извеждане. Може да се види че манипуляторът на **Integer** обект автоматично се превръща в **String** представящ стойността в обекта.

Възможно е също да се инициализират масиви от обекти чрез затворен в големи скоби списък. Има две форми, първата от които е единствената позволена в Java 1.0. Втората (еквивалентна) е позволена от Java 1.1 нататък:

```
//: c04:ArrayInit.java
// Array initialization

public class ArrayInit {
    public static void main(String[] args) {
        Integer[] a = {
            new Integer(1),
            new Integer(2),
            new Integer(3),
        };

        // Java 1.1 only:
        Integer[] b = new Integer[] {
            new Integer(1),
            new Integer(2),
            new Integer(3),
        };
    }
} ///:~
```

Това е полезно понякога, но е по-ограничено понеже дължината на масива се определя по време на компилация. Завършващата запетая в списъка на инициализаторите е незадължителна. (Тази черта помага за поддръжката на дълги списъци.)

Втората форма, въведена в Java 1.1, дава удобен начин да се пишат и викат методи, които произвеждат същия ефект като *променливите аргументни списъци* (известни още като "varargs") в C. Това включва неопределен брой аргументи и с неизвестен тип. Тъй като всички класове сигурно са наследници на общ клас **Object**, може да създадете метод който приема масив от **Object** и да го викате така:

```
//: c04:VarArgs.java
// Using the Java 1.1 array syntax to create
// variable argument lists

class A { int i; }

public class VarArgs {
    static void f(Object[] x) {
        for(int i = 0; i < x.length; i++)
            System.out.println(x[i]);
    }
    public static void main(String[] args) {
        f(new Object[] {
            new Integer(47), new VarArgs(),
            new Float(3.14), new Double(11.11)});
        f(new Object[] {"one", "two", "three"});
        f(new Object[] {new A(), new A(), new A()});
    }
} ///:~
```

На този етап няма много какво да правите с тези неизвестни обекти и тази програма използва автоматична **String** конверсия за да направи нещо полезно с **Object**. В глава 11 (run-time type identification или RTTI) ще научите как да намирате точния тип по време на изпълнение, за да можете да правите по-интересни неща.

Многомерни масиви

Java позволява лесно да се създават многомерни масиви:

```
//: c04:MultiDimArray.java
// Creating multidimensional arrays.
import java.util.*;

public class MultiDimArray {
    static Random rand = new Random();
    static int pRand(int mod) {
        return Math.abs(rand.nextInt()) % mod + 1;
    }
    public static void main(String[] args) {
        int[][] a1 = {
            { 1, 2, 3 },
            { 4, 5, 6 },
        };
        for(int i = 0; i < a1.length; i++)
            for(int j = 0; j < a1(i).length; j++)
                prt("a1(" + i + ")(" + j +
                    ") = " + a1(i)(j));
        // 3-D array with fixed length:
        int[][][] a2 = new int[2][2][4];
        for(int i = 0; i < a2.length; i++)
            for(int j = 0; j < a2(i).length; j++)
                for(int k = 0; k < a2(i)(j).length;
                    k++)
                    prt("a2(" + i + ")(" +
                        j + ")" + k +
                        ") = " + a2(i)(j)(k));
        // 3-D array with varied-length vectors:
        int[][][] a3 = new int[pRand(7)][];
        for(int i = 0; i < a3.length; i++) {
            a3(i) = new int[pRand(5)];
            for(int j = 0; j < a3(i).length; j++)
                a3(i)(j) = new int[pRand(5)];
        }
        for(int i = 0; i < a3.length; i++)
            for(int j = 0; j < a3(i).length; j++)
                for(int k = 0; k < a3(i)(j).length;
                    k++)
                    prt("a3(" + i + ")(" +
                        j + ")" + k +
                        ") = " + a3(i)(j)(k));
        // Array of non-primitive objects:
        Integer[][] a4 = {
            { new Integer(1), new Integer(2) },
            { new Integer(3), new Integer(4) },
            { new Integer(5), new Integer(6) },
        };
        for(int i = 0; i < a4.length; i++)
            for(int j = 0; j < a4(i).length; j++)
                prt("a4(" + i + ")(" + j +
                    ") = " + a4(i)(j));
        Integer[] a5;
        a5 = new Integer[3];
    }
}
```

```

for(int i = 0; i < a5.length; i++) {
    a5(i) = new Integer(3);
    for(int j = 0; j < a5(i).length; j++)
        a5(i)(j) = new Integer(i*j);
}
for(int i = 0; i < a5.length; i++)
    for(int j = 0; j < a5(i).length; j++)
        prt("a5(" + i + ")" + j +
            ") = " + a5(i)(j));
}
static void prt(String s) {
    System.out.println(s);
}
} //:~

```

Кодът за извеждането използва **length** и не зависи от фиксирана дължина на масивите.

Първият пример показва многомерен масив от примитиви. Отделяме всеки вектор с големи скоби:

```

int() a1 = {
    { 1, 2, 3, },
    { 4, 5, 6, },
};

```

Всяко множество в големи скоби ви придвижква в следващото ниво на масива.

Следващият пример показва тримерен масив, алокиран с **new**. Тук челият масив се алокира наведнък:

```
| int() a2 = new int(2)(2)(4);
```

Но в третия пример се вижда че всеки вектор в масива, който съставя матрицата, може да бъде с произволна дължина:

```

int() a3 = new int(pRand(7))();
for(int i = 0; i < a3.length; i++) {
    a3(i) = new int(pRand(5))();
    for(int j = 0; j < a3(i).length; j++)
        a3(i)(j) = new int(pRand(5));
}

```

Първият **new** създава масив със първи елемент-случайна дължина и другите неопределени. Вторият **new** вътре във for цикъла запълва елементите но оставя третият индекс неопределен до третия **new**.

Ще видите от римера, че масивите се инициализират с нула, ако не ги инициализирате вие.

За масивите от не-примитиви е същото, както е показано в четвъртия пример, демонстриращ възможността да се съберат много **new** изрази с големи скоби:

```

Integer() a4 = {
    { new Integer(1), new Integer(2)},
    { new Integer(3), new Integer(4)},
    { new Integer(5), new Integer(6)},
};

```

Петият пример показва как масив от непримитиви може да бъде построен парче по парче:

```

Integer() a5;
a5 = new Integer(3)();

```

```
for(int i = 0; i < a5.length; i++) {  
    a5(i) = new Integer(3);  
    for(int j = 0; j < a5(i).length; j++)  
        a5(i)(j) = new Integer(i*j);  
}
```

`i*j` е само за да сложи интересна стойност в **Integer**.

Резюме

Изглеждащият сложен механизъм за инициализация, конструкторът, ви дава важно доказателство за важността на въпросите на инициализацията. Когато Stroustrup проектира C++ едно от първите наблюдения които направил за производителността на С било че неправилната инициализация води до значителна част от програмистките проблеми. Тези видове грешки са трудни за откриване и подобно е палажението с изчистването. Понеже конструкторите позволяват да се гарантира правилна инициализация и изчистване (компилаторът няма да позволи създаването на обект без правилни викания на конструктор), получавате пълен контрол и безопасност.

В C++ разрушаването е важно, понеже обектите създадени с **new** трябва да бъдат явно разрушавани. В Java боклучарят освобождава паметта на обектите, затова подобен механизъм не е много необходим. В случаите когато не се нуждаете от деструктороподобно поведение боклучарят на Java много опростява програмирането и добавя много необходима безопасност в управлението на паметта. Някои боклучари даже чистят други ресурси като графики и файлови манипулатори. Обаче боклучарят си има цена, която е трудно да се сложи в перспектива поради общата бавност на Java интерпретаторите към момента на написването на книгата. Щом това се промени ще можем да преценим дали цената на боклукосъбирането ще предотврати използването на Java за някои типове програми. (Един от проблемите е непредсказуемостта на боклучаря.)

Поради гаранцията че всички обекти ще се конструират има повече за конструкторите от показаното тук. В частност, когато конструирате нови класове използвайки **композиция** или **наследяване** гаранцията също важи и е нужен допълнителен синтаксис за да се опише това. Ще научите за композицията, наследяването и как те засягат конструкторите в следващите глави.

Упражнения

1. Създайте клас с конструктор по подразбиране (такъв без аргументи) който извежда съобщение. Създайте обект от този клас.
2. Създайте претоварен конструктор за упражнение 1 който взима **String** аргумент и го извежда освен вашето съобщение.
3. Създайте масив от манипулатори на обектите за класа от упражнение 2, но не създавайте обекти за запълване на масива. Когато пуснете програмата, вижте дали се извеждат съобщения за инициализация.
4. Завършете упражнение 3 със създаване на обекти за свързване с масива от манипулатори.
5. Експериментирайте с **Garbage.java** чрез използване на аргументи "before," "after" and "none." Повторете процеса и вижте дали има нещо характерно в

изхода. Променете кода така, че `System.runFinalization()` да се вика преди `System.gc()` и наблюдавайте резултата.

5: Скриване на реализацията

Основно съображение в ООП е “разделяне на нещата, които се променят от нещата, които остават същите.”

Това в частност е важно за библиотеките. Потребителят (клиент-програмистът) на библиотеката трябва да може да разчита на това, което използва и да знае, че няма да има нужда да пренаписва кода си, ако излезе нова версия на библиотеката. От другата страна, създателят на библиотеката трябва да има възможност да подобрява кода си, без това да засяга кона на клиент-програмистите.

Това може да се постигне чрез съглашение. Например програмистът на библиотеката трябва да се съгласи да не премахва методи, понеже това ще наруши работата на клиентската програма. Обратната ситуация е деликатна, обаче. В случая на член-променливи откъде да знае програмистът на класа кой от тях използва клиентската програма? Това също се отнася и за методите които са част от реализацията на класа и не са предназначени за директно използване от програмист. Ами ако създателят на библиотеката иска да премахне стара реализация и да направи нова? Пробяната на който и да е от членоете би могла да развали работата на приложната програма. Така създателят на библиотеката е в усмирителна риза и не може да променя нищо.

За решаването на този проблем Java има спецификатори на достъпа за да се позволи на създателят на библиотеката да каже кое е достъпно за клиент-програмиста и кое не е. Нивата на достъп от “най-голям достъп” до “най-малък достъп” са **public**, “приятелско” (за него няма ключова дума), **protected** и **private**. От предишния параграф може да сте останали с впечатление че, като проектант на библиотеката, ще искате да даржите всичко “private” доколкото е възможно и да показвате само нещата които искате клиент-програмистът да използва. Това е точно така, макар и обикновено да противоречи на интуицията на програмисти с други езици (especially C) които са свикнали да имат достъп до всичко без ограничение. В края на тази глава ще бъдете убедени, че е ценен контролът на достъпа в Java.

Концепцията за библиотека от компоненти и кой има достъп до тях не е завършена, обаче. Още остава въпросът как тези компоненти ще се свържат в единна библиотечна единица. Това се управлява с ключовата дума **package** в Java и спецификаторите на достъпа се засягат от това дали класът е в същата библиотека или в друга. Така че за начало на тази глава ще научите как компонентите се слагат в библиотека. След това ще може да разберете пълното значение на спецификаторите на достъпа.

package: библиотечната единица

Това, което се взима когато се използва ключовата дума **import** за цяла библиотека е пакетът, както в

```
| import java.util.*;
```

Това взима цялата библиотека ютилита която е част от дистрибутивния материал на Java. Тъй като **ArrayList** е в **java.util**, може или да посочите цялото име **java.util.ArrayList** (което може да направите без ключовата дума **import**), или просто да напишете **ArrayList** (с **import**).

Ако искате само един клас, може да го споменете с **import**

```
| import java.util.ArrayList;
```

Сега може да ползвате **ArrayList** без квалификация. Но никой друг клас от **java.util** не е достъпен.

Причината за цялото това импортиране е да може да се управляват "пространствата на имената." Имената на всички ваши членове са изолирани едно от друго. Метод **f()** вътре в клас **A** няма да се бърка с **f()** който има същата сигнатура (списък аргументи) в клас **B**. Ами за имената на класовете? Да допуснем че създавате **stack** клас който се инсталира на машина където има вече **stack** написан от някой друг? С Java в Мрежата това може да стане понеже потребителят не знае какви класове ще се свалят при употребата на Java програмата.

Поради този потенциален конфликт на имената е важно да се управляват те в Java и да може да се конструират уникални имена независимо от ограниченията на Мрежата.

До тук повечето от примерите се помещаваха в един файл и бяха проектирани за локална употреба и нямаше притеснения от имената на пакетите. (В този случай името на класа се слага в "пакета по подразбиране.") Това е по избор и този подход ще се използва на повечето места в книгата. Ако планирате да направите програма, която е "Internet friendly," обаче, трябва да мислите за предотвратяване на конфликт на имената.

Сорсът на Java обикновено се нарича **компиляционна единица** (понякога **транслационна единица**). Всяка компиляционна единица трябва да има име завършващо с **.java** и вътре в нея може да има публичен клас който трябва да има същото име (включително капитализацията, но без **.java** разширението на файловото име). Ако не направите така компилаторът ще се оплаква. Може да има един **public** във всяка компиляционна единица (иначе компилаторът пак ще се оплаква). Останалите класове в единицата, ако ги има, са скрити от света извън пакета понеже *не са public*, те включват "поддържащите" класове за главния **public** клас.

Когато се компилира **.java** файл се получава изходен файл с точно същото име но с разширение **.class** за **всеки клас** в **.java** файла. Така може да се получат няколко **.class** файла от всеки **.java** файл. Ако сте програмирали на компилируем език може да сте свикнали компилаторът да изплюва някаква междинна форма (обикновено "obj" файл) която после се опакова с други от същия род чрез линкера (за да се създаде изпълнимия файл) или библиотекаря (за библиотека). Това не е начинът на работа на Java. Изпълнимата програма е куп **.class** файлове, които може да бъдат компресирани и пакетирани в JAR файл (чрез **jar** ютилитата в Java 1.1). Java интерпретаторът е отговорен за намирането, свързването и товаренето на тези файлове.¹

Библиотеката е също куп такива файлове. Всеки има един клас **public** (не е задължително да има **public** клас, но е типично), така че има един компонент във всеки файл. Ако искате да кажете че всички тези файлове вървят заедно (които са в свои отделни **.java** и **.class** файлове) това е мястото, където ключовата дума **package** влиза в ролята си.

Когато напишем:

```
| package mypackage;
```

¹ There's nothing in Java that forces the use of an interpreter. There exist native-code Java compilers that generate a single executable file.

в началото на файл, където **package** операторът трябва да е първият некоментарен ред във файла, вие казвате, че тази компилационна единица е част от библиотека наречена **турсаке**. Или, с други думи, казвате че **public** клас името в единицата е под чадъра на **турсаке**, и ако някой иска да използва името или трябва да го специфицира напълно или да употреби ключовата дума **import** комбинирана с **турсаке** (използвайки избраните преди неща). Забележете че конвенцията за Java пакетите е да се използват навсякъде малки букви, даже и за междинните думи.

Например да предположим че името на класа е **MyClass.java**. Това значи че може да има един и само един **public** клас в този файл че името на класа може да бъде само **MyClass** (капитализацията е същата):

```
| package турсаке;
| public class MyClass {
|   // ...
```

Сега ако някой иска да използва **MyClass** или, по подобен начин, който и да е от **public** класовете в **турсаке**, той трябва да използва ключовата дума **import** за да направи името или имената в **турсаке** достъпни. Алтернативата е да се даде пълно име за всеки клас:

```
| турсаке.MyClass m = new турсаке.MyClass();
```

Ключовата дума **import** може да го направи много по-просто:

```
| import турсаке.*;
| // ...
| MyClass m = new MyClass();
```

Заслужава си да се помни, че ключовите думи **package** и **import** позволяват на проектанта на библиотеката да си отдели специално пространство на имената, без значение колко души са в Internet и пишат класове на Java.

Създаване на уникални имена на пакети

Бихте могли да забележите че, понеже пакетът никога не се реално “пакетира” в единствен файл, той може да се състои от много **.class** файлове и става малка бъркотия. За да се предотврати тя логично е да се кложат всички **.class** файлове на отделен пакет в отделна директория; тоест да се използва юрархичната структура на операционната система във ваша полза. Така Java се оправя с проблема с бъркотията.

Това също решава и други два проблема: създаване на уникални имена на пакети и изравяне на класове, които могат да са забутани някъде в директорната структура. Това се прави, както беше казано в глава 2, чрез кодирането на пътя до **.class** файла в името на **package**. Компилаторът изисква това, но по уговорка първата част от името на **package** е Internet домейновото име на създателя на класа, обърнато. Тъй като домейновите имена в Internet гарантирано са уникални (от InterNIC,² които контролират присвояването им) ако следвате това договаряне имената на вашите **package** винаги ще бъдат уникални и няма да има конфликт на имена. (Докато някой друг вземе домейновото име от вас и започне да пише Java код със същите директорни пътища като вие сте писали.) Разбира се, ако нямаете собствено домейново име трябва да измислите някаква уникална комбинация (като първото име и фамилията ви) за да се създават уникални имена на пакетите. Ако сте решили да публикувате Java код относително малкото усилие да вземете домейново име си струва.

Втората част от трика е разхвърлянето на **package** името в директория на вашата машина, та когато Java програма се пусне и трябва да се натовари **.class** файл (което става динамично, в момента когато трябва да се създаде обект от въпросния клас или има достъп до **static** член на класа), тя да може да намери къде се намира **.class** файла.

² [ftp://ftp.internic.net](http://ftp.internic.net)

Интерпретаторът на Java процедира по следния начин. Първо намира променливата CLASSPATH от работната среда (която е сложена от операционната система когато Java или инструмент като Java-способен броузър се е инсталирал на машината). CLASSPATH съдържа една или повече директории които служат за корен при търсенето на **.class** файлове. Започвайки от него корен интерпретаторът ще вземе името и на мястото на всяка точка ще сложи наклонена черта за да формира пътя от CLASSPATH корена (така **package foo.bar.baz** става **foo\bar\baz** или **foo/bar/baz** в зависимост от операционната система). Това после се съединява с различните пътища в CLASSPATH. Каквото се получава — там се гледа за **.class** файл с име отговарящо на класа който се търси. (Претърсват се също и някои стандартни директории в зависимост от разположението на интерпретатора).

За да стане разбираемо, нека вземем моето домейново име което е **bruceeckel.com**. Като го обрънем, **com.bruceeckel** установява уникално глобално име за моите класове. (com, edu, org и т.н. разширението се капитализираше по-рано в Java пакетите, но това се промени в Java 2 така че цялото име на пакета е с малки букви.) Мога да подразделя полученото понятие решавайки да създам библиотека с име **util**, така че ще завърша с име на пакет:

```
| package com.bruceeckel.util;
```

Сега това име на пакет може да бъде използвано като захлупващо за пространството на имената на следните два файла:

```
//: com:bruceeckel:util:Vector.java
// Creating a package
package com.bruceeckel.util;

public class Vector {
    public Vector() {
        System.out.println(
            "com.bruceeckel.util.Vector");
    }
} //:~
```

Когато създавате свои пакети ще откриете, че **package** операторът трябва да бъде първият некоментарен ред във файла. Вторият сурс изглежда доста подобно:

```
//: com:bruceeckel:util>List.java
// Creating a package
package com.bruceeckel.util;

public class List {
    public List() {
        System.out.println(
            "com.bruceeckel.util.List");
    }
} //:~
```

И двета файла се слагат в поддиректория на моята система:

```
| C:\DOC\JavaT\com\bruceeckel\util
```

Ако погледнете отзад напред ще видите **com.bruceeckel.util**, но как е работата с първата част на пътя? Тя е от CLASSPATH променливата която на моята машина е:

```
| CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT
```

Може да се види че CLASSPATH може да съдържа няколко алтернативни пътя. Обаче при използване на JAR файлове има вариация. Трябва да се сложи името на JAR файла в classpath, не само пътят на който е разположен. Така за JAR наречен **grape.jar** classpath би включвал:

```
| CLASSPATH=.;D:\JAVA\LIB;C:\flavors\grape.jar
```

Щом като веднък пътят до класовете е установен вярно следния файл може да се сложи където и да е: (Виж стр.63 ако има проблеми с изпълнението на тази програма.):

```
//: c05:LibTest.java
// Uses the library
package c05;
import com.bruceeeckel.util.*;
public class LibTest {
    public static void main(String[] args) {
        Vector v = new Vector();
        List l = new List();
    }
} //:~
```

Когато компилаторът намери **import** оператора той започва търсене от директориите споменати в CLASSPATH гледайки за поддиректория com\bruceeeckel\util, а после преглежда файловете за подходящи имена (**Vector.class** за **Vector** и **List.class** за **List**). Забележете че и двата класа **Vector** и **List** трябва да са **public**.

Автоматична компиляция

Първият път когато се създава обект от импортиран клас (или се прави достъп до **static** член на клас), компилаторът ще тръгне на лов за **.class** със същото име (така че като създавате обект от клас **X**, търси се **X.class**) в съответната директория. Ако намери само **X.class**, това трябва да използва. Ако обаче намери **X.java** в същата директория, компилаторът ще сравни времената на създаване на двета файла и ако **X.java** е по-късно създаден от **X.class** ще прекомпилира автоматично **X.java** за да генерира усъвременен **X.class**.

Ако класът не е в **.java** файл със същото име като класа това поведение не е в сила за този клас.

Колизии

Какво става когато две библиотеки са импортирани със * и те включват еднакви имена? Например да предположим че програмата прави това:

```
import com.bruceeeckel.util.*;
import java.util.*;
```

Тъй като **java.util.*** също съдържа клас **Vector**, имаме потенциална колизия. Обаче докато колизията не възникне наистина всичко е OK – това е хубаво, защото иначе ще трябва да се пишат много редове програма за да се предотвратят колизии които никога няма да възникнат.

Колизията се получава ако се опитате да направите **Vector**:

```
| Vector v = new Vector();
```

За кой клас **Vector** става дума? Комилаторът не може да знае, а също и читателят. Така че компилаторът се сърди и ви кара да бъдете експлицитни. Ако искам стандартния Java **Vector**, например, трябва да напиша:

```
| java.util.Vector v = new java.util.Vector();
```

Тъй като така (и с CLASSPATH) напълно се определя мястото на искания **Vector**, няма нужда от **import java.util.*** оператор освен ако не използвам нещо друго от **java.util**.

Библиотека собствени инструменти

С това знание можете да създавате собствени библиотеки с инструменти за да намалите или елиминирате дублицирането на код. Например да създадем псевдоним на **System.out.println()** за да намалим писането. Това може да бъде част от пакет наречен **tools**:

```
//: com:bruceeckel:tools:P.java
// The P.print & P.println shorthand
package com.bruceeckel.tools;

public class P {
    public static void print(Object obj) {
        System.out.print(obj);
    }
    public static void print(String s) {
        System.out.print(s);
    }
    public static void print(char[] s) {
        System.out.print(s);
    }
    public static void print(char c) {
        System.out.print(c);
    }
    public static void print(int i) {
        System.out.print(i);
    }
    public static void print(long l) {
        System.out.print(l);
    }
    public static void print(float f) {
        System.out.print(f);
    }
    public static void print(double d) {
        System.out.print(d);
    }
    public static void print(boolean b) {
        System.out.print(b);
    }
    public static void println() {
        System.out.println();
    }
    public static void println(Object obj) {
        System.out.println(obj);
    }
    public static void println(String s) {
        System.out.println(s);
    }
    public static void println(char[] s) {
        System.out.println(s);
    }
    public static void println(char c) {
        System.out.println(c);
    }
    public static void println(int i) {
        System.out.println(i);
    }
    public static void println(long l) {
```

```

    System.out.println();
}
public static void println(float f) {
    System.out.println(f);
}
public static void println(double d) {
    System.out.println(d);
}
public static void println(boolean b) {
    System.out.println(b);
}
} //:~

```

Всичките различни данниови типове могат сега да се извеждат с нов ред (`P.println()`) или без нов ред (`P.print()`).

Може да познаете, че този файл трябва да се намира на директория, която е на един от пътищата започващи в `CLASSPATH`, после продължава `com/bruceeckel/tools`. След компилирането `P.class` файла може да бъде използван къде да е във вашата система чрез използването на `import`:

```

//: c05:ToolTest.java
// Uses the tools library
import com.bruceeckel.tools.*;

public class ToolTest {
    public static void main(String[] args) {
        P.println("Available from now on!");
    }
} //:~

```

Така от сега нататък щом се сдобиете с хубаво ютилити може да го сложите в `tools` директорията. (Или във вашата собствена `util` или `tools` директория.)

Капанът с classpath

Файлът `P.java` изважда наяве интересен капан. Особено в ранните реализации на Java слагането на коректен classpath изобщо си е главоболие. През време на разработката на тази книга файлът `P.java` беше създаден и тестван и си работеше чудесно, но по едно време работата се развали. Дълго време си мислех че това се дължи на грешка в реализациите на Java или някаква друга грешка, но накрая открих че в едната точка където бях въвел програмата (`CodePackager.java`, показан в глава 17) че се използва различен клас `P`. Понеже беше използван като инструмент, той понякога биваше слаган в classpath. Когато фигурираше там, `P` в `CodePackager.java` беше намиран първо от Java когато се използваше програма и се гледаше в `com.bruceeckel.tools` и компилаторът казваше, че конкретният метод липсва. Това беше разочароваващо, понеже в горния файл може да се види клас `P` и никаква допълителна информация не се извеждаше, която да подсвети че се намира съвсем друг клас. (Който даже не беше `public`.)

На пръв поглед това изглежда като бъг на компилатора, но ако се погледне `import` той казва само "ето тук може да се намери `P`." Обаче компилаторът ще гледа навсякъде по classpath, така че `P` се използва от там, от където се намери, а в случая се намира "неправилен" клас първо по време на търсенето и то се прекратява. Това е малко по-различно от написаното понеже и двата класа са си в пакетите и ето `P` което не беше в пакета, но може да бъде намерено при търсенето по classpath.

Ако имате подобни преживявания, осигурете наличието само на един клас с едно име по classpath.

Използване на импортиране за промяна на поведението

Едно нещо което липсва в Java е компиляцията при условия на C, която позволява да промените ключ и да получите различно поведение без да се променя нищо друго. Причината това да се изостави в Java е вероятно че то най-често се използва в C за да се решат междуплатформени въпроси: различни части на кода се компилират според платформата за която е предназначена компилацията. Тъй като Java е замислен да бъде автоматично междуплатформен такава функция не трябва да е необходима.

Има обаче и други различни нужди от условна компилация. Често срещано е компилирането на код за улесняване на тестването. Чертите за дебъгинга се включват по време на разработката и се изключват за търговската версия. Allen Holub (www.holub.com) излезе с идеята да използва пакетите за подражаване на условната компилация. Той създаде Java версия на много полезния *assertion* механизъм на C, чрез който може да се каже "това ще е истина" или "това ще е лъжка" и ако операторът не е съгласен с вас ще си проличи. Такъв инструмент е много полезен при тестването.

Ето класа който да се използва при тестване:

```
//: com:bruceeckel:tools:debug:Assert.java
// Assertion tool for debugging
package com.bruceeckel.tools.debug;

public class Assert {
    private static void perr(String msg) {
        System.err.println(msg);
    }
    public final static void is_true(boolean exp) {
        if(!exp) perr("Assertion failed");
    }
    public final static void is_false(boolean exp){
        if(exp) perr("Assertion failed");
    }
    public final static void
    is_true(boolean exp, String msg) {
        if(!exp) perr("Assertion failed: " + msg);
    }
    public final static void
    is_false(boolean exp, String msg) {
        if(exp) perr("Assertion failed: " + msg);
    }
} ///:~
```

Този клас просто капсулира булеви тестове, които извеждат съобщение ако са отрицателни. В глава 9 ще научите за по-напреднал метод за работа при грешки наречен *exception handling*, а **perr()** методът и до там ще работи перфектно.

Когато искате да използвате този клас добавяте реда:

```
import com.bruceeckel.tools.debug.*;
```

За да махнете твърденията и да може да разпространите кода, втори **Assert** се създава, но в различен пакет:

```
//: com:bruceeckel:tools:Assert.java
// Turning off the assertion output
// so you can ship the program.
```

```
package com.bruceeckel.tools;

public class Assert {
    public final static void is_true(boolean exp){}
    public final static void is_false(boolean exp){}
    public final static void
    is_true(boolean exp, String msg) {}
    public final static void
    is_false(boolean exp, String msg) {}
} //:~
```

Сега ако промените предишния **import** оператор на:

```
import com.bruceeckel.tools.*;
```

програмата няма повече да извежда твърденията. Ето пример:

```
//: c05:TestAssert.java
// Demonstrating the assertion tool
package c05;
// Comment the following, and uncomment the
// subsequent line to change assertion behavior:
import com.bruceeckel.tools.debug.*;
// import com.bruceeckel.tools.*;

public class TestAssert {
    public static void main(String[] args) {
        Assert.is_true((2 + 2) == 5);
        Assert.is_false((1 + 1) == 2);
        Assert.is_true((2 + 2) == 5, "2 + 2 == 5");
        Assert.is_false((1 + 1) == 2, "1 + 1 != 2");
    }
} //:~
```

Чрез промяна на импортирания пакет **package** може да смените кода от дебъг версията към продукционната версия. Тази техника може да се използва за всеки вид условен код.

Относно пакетите

Заслужава си да се помни, че щом се създаде пакет неявно се задава път. Пакетът трябва да е наличен в директорията зададена от името му, като тя трябва да може да се намери като се започне от CLASSPATH. Експериментирането с ключовата дума **package** може да бъде малко разочароващо отначало, понеже докато не спазите условието за зависимостта между името на файла и пътя ще получавате много мистериозни съобщения че даден клас не може да се намери, даже и класът да си е там в директорията. Ако получите подобно съобщение опитайте да изкоментирате **package** и ако тръгне ще знаете къде е проблемът.

Спецификатори на достъпа в Java

Спецификаторите на достъпа в Java **public**, **protected** и **private** се слагат пред всяка дефиниция на всеки член в класовете, бил той член-данни или метод. Всеки спецификатор управлява достъпа до само тази конкретна дефиниция. Това е значителен контраст спрямо C++, където спецификаторът на достъпа управлява всички следващи дефиниции докато не се появии нов спецификатор на достъпа.

По един или друг начин за всяко нещо е определен вид достъп. В следващите секции ще научите всичко за достъпа, започвайки с този по подразбиране.

“Приятелски”

Какво става ако въобще не дадете спецификатор на достъпа както във всичките примери досега от тази глава? Достъпът по подразбиране няма ключова дума но често се споменава като “приятелски.” Това значи че всички други класове от пакета имат достъп до приятелския член, но всички класове извън пакета нямат достъп. Тъй като компилационната единица – файл – може да принадлежи само на един пакет, всичките класове в компилационна единица са автоматично приятелски един на друг. Така приятелските елементи още казваме че имат пакетен достъп.

Приятелският достъп позволява да се групират класовете в пакет така че да имат възможност лесно да си взаимодействват. Като сложите класове заедно в пакет (давайки с това изключителен достъп до техните приятелски членове; т.е. правейки ги “приятели”) вие “владеете” кода в пакета. Има смисъл само кода който владеете да има достъп до код който вие владеете. Би могло да се каже, че приятелският достъп дава обяснение или причина за групиране на класовете във файлове. В много езици начинът на организиране на файловете е щеш-нешеш, но в Java сте задължени да го направите по смислен признак. В добавка вие вероятно бихте предпочели да изключите (от файла - бел.пр.) класа който не трябва да има достъп до класовете във файла.

Важен въпрос във всяка зависимост е “Кой има достъп до моята **private** реализация?” Класът контролира кой код има достъп до неговите членове. Няма магически начин да се “пробие;” някой в друг клас не може да дефинира нов клас и да каже, “Ей, аз съм приятелски на класа на **Bob!**” и да очаква да види **protected**, приятелски, и **private** членове на **Bob**. Единствения начин да се даде достъп до член е да:

1. Се направи члена **public**. Тогава всеки и навсякъде има достъп до него.
2. Се направи члена приятелски, като се остави без спецификатор на достъпа и да се сложат другите класове в същия пакет. Тогава другите класове имат достъп до члена.
3. Както ще видите в по-късна глава където се въвежда наследяването, наследникът има достъп до **protected** член а също и до **public** член (но не **private** членове). Той има достъп до приятелските членове само ако давата класа са в един пакет. Но не се занимавайте с това сега.
4. Се даде “accessor/mutator” методи (известни още като “get/set” методи) които четат/променят стойноста. Това е най-цивилизованият подход в термините на ООП и той е основен за Java Beans, както ще видите в глава 13.

public: интерфейсен достъп

Като се използва ключовата дума **public** това значи, че членът който непосредствено я следва е достъпен за всеки, в частност за клиент-програмиста, който използва библиотеката. Нека дефинираме пакет **dessert** съдържащ следната компилационна единица: (Виж страница 63 ако има проблеми с пускането на програмата.)

```
//: c05:dessert:Cookie.java
// Creates a library
package c05.dessert;

public class Cookie {
```

```
public Cookie() {  
    System.out.println("Cookie constructor");  
}  
void foo() { System.out.println("foo"); }  
} ///:~
```

Помните, **Cookie.java** трябва да е в поддиректория с име **dessert**, в директория под **C05** (индициращо глава 5 на тази книга) която трябва да е под една от CLASSPATH директориите. Не правете грешката да мислите, че Java винаги ще гледа на текущата директория като на една от началните точки на търсенето. Ако няма '.' като един от пътищата във външния CLASSPATH Java няма да търси там.

Ако сега създадем програма която използва **Cookie**:

```
//: c05:Dinner.java  
// Uses the library  
import c05.dessert.*;  
  
public class Dinner {  
    public Dinner() {  
        System.out.println("Dinner constructor");  
    }  
    public static void main(String[] args) {  
        Cookie x = new Cookie();  
        //! x.foo(); // Can't access  
    }  
} ///:~
```

Може да създадете **Cookie** обект, тъй като конструкторът му е **public** и класът е **public**. (Покъсно ще разгледаме публичните класове още.) Обаче **foo()** членът е недостъпен вътре в **Dinner.java** понеже **foo()** е приятелски само в пакета **dessert**.

Пакетът по подразбиране

Може да сте изненадани че следният код се компилира, макар че сякаш нарушава правилата:

```
//: c05:Cake.java  
// Accesses a class in a separate  
// compilation unit.  
  
class Cake {  
    public static void main(String[] args) {  
        Pie x = new Pie();  
        x.f();  
    }  
} ///:~
```

Във втория файл, в същата директория:

```
//: c05:Pie.java  
// The other class  
  
class Pie {  
    void f() { System.out.println("Pie.f()"); }  
} ///:~
```

На пръв поглед може да ви се сторят напълно чужди файлове и все пак **Cake** може да създаде **Pie** обект и да извика неговия **f()** метод! Типично се очаква че **Pie** и **f()** са приятелски

и затова недостъпни за **Cake**. Те са приятелски – тази част е точна. Причината да са достъпни в **Cake.java** че са в една директория и нямат явно пакетно име. Java третира файлове като тези неявно като от “пакет по подразбиране” за няя директория, следователно като приятелски на всички файлове в няя директория.

private: не може да пипате това!

Ключовата дума **private** означава, че никой няма право на достъп освен конкретния клас, вътре в методите на класа. Други класове в пакета нямат достъп до **private** членовете, като чели класът е изолиран и от самия себе си. От друга страна, не е невероятно класът да е създаден от няколко души работещи заедно, така че **private** позволява членът да се променя без да се засягат други класове в пакета. Подразбиращия се “приятелски” достъп в пакета е често достатъчна степен на скриване; запомните, “приятелски” член е недостъпен за потребител на пакета. Това е добре, той като нормално се използва достъпът по подразбиране. Така типично членовете, които ще са достъпни за клиента-програмист ще са **public** и като резултат първоначално може да считате че няма много да използвате **private** тъй като сносно може да се разминете без тази ключова дума. (Това е значителен контраст със C++.) Оказва се обаче че смисленото използване на **private** е много важно, специално когато се прави многонишковост. (Както ще видите в глава 14.)

Ето пример за използване на **private**:

```
//: c05:IceCream.java
// Demonstrates "private" keyword

class Sundae {
    private Sundae() {}
    static Sundae makeASundae() {
        return new Sundae();
    }
}

public class IceCream {
    public static void main(String[] args) {
        //! Sundae x = new Sundae();
        Sundae x = Sundae.makeASundae();
    }
} ///:~
```

Това е пример където **private** се оказва удобно: бихте могли да искате да управлявате създаването на обект и да не допуснете някой да използва даден конструктор (или всичките). В примера по-горе не може да се създаде **Sundae** обект чрез неговия конструктор; трябва да се извика вместо това **makeASundae()** за да го създаде.³

Всеки метод за който сте сигурни че е спомагателен в класа може да бъде направен **private** за да се осигури неупотребата му на друго място в класа и по този начин да се остави възможност за замяната му. Правенето на метода **private** това. (Обаче това че манипуляторът е **private** не значи, че други обекти не могат да имат **public** манипулятор към същия обект. Виж глава 12 за въпросите на алиасинга (псевдонимите).)

³ There's another effect in this case: [Since the default constructor is the only one defined, and it's private, it will prevent inheritance of this class.](#) ([A subject that will be introduced in Chapter 6.](#))

protected: “вид приятелски”

Спецификаторът на достъп **protected** изисква скок напред за да бъде разбран. Първо, ще бъдете предупредени, че разбирането на въпроса не е наложително докато не разберете наследяването. Но за пълнота тук има кратко описание и примери с **protected**.

Ключовата дума **protected** се преплита с концепцията за *наследяване*, която взема съществуващ клас и добавя членове без да се пипа съществуващия клас, който ще наричаме *базов клас*. Може също да се промени поведението на съществуващ член на клас. За да наследите от съществуващ клас казвате че новият **extends** (разширява - бел.пр.) съществуващ клас, както тук:

```
| class Foo extends Bar {
```

Останалата част от дефиницията на класа изглежда по същия начин.

Ако създавате нов пакет и наследявате от друг пакет, единствените членове до които имате достъп са **public** членовете на оригиналния пакет. (Разбира се, ако наследявате в *същия* пакет ще имате нормалния пакетен достъп до всички “приятелски” членове.) Понякога създателят на клас иска да вземе отделни членове и да даде достъп до тях само на наследниците на класа, а не на целия свят. Това е, което прави **protected**. Ако погледнете пак **Cookie.java** на стр. 140, следващият клас няма достъп до “приятелския” член:

```
//: c05:ChocolateChip.java
// Can't access friendly member
// in another class
import c05.dessert.*;

public class ChocolateChip extends Cookie {
    public ChocolateChip() {
        System.out.println(
            "ChocolateChip constructor");
    }
    public static void main(String[] args) {
        ChocolateChip x = new ChocolateChip();
        //! x.foo(); // Can't access foo
    }
} ///:~
```

Едно от интересните неща за наследяването е че ако методът **foo()** съществува в клас **Cookie**, той също съществува и във всеки клас, наследен от **Cookie**. Но доколкото **foo()** е “приятелски” във външен пакет, той е недостъпен за нас в този. Разбира се, бихте могли да го направите **public**, но тогава всеки би имал достъп и това може да не е което искате. Ако променим класа **Cookie** така:

```
public class Cookie {
    public Cookie() {
        System.out.println("Cookie constructor");
    }
    protected void foo() {
        System.out.println("foo");
    }
}
```

тогава **foo()** още има “приятелски” достъп в пакета **dessert**, но също е достъпен за всеки, който наследява от **Cookie**. Обаче не е **public**.

Интерфейс и реализация

Управлението на достъпа често се свързва със скриване на реализациите. Вграждането на данни и методи в класовете (комбинирано със скриване на реализациите често наричано капсулиране) дава даннов тип с характеристики и поведение, но контролът на достъпа слага граници в този даннов тип по две важни причини. Първата е да се зададе какво клиентът програмист може и какво не може да използва. Не може да изградите вътрешните механизми в структура и да очаквате клиент-програмиста да не я смята за част от интерфейса.

Това подхранва направо следващата причина, която е: да се раздели интерфейса от реализациите. Ако структурата се използва в няколко програми, като потребителите могат само да изпращат съобщения до **public** интерфейса, тогава може да се променя всичко което **не е public** (т.е. "приятелско," **protected** или **private**) без да са необходими промени в техния код.

Сега сме в света на ООП, където **class** фактически описва "клас от обекти," както бихте описали класа на рибите и класа на птиците (в биологията - бел.пр.). Всеки обект от класа ще споделя единни характеристики и поведение. Класът е описание на начина на действие и поведението на обектите от този клас.

В първия ООП език, Simula-67, ключовата дума **class** беше използвана за описание на нов даннов тип. Същата дума се използва в повечето ООП езици. Фокусната точка на целия език е: създаване на нови даннови типове които са повече от просто кутии за данни и методи.

Класът е фундаментална ОО концепция в Java. Това е една от ключовите думи която няма да бъде с удебелени букви в тази книга – дразнещо става, ако се подчертава толкова често срещене дума като "class."

За яснота може да предпочетете маниер на създаване на класове при който **public** членовете са в началото, следвани от **protected**, friendly и **private** членовете. Хубавото е, че потребителят като започне да чете отгоре надолу ще срещне това, което му трябва първо (**public** членовете, понеже те могат да се викат извън файла) и ще спре да чете, когато види непублични членове, които са част от вътрешната реализация. Обаче с коментарите за документация поддържани от javadoc (описан в глава 2) въпросът за четимостта на програмите става по-малко важен.

```
public class X {  
    public void pub1() { /*...*/ }  
    public void pub2() { /*...*/ }  
    public void pub3() { /*...*/ }  
    private void priv1() { /*...*/ }  
    private void priv2() { /*...*/ }  
    private void priv3() { /*...*/ }  
    private int i;  
    // ...  
}
```

Това може да помогне на членето само частично, защото все още интерфейсът и реализациите са смесени. Тоест все още се вижда сорсът – реализацията – понеже си е в класа. Извеждането на интерфейса за потребителя на класа си е работа на *class browser*-а, инструмент, чиято работа е да гледа класовете и да ви показва какво може да правите с тях (т.е. какви членове имат) по полезен начин. Когато чуете тази книга, добри обектни броузъри трябва да се обакват във всички добри Java среди за развой.

ДОСТЪП ДО КЛАС

В Java спецификаторите на достъпа може да се използват и за да се посочи кои класове в библиотека ще бъдат достъпни за потребителите на същата библиотека. Ако искате клас да бъде достъпен за клиент-програмиста слагате ключовата дума **public** някъде преди отварящата фигурна скоба на тялото на класа. Това управлява даже дали клиентът може да създаде обект от класа.

За да се управлява достъпът до клас, спецификаторът трябва да се появи преди **class**. Така може да се напише:

```
| public class Widget {
```

Тоест името на библиотеката е **mylib** и всеки клиент може да има достъп до **Widget** чрез

```
| import mylib.Widget;
```

или

```
| import mylib.*;
```

Има обаче двойка допълнителни ограничения:

1. Може да има само един **public** за компилационна единица (файл). Идеята е че всяка компилационна единица има единствен интерфейс представян от този клас. Може да има колкото трябват "приятелски" спомагателни класове. Ако имате повече от един **public** в компилационната единица компилаторът ще издае съобщение за грешка.
2. Името на **public** класа трябва точно да съвпада с името на компилационната единица, която го съдържа, включително капитализацията. Така за **Widget** името на файла трябва да е **Widget.java**, не **widget.java** или **WIDGET.java**. Ще излезе грешка при компилация ако те не съвпадат.
3. Възможно е, макар и да не е типично, да има компилационна единица без никакъв публичен клас. В този случай може да я наречете както желаете.

Какво ако сте вкарали клас в **mylib** който използвате за изпълнение на работите на **Widget** или друг **public** клас в **mylib**? Не искате да правите документация за клиент-програмист и смятате след време да промените нещата и да махнете изцяло този клас, замествайки го с друг. За да имате тази гъвкавост трябва да осигурите че клиент-програмистът не зависи от конкретната реализация скрита в **mylib**. За да се направи това махате **public** ключовата дума от класа, с което той става приятелски. (Такъв клас може да се използва само в пакета.)

Забележете че клас не може да бъде **private** (което ще го направи недостъпен за всичко освен за него), или **protected**.⁴ Така че имате два избора за достъп до клас: "приятелски" или **public**. Ако не искате никой да има достъп до този клас, може да направите всички конструктори **private**, предотвратявайки всеки освен вас, от **static** член на класа, да може да направи обект.⁵ Ето пример:

```
//: c05:Lunch.java
// Demonstrates class access specifiers.
// Make a class effectively private
```

⁴ Actually, a Java 1.1 *inner class* can be private or protected, but that's a special case. These will be introduced in Chapter 7.

⁵ [You can also do it by](#) inheriting (Chapter 6) from that class.

```

// with private constructors:

class Soup {
    private Soup() {}
    // (1) Allow creation via static method:
    public static Soup makeSoup() {
        return new Soup();
    }
    // (2) Create a static object and
    // return a reference upon request.
    // (The "Singleton" pattern):
    private static Soup ps1 = new Soup();
    public static Soup access() {
        return ps1;
    }
    public void f() {}
}

class Sandwich { // Uses Lunch
    void f() { new Lunch(); }
}

// Only one public class allowed per file:
public class Lunch {
    void test() {
        // Can't do this! Private constructor:
        //! Soup priv1 = new Soup();
        Soup priv2 = Soup.makeSoup();
        Sandwich f1 = new Sandwich();
        Soup.access().f();
    }
} ///:~

```

До сега повечето от методите връщаха или **void** или първичен тип, така че дефиницията:

```

public static Soup access() {
    return ps1;
}

```

може да изглежда малко смущаваща на пръв поглед. Думата преди името на метода (**access**) казва какво той връща. До тук най-често това беше **void**, което значи че не връща нищо. Но също би могло да се върне манипулятор на обект, както е в случая. Този метод връща манипулятор на обект **Soup**.

Класът **Soup** показва как да се избегне възможността за непосредствено създаване на обекти като се направят всички конструктори **private**. Помнете че ако не направите явно поне един конструктор, конструктор по подразбиране (конструктор без аргументи) ще бъде създаден автоматично. Като напишете конструктор по подразбиране той няма да се създаде автоматично. Като го направите **private** никой не може да създаде обект от него клас. Но как може да се използва класа? Горният пример показва две възможности. Първо, **static** метод да се създаде, който създава **Soup** и връща манипулятор към него. Това може да бъде полезно ако изпълнявате допълнителни операции над **Soup** преди да го върнете, или искате да запазите броя на създаваните **Soup** обекти (може би за регистрация на популацията им).

Втората възможност използва т.н. *design pattern*, което ще се дискутира по-късно в тази книга. Този конкретно шаблон се нарича "singleton" понеже позволява да се създаде само един обект изобщо. Обектът от клас **Soup** се създава като **static private** член на **Soup**, така че има един и само един и не може да се доберете до него освен с **public** метода **access()**.

Както вече беше споменато, ако не сложите спецификатор за достъп той става по подразбиране "приятелски." Това значи че обект от този клас може да се създаде от всеки друг клас в пакета, но не и от клас извън него. (Помните, всичките файлове в една директория които нямат явни **package** декларации са неявно част от пакета по подразбиране за същата директория.) Обаче ако **static** член от този клас е **public**, клиент-програмистът има достъп до **static** члена въпреки че не може да създаде обект от този клас.

Резюме

Във всяка една взаимозависимост е важно да има граници, общопризнати от всички участници. Като създавате библиотека вие задавате зависимост с потребителя – клиент-програмиста – който е друг програмист, строящ приложение или по-голяма библиотека.

Без правила клиент-програмистите биха могли да правят всичко с членовете на класовете, въпреки че вие може би бихте предпочели друго. Всичко е изложено на показ пред света.

В тази глава видяхме как класовете изграждат библиотеки; първо начинът по който група класове се пакетира в библиотека и второ начинът по който се управлява достъпът до класовете.

Оценено е че програмен проект на С започва да се проваля някъде между 50K и 100K реда големина понеже С има едно общо "пространство на имената", така че започват колизии, причиняващи проблеми с управлението на проекта. В Java ключовата дума **package**, схемата за имената на пакетите и ключовата дума **import** позволяват пълен контрол на имената, така че въпросът за колизиите на имената лесно се заобикаля.

Има две причини да се управлява достъпът до членовете. Първата е да се държи клиентът далеч от нещата, които не пръбва да пипа; инструментите които са необходими за вътрешните механизми на данновия тип, но не са част от интерфейса използван от потребителя. Така че правенето на методи и полета **private** е в служба на потребителя, който с това знае кое му трябва и не се занимава с другото. Това опростява и разбирането на класа.

Втората и по-важна причина е да се даде възможност на проектантът на библиотеката да промени вътрешностите без да се притеснява как това ще се отрази на приложните програми. Може да сте построили класа по един начин отначало и после да сте открили, че реконструкция на кода ще даде много по-голяма скорост. Ако интерфейсът и реализацията са добре разделени и защитени, това може да се направи без да се кара потребителят да препиши своя код.

Спецификаторите на достъпа в Java дават ценни възможности за управление на създателя на класа. Потребителите на класа виждат точно какво да използват и какво могат да игнорират. По-важното, по-нататък, е че може да се осигури независимост на потребителите от реализацията на класа. Ако знаете това като създател на класове, може да променяте подлежащата реализация понеже е сигурно, че това няма да засегне потребителите — те нямат достъп до тази част въобще.

Като имате възможността да променяте реализацията в бъдеще не само може да си подобрявате кода, но също имате свободата да правите грешки. Без значение колко грижливо се планира и изпълнява винаги се правят грешки. Като знаете това, ще експериментирате повече, ще направите по-бързо по-хубав код и по-бързо ще си завършите проекта.

Public интерфейса на клас се вижда от потребителя, така че тази част е най-важно да се направи както трябва по време на анализа и проектирането. Даже и това дава възможност за промени. Ако не стане интерфейсът изведенък, може да добавите методи, само няма да махате — тъй като може клиентът да ги е използвал.

Упражнения

1. Създайте клас с **public**, **private**, **protected**, и “приятелски” членове-данни и методи. Създайте обекти и вижте какви съобщения от компилатора ще получите като се опитате да използвате всевъзможните членове. Помнете че класовете в една директория са част от “пакет по подразбиране”.
2. Създайте клас с данни **protected**. Създайте друг клас в същия файл който манипулира **protected** данните в първия клас.
3. Създайте нова директория и редактирайте вашия CLASSPATH за да я включите. Копирайте **P.class** файла там и после сменете имената на файла, на **P** класа вътре и имената на методите. (Бихте могли да добавите оператори за извеждане за да проследите работата.) Създайте друга програма в друга директория която използва вашия клас.
4. Създайте следния файл в c05 директорията (предполага се че е на вашия CLASSPATH):

```
///: c05:PackagedClass.java
package C05;
class PackagedClass {
    public PackagedClass() {
        System.out.println(
            "Creating a packaged class");
    }
} ///:~
```

После създайте следния файл в различна от c05 директория:

```
///: c05:foreign:Foreign.java
package C05.foreign;
import C05.*;
public class Foreign {
    public static void main (String[] args) {
        PackagedClass pc = new PackagedClass();
    }
} ///:~
```

Обяснете защо компилаторът генерира грешка. Би ли правенето на **Foreign** класа част от **c05** пакета променило нещо?

6: Многократно използване на класове

Една от най-привлекателните черти на Java е многократното използване на кода. Но за да бъде революционно, трябва да можем много повече отколкото да копираме код и леко да го променяме.

Това последното е подходът в процедурните езици като С и то не работи много добре. Както всичко в Java, решението се върти около класовете. Пре-използвате кода чрез създаване на нови класове, но заместо да ги създавате изцяло наново, използвате съществуващи класове които някой вече е създал и тествал.

Трикът е да се използва съществуващ код без да се пипа. В тази глава ще видите два начина да се стори това. Първият е съвсем праволинеен: Просто създавате обекти от съществуващ клас вътре в даден клас. Това се нарича **композиция** понеже новият клас е композиран от старите. Просто използвате функционалността на кода, не формата му.

Вторият подход е по-изтънчен. Създава се нов клас от типа на съществуващ клас. Буквално взимате формата на съществуващ клас и добавяте код за разширяване на функционалността, без да се пипа съществуващия клас. Този магически акт се нарича **наследяване** и компилаторът върши повечето работа. Наследяването е крайъгълен камък на ООП и има и други употреби, които ще бъдат разгледани в следващата глава.

Оказва се, че повечето синтаксис и повезение са еднакви за композицията и наследяването (нищо чудно, понеже те са начини за добиване на нов тип от съществуващ). В тази глава ще научите за тези механизми за повторно използване на код.

Синтаксис на композицията

Композиция беше изпозвана доста често до този момент. Просто се слагат манипулятори на обекти в новия клас. Например нека искаме обект, който съдържа няколко **String** обекта, двойка примитиви и обект от друг клас. За непримитивните обекти само слагаме манипулятора в новия клас, а за примитивите просто ги декларираме в новия клас: (Виж стр. 63 ако има проблеми с пускането на тази програма.)

```
//: c06:SprinklerSystem.java
// Composition for code reuse
package c06;
```

```
class WaterSource {
```

```

WaterSource() {
    System.out.println("WaterSource()");
    s = new String("Constructed");
}
public String toString() { return s; }
}

public class SprinklerSystem {
    private String valve1, valve2, valve3, valve4;
    WaterSource source;
    int i;
    float f;
    void print() {
        System.out.println("valve1 = " + valve1);
        System.out.println("valve2 = " + valve2);
        System.out.println("valve3 = " + valve3);
        System.out.println("valve4 = " + valve4);
        System.out.println("i = " + i);
        System.out.println("f = " + f);
        System.out.println("source = " + source);
    }
    public static void main(String[] args) {
        SprinklerSystem x = new SprinklerSystem();
        x.print();
    }
} //:~

```

Един от методите определени в **WaterSource** е специален: **toString()**. Ще научите по-късно в тази глава, че всеки непримитивен обект има **toString()** метод и той се вика в специални ситуации когато компилаторът иска **String** но работи с един от тези обекти. Така в израза:

```
System.out.println("source = " + source);
```

Компилаторът вижда че се опитваме да добавим **String** обект ("source = ") към **WaterSource**. Това е безсмислено за него, понеже може само да се "събира" **String** с друг стринг **String**, така че казва "Ще превърна **source** в **String** чрез извикване на **toString()**!" След като свърши това той може да комбинира двета **Stringa** и да прати резултата **String** на **System.out.println()**. Всеки път когато искате да осигурите такова поведение за писан от вас клас трябва само да напишете **toString()** метод.

На пръв поглед би могло да се предположи че компилаторът автоматично ще конструира обекти за всеки от манипуляторите в горния код, например извиквайки конструктор по подразбиране за **WaterSource** за да инициализира **source**. Изходът е в действителност:

```

valve1 = null
valve2 = null
valve3 = null
valve4 = null
i = 0
f = 0.0
source = null

```

Примитивите които са полета в клас автоматично са инициализирани с нула, както отбеляхме в глава 2. Но манипуляторите на обекти се инициализират с **null** и ако се опитате да извикате метод на кийто и да е от тях ще получите изключение. Много добре е наистина (и полезно) че все пак може да ги изпечатате без изхвърляне на изключение.

Смисълът да не се създават обекти за всеки манипулятор веднага е че това често би било напразно. Ако искате да се инициализират манипуляторите, може да го направите:

1. В точката където се декларира обектът. Това значи че винаги ще бъдат инициализирани преди да се извика конструкторът на обекта.
2. В конструктора на съответния клас
3. Непосредствено преди употребата на съответния обект. Това може да намали ненужната работа в случаи, когато обектът може и да не се наложи да бъде създаден.

Всичките три подхода са показани тук:

```
//: c06:Bath.java
// Constructor initialization with composition

class Soap {
    private String s;
    Soap() {
        System.out.println("Soap()");
        s = new String("Constructed");
    }
    public String toString() { return s; }
}

public class Bath {
    private String
        // Initializing at point of definition:
        s1 = new String("Happy"),
        s2 = "Happy",
        s3, s4;
    Soap castille;
    int i;
    float toy;
    Bath() {
        System.out.println("Inside Bath()");
        s3 = new String("Joy");
        i = 47;
        toy = 3.14f;
        castille = new Soap();
    }
    void print() {
        // Delayed initialization:
        if(s4 == null)
            s4 = new String("Joy");
        System.out.println("s1 = " + s1);
        System.out.println("s2 = " + s2);
        System.out.println("s3 = " + s3);
        System.out.println("s4 = " + s4);
        System.out.println("i = " + i);
        System.out.println("toy = " + toy);
        System.out.println("castille = " + castille);
    }
    public static void main(String[] args) {
        Bath b = new Bath();
        b.print();
    }
}
```

```
| } //:/~
```

Забележете че в **Bath** се изпълнява оператор преди всички инициализации да са направени. Когато не инициализирате при декларацията, няма гаранция, че ще извършите инициализация преди да подадете съобщение към обектовия манипулятор – освен за неизбежното изключение по време на изпълнение.

Ето изходът от програмата:

```
Inside Bath()
Soap()
s1 = Happy
s2 = Happy
s3 = Joy
s4 = Joy
i = 47
toy = 3.14
castille = Constructed
```

Когато **print()** се извика той попълва **s4** така че всички полета са правилно инициализирани преди да са използвани.

Синтаксис за наследяването

Наследяването е толкова интегрална част от Java (и ООП езици изобщо), че беше въведена в глава 1 и биваше използвана на места преди тази глава в ситуации където е необходима. Освен това винаги се прави наследяване, когато се създава клас, понеже ако и да не се пише нищо специално се наследява стандартния коренен клас **Object** на Java.

Синтаксисът за композиция е очевиден, но за наследяване е различно. Когато се наследява, казваме "Този нов клас прилика на оня стар клас." Това се изрича в програмата чрез даване на име на новия клас както обикновено, но преди отварящата скоба на тялото на класа се слага ключовата дума **extends** следвана от името на базовия клас. Като направите така автоматично взимате всички членове-данни и методи на базовия клас. Ето пример:

```
//: c06:Detergent.java
// Inheritance syntax & properties

class Cleanser {
    private String s = new String("Cleanser");
    public void append(String a) { s += a; }
    public void dilute() { append(" dilute()"); }
    public void apply() { append(" apply()"); }
    public void scrub() { append(" scrub()"); }
    public void print() { System.out.println(s); }
    public static void main(String[] args) {
        Cleanser x = new Cleanser();
        x.dilute(); x.apply(); x.scrub();
        x.print();
    }
}

public class Detergent extends Cleanser {
    // Change a method:
    public void scrub() {
        append(" Detergent.scrub()");
        super.scrub(); // Call base-class version
    }
}
```

```

}
// Add methods to the interface:
public void foam() { append(" foam()"); }
// Test the new class:
public static void main(String[] args) {
    Detergent x = new Detergent();
    x.dilute();
    x.apply();
    x.scrub();
    x.foam();
    x.print();
    System.out.println("Testing base class:");
    Cleanser.main(args);
}
} //:~

```

Това демонстрира няколко неща. Първо, в **Cleanser.append()** метода **String**овете се конкатенират в **s** чрез използване на **+=** оператор, който е един от операторите (заедно с **+**) които проектантите на Java "претовариха" за да работят със **String**ове.

Второ, и **Cleanser** и **Detergent** съдържат **main()** метод. Може да създадете **main()** за всеки от вашите класове и често се препоръчва да се програмира по този начин за да се тества кодът затворен в класа. Даже и да имате много класове в програмата само **main()** за **public** класа извикан в командния ред на програмата ще бъде извикан. (И може да имате само един **public** клас на файл.) Така в този случай като напишем **java Detergent**, **Detergent.main()** ще бъде извикано. Но може също да се напише **java Cleanser** за да се извика **Cleanser.main()**, въпреки че **Cleanser** не е **public** клас. Тази техника на слагане на **main()** във всеки клас позволява лесно тестване на единиците във всеки клас. И не е необходимо да махате **main()** когато приключите тестването; може да го оставите за бъдещо тестване.

Може да видите как **Detergent.main()** вика **Cleanser.main()** явно.

Важно е че всички методи в **Cleanser** са **public**. Помните че ако не сложите никакъв спецификатор класът става "приятелски," което позволява достъп само до членовете в пакета. Така в същия пакет всеки би могъл да използва тези методи, ако нямат спецификатор на достъп. **Detergent** не би имал проблеми, например. Обаче ако клас от друг пакет наследи **Cleanser** той би имал достъп само до **public** членовете. Така че при планиране на наследяването правете всички членове **private** и всички методи **public**. (**protected** членовете също дават достъп на извлечения клас; ще изучите това по-късно.) Разбира се, за конкретен клас ще се постъпи конкретно, тук даваме една полезна насока.

Забележете че **Cleanser** има множество методи в интерфейса си: **append()**, **dilute()**, **apply()**, **scrub()** и **print()**. Понеже **Detergent** е извлечен от **Cleanser** (чрез ключовата дума **extends**) той автоматично взима всички тези методи в своя интерфейс, даже и да не ги виждате явно декларириани в **Detergent**. При това положение може да мислите за наследяването като за повторно използване на интерфейса. (Реализацията я имате бесплатно, но тази част не е по-важната.)

Както се вижда в **scrub()**, възможно е да се вземе метод от базовия клас и да се модифицира. В този случай може да поискате да използвате версията от базовия клас. Но в **scrub()** не може просто да извикате **scrub()**, тъй като това би произвело рекурсивно извикване, което не е желаното от вас. За да реши този проблем Java има ключовата дума **super** която идва от "superclass" което е класът от който е извлечен въпросният клас. Така изразът **super.scrub()** извика метода **scrub()** във версията на базовия клас.

Когато наследявате не сте ограничени до методите на базовия клас. Може също да добавите нови методи в новия клас точно както се прави това в клас: просто ги дефинирате.

Ключовата дума **extends** предполага че ще добавите нови методи в интерфейса на класа и методът **foam()** е пример за това.

В **Detergent.main()** може да видите че за **Detergent** може да викате всички методи достъпни в **Cleanser** както и в **Detergent** (т.е.. **foam()**).

Инициализиране на базовия клас

Тъй като сега участват два класа – базовия клас и извлечения клас – вместо само един, може да е малко смущаващо ако се опитаме да си представим обект, създаден от извлечения клас. Погледнато отвън може да изглежда, че новият метод има интерфейса на стария и евентуално нови методи. Но наследяването не просто копира интерфеса на стария клас. Когато създавате обект от извлечения клас той съдържа в себе си подобект от базовия клас. Този обект е същият както ако го създадете от самия базов клас. Погледнато отвън това е точно базовият клас да бъде обграден от извлечения.

Разбира се, че съдържаният обект трябва да се инициализира правилно и това може да стане само по един начин: езпълнява се инициализация в конструктора, като се извика конструктора на базовия клас, който е с достатъчно знание и привилегии за да се свърши необходимото. Java автоматично вмъква извиквания на конструктора на базовия клас. Следващия пример показва как става това за три нива на наследяване:

```
//: c06:Cartoon.java
// Constructor calls during inheritance

class Art {
    Art() {
        System.out.println("Art constructor");
    }
}

class Drawing extends Art {
    Drawing() {
        System.out.println("Drawing constructor");
    }
}

public class Cartoon extends Drawing {
    Cartoon() {
        System.out.println("Cartoon constructor");
    }

    public static void main(String[] args) {
        Cartoon x = new Cartoon();
    }
} ///:~
```

Изходът от тази програма показва автоматичните извиквания:

```
Art constructor
Drawing constructor
Cartoon constructor
```

Може да се види, че конструирането става "външно", за базовия клас, така че базовият клас се инициализира преди конструкторът на извлечения клас да има достъп до него.

Даже ако не създавате конструктор за **Cartoon()**, компилаторът ще синтезира сам конструктор по подразбиране, който ще вика конструктора на базовия клас.

Конструктори с аргументи

Горният пример има конструктори по подразбиране; тоест те нямат никакви аргументи. Лесно е за компилатора да вика такива конструктори, понеже не стои въпросът какви аргументи да се дадат. Ако вашият конструктор има аргументи или искате да извикате конструктора на базовия клас с аргументи, използвате ключовата дума **super** и съответния списък аргументи:

```
//: c06:Chess.java
// Наследяване, конструктори и аргументи

class Game {
    Game(int i) {
        System.out.println("Game constructor");
    }
}

class BoardGame extends Game {
    BoardGame(int i) {
        super(i);
        System.out.println("BoardGame constructor");
    }
}

public class Chess extends BoardGame {
    Chess() {
        super(11);
        System.out.println("Chess constructor");
    }
    public static void main(String[] args) {
        Chess x = new Chess();
    }
} //:~
```

Ако не викате конструктора на базовия клас в **BoardGame()**, компилаторът ще се оплаква че не може да намери конструктор за **Game()**. В добавка извикването на конструктора на базовия клас трябва да бъде първото нещо, което се прави в конструктора на извлечения клас. (Компилаторът ще ви напомни ако не го направите както трябва.)

Хващане на изключенията на базовия конструктор

Както току-що беше отбелязано, компилаторът ви принуждава да сложите извикването на конструктора на базовия клас като първо нещо в тялото на конструктора. Това значи че нищо друго не може да се появи преди него. Както ще видите в глава 9, това също предотвратява прихващането на изключение изхвърлено от базовия клас от извлечения клас. Това може да бъде неудобно в някои случаи.

Комбиниране на композиция и наследяване

Много често композицията и наследяването се използват заедно. Следният пример показва създаване на по-сложен клас чрез използване на наследяване и композиция, заедно с необходимата инициализация в конструкторите:

```
//: c06:PlaceSetting.java
// Combining composition & inheritance
```

```

class Plate {
    Plate(int i) {
        System.out.println("Plate constructor");
    }
}

class DinnerPlate extends Plate {
    DinnerPlate(int i) {
        super(i);
        System.out.println(
            "DinnerPlate constructor");
    }
}

class Utensil {
    Utensil(int i) {
        System.out.println("Utensil constructor");
    }
}

class Spoon extends Utensil {
    Spoon(int i) {
        super(i);
        System.out.println("Spoon constructor");
    }
}

class Fork extends Utensil {
    Fork(int i) {
        super(i);
        System.out.println("Fork constructor");
    }
}

class Knife extends Utensil {
    Knife(int i) {
        super(i);
        System.out.println("Knife constructor");
    }
}

// A cultural way of doing something:
class Custom {
    Custom(int i) {
        System.out.println("Custom constructor");
    }
}

public class PlaceSetting extends Custom {
    Spoon sp;
    Fork frk;
    Knife kn;
    DinnerPlate pl;
    PlaceSetting(int i) {
        super(i + 1);
        sp = new Spoon(i + 2);
    }
}

```

```

frk = new Fork(i + 3);
kn = new Knife(i + 4);
pl = new DinnerPlate(i + 5);
System.out.println(
    "PlaceSetting constructor");
}
public static void main(String[] args) {
    PlaceSetting x = new PlaceSetting(9);
}
} //:~

```

Докато компилаторът ви кара да инициализирате базовите класове и иска това да стане в самото начало на конструктора, той не ви следи дали инициализирате член-обектите, така че трябва да помните и да внимавате за това.

Гарантиране на провилно почистване

Java няма концепцията на C++ за деструктор, метод който автоматично се вика когато обектът се разрушава. Причината вероятно е че Java практиката е просто да забравим за обектите заместо да ги разрушаваме, позволявайки на боклучаря да освободи паметта когато е необходимо.

Често това е добре, но има случаи когато даден клас може да има активности, които после да изискват почистване на нещо. Както се спомена в глава 4, не се знае кога боклучарят ще се активира и дали въобще ще се активира. Така че ако искате нещо да се почиства, трябва да напишете специален метод който да прави това и да осигурите използването му от страна на клиент-програмиста. Отгоре на това, както е описано в глава 9 (exception handling), трябва да се предпазите от изключения слагайки го във **finally** клаузата.

Да вземем пример с CAD система която чертае нещо на екран:

```

//: c06:CADSystem.java
// Ensuring proper cleanup
import java.util.*;

class Shape {
    Shape(int i) {
        System.out.println("Shape constructor");
    }
    void cleanup() {
        System.out.println("Shape cleanup");
    }
}

class Circle extends Shape {
    Circle(int i) {
        super(i);
        System.out.println("Drawing a Circle");
    }
    void cleanup() {
        System.out.println("Erasing a Circle");
        super.cleanup();
    }
}

class Triangle extends Shape {
    Triangle(int i) {

```

```

super(i);
System.out.println("Drawing a Triangle");
}
void cleanup() {
System.out.println("Erasing a Triangle");
super.cleanup();
}
}

class Line extends Shape {
private int start, end;
Line(int start, int end) {
super(start);
this.start = start;
this.end = end;
System.out.println("Drawing a Line: " +
start + ", " + end);
}
void cleanup() {
System.out.println("Erasing a Line: " +
start + ", " + end);
super.cleanup();
}
}

public class CADSystem extends Shape {
private Circle c;
private Triangle t;
private Line[] lines = new Line[10];
CADSystem(int i) {
super(i + 1);
for(int j = 0; j < 10; j++)
lines[j] = new Line(j, j*j);
c = new Circle(1);
t = new Triangle(1);
System.out.println("Combined constructor");
}
void cleanup() {
System.out.println("CADSystem.cleanup()");
t.cleanup();
c.cleanup();
for(int i = 0; i < lines.length; i++)
lines[i].cleanup();
super.cleanup();
}
public static void main(String[] args) {
CADSystem x = new CADSystem(47);
try {
// Code and exception handling...
} finally {
x.cleanup();
}
}
}
} //:~

```

Всичко в тази система е някакъв вид **Shape** (което самото е вид **Object** понеже неявно е наследено от него клас). Всеки клас предефинира **cleanup()** на **Shape** в добавка на това че

вика същия метод на базовия клас чрез `super`. Специфичните `Shape` класове `Circle`, `Triangle` и `Line` всички имат конструктори които “чертаят,” а и всеки метод извикан по време на живота на програмата може да бъде подозиран че прави нещо, което иска почистване после. Всеки клас има свой собствен `cleanup()` метод за реставриране на нещата, които не са памет, до тяхното състояние преди създаването на обекта.

В `main()` може да се видят две нови ключови думи, които няма официално да се въвеждат до глава 9: `try` и `finally`. Ключовата дума `try` показва че блокът който следва (отделиен с фигурни скоби) е пазен регион, което значи че се третира специално. Една част от това специално третиране е че `finally` клаузата в този регион *винаги* се изпълнява, без значение как завърши `try` блокът. (С изключението е възможно `try` да завърши по необичайни начини.) Тук `finally` клаузата казва “винаги викай `cleanup()` за `x`, без значение какво се случва.” Тези ключови думи са обяснени напълно в глава 9.

Забележете, че вашият почистващ метод трябва да се грижи за реда на извикванията, ако подобектът зависи от друг обект. Изобщо ще следвате формата приета в C++ за деструкторите: Първо се извършва всичко специфично за вашия клас (което може да изисква елементите на базовия клас да са още жизнеспособни) и тогава да се извика почистващия метод на базовия клас, както е в примера.

Може да има много случаи, когато почистването не стои като проблем; просто оставяте боклучарят да си свърши работата. Но когато трябва да се намесите вие, приложение и старание са необходими.

Ред при почистването на боклука

Не е много това, на което може да се разчита, че се отнася до събирането на боклука. Боклучарят може никога да не се извика. Ако сработи, може да чисти обектите във всякакъв ред както си иска. В добавка реализациите на боклучаря в Java 1.0 често не викат `finalize()` методите. Най-добре е да не се разчита за нищо друго на боклучаря освен за освобождаването на паметта. Ако искате да правите почистване, направете си собствени методи и не разчитайте на `finalize()`. (Както се спомена по-рано Java 1.1 може да бъде заставен да вика всичките финализатори.)

Скриване на имената

Само C++ програмистите може да се изненадат от скриването на имената, понеже то работи различно в него език. Ако Java базов клас има име на метод което е претоварвано някако пъти, повторното дефиниране на това име в извлечен клас *не* скрива никоя от тези версии в базовия клас. Така претоварването работи независимо дали е станало сега или по-рано:

```
//: c06:Hide.java
// Overloading a base-class method name
// in a derived class does not hide the
// base-class versions

class Homer {
    char doh(char c) {
        System.out.println("doh(char)");
        return 'd';
    }
    float doh(float f) {
        System.out.println("doh(float)");
        return 1.0f;
    }
}
```

```

class Milhouse {}

class Bart extends Homer {
    void doh(Milhouse m) {}
}

class Hide {
    public static void main(String[] args) {
        Bart b = new Bart();
        b.doh(1); // doh(float) used
        b.doh('x');
        b.doh(1.0f);
        b.doh(new Milhouse());
    }
} ///:~

```

Както ще видите в седващата глава, много по-често се подтискат методи чрез използване на точно същото име и сигнатурата, както в базовия клас. Това може да бъде смущаващо все пак (понеже C++ го забранява за да не позволи да направите нещо, което вероятно е грешка).

Избиране на композиция vs. наследяване

И композицията и наследяването позволяват да се слага подобекти в клас. Може да се чудите за разликата помежду им и кога да изберете едното или другото.

Изобщо композицията се използва когато искате реализацията на един клас в друг, но не искате интерфейса. Тоест вграждате обект за да го използвате в новия си клас, но потребителят на новия клас вижда интерфейса който вие сте определили, а не интерфейса на онзи клас. За тази цел вграждате **private** обекти от съществуващи класове във вашите нови класове.

Понякога има смисъл да се позволи на потребителя на новия клас направо да има достъп до композицията му; тоест да се направят член-обектите **public**. Член-обектите използват реализацията скривайки се, така че това е безопасно да се направи и когато потребителят знае, че събирате много части заедно това прави интерфейса лесен за разбиране. **car** обектът е добър пример:

```

//: c06:Car.java
// Composition with public objects

class Engine {
    public void start() {}
    public void rev() {}
    public void stop() {}
}

class Wheel {
    public void inflate(int psi) {}
}

class Window {
    public void rollup() {}
    public void rolldown() {}
}

```

```

}

class Door {
    public Window window = new Window();
    public void open() {}
    public void close() {}
}

public class Car {
    public Engine engine = new Engine();
    public Wheel() wheel = new Wheel(4);
    public Door left = new Door(),
        right = new Door(); // 2-door
    Car() {
        for(int i = 0; i < 4; i++)
            wheel(i) = new Wheel();
    }
    public static void main(String[] args) {
        Car car = new Car();
        car.left.window.rollup();
        car.wheel(0).inflate(72);
    }
} ///:~

```

Понеже композицията на кола е част от анализа на проблема (а не просто част от подлежащото проектиране), правенето на членовете публични помага на клиента за разбирането и изиска по-малко сложен код от създателя на класа.

Когато се наследява се взема съществуващ клас и се прави специална негова версия. Изобщо това означава че се взема клас за обща употреба и се специализира за конкретни нужди. С малко размисъл ще видите, че е безсмислено да се композира кола с обект "превозно средство" – колата не съдържа превозно средство, тя е превозно средство. Тази е-зависимост се изразява с наследяването, а има-зависимостта се изразява с композицията.

protected

Сега като сте запознати с наследяването ключовата дума **protected** най-накрая има значение. В идеалния свят **private** биха били винаги непроменимо **private**, но в реалните проекти понякога искахме да направим нещо скрито от широкия свят и същевременно да оставим достъп на наследниците. Ключовата дума **protected** е съгласие с прагматизма. Тя казва "Това е **private** що се отнася до потребителя на класа, но е достъпно за наследниците на класа или за друг от същия **package**." Тоест **protected** в Java автоматично е "приятелски."

Най-добрия начин е да оставим членовете-данни **private** – винаги ще запазвате правото си да променяте подлежащата реализация. Може тогава да се позволи на потребителите контролиран достъп чрез **protected** методи:

```

//: c06:Orc.java
// The protected keyword
import java.util.*;

class Villain {
    private int i;
    protected int read() { return i; }
    protected void set(int ii) { i = ii; }
    public Villain(int ii) { i = ii; }
}

```

```

public int value(int m) { return m*i; }

}

public class Orc extends Villain {
    private int j;
    public Orc(int jj) { super(jj); j = jj; }
    public void change(int x) { set(x); }
} //:~

```

Може да видите че **change()** има достъп до **set()** понеже е **protected**.

Постъпкова разработка

Едно от предимствата на наследяването е че то поддържа *incremental development* чрез възможността да се добавя код без да се засяга съществуващ код. Това също изолира новите грешки в новия код. Чрез наследяване на съществуващ, функционален клас и добавяне на данни и методи (и предефиниране на съществуващи методи) оставяте съществуващия код – който някой друг може би още използва – недокоснат и небъгиран. Ако се случи да има грешка, знае се че тя е в новия код, който е много по-лесно да се прочете отколкото съществуващия такъв.

Малко изумяващо е колко чисто класовете се разделят. Даже не ви трябва сурсът за да използвате кода отново. Най-много да импортирате пакет. (Това е в сила и за композицията и за наследяването.)

Важно е да се разбере че разработката на програми е постъпков процес, точно както човешкото учене. Може да направите всичкия анализ на който сте способни, но още не знаете всичките отговори когато седнете над проекта. Ще имате много по-голям успех – и по-непосредствена обратна връзка – ако започнете да “израствате” своя проект като органическо, еволюционно творение, отколкото ако го направите изведенъж както се прави небостъргач от остьклени кутии.

Макар и наследяването за експериментиране да е полезна техника, в някаква точка когато нещата се постабилизират трябва да обгърнете цялата ѹерархия с намерение да се опости и реорганизира. Запомнете че зад наследяването се крие отношение което назава “*Този нов клас е от типа на онзи стар клас*.” Вашата програма не трябва да се занимава със сътвание и ресетване на битове, а със създаване на обекти, които се определя какви да бъдат от същината на проблемното пространство.

Upcasting

Най-важният аспект на наследяването не е, че дава методи на новия клас. Това е зависимостта между новия и стария клас. Тази зависимост може да бъде резюмирана като се каже “*новият клас е от типа на съществуващия клас*.”

Това описание не е просто фантазъорски начин да се изрази наследяването – то се поддържа направо от езика. Като пример да вземем базов клас наречен **Instrument** който представя музикалните инструменти и извлечен клас **Wind**. Понеже наследяването значи че всичките методи на базовия клас са достъпни и за извлечения клас, всяко съобщение изпратено до базовия клас също може да бъде изпратено до извлечения клас. Ако класа **Instrument** има **play()** метод, и **Wind** инструментите ще имат. Това значи че можем правилно да кажем че **Wind** обектът е също тип **Instrument**. Следващия пример показва как компилаторът поддържа това нещо:

```

//: c06:Wind.java
// Inheritance & upcasting

```

```

import java.util.*;

class Instrument {
    public void play() {}
    static void tune(Instrument i) {
        // ...
        i.play();
    }
}

// Wind objects are instruments
// because they have the same interface:
class Wind extends Instrument {
    public static void main(String[] args) {
        Wind flute = new Wind();
        Instrument.tune(flute); // Upcasting
    }
} //:~

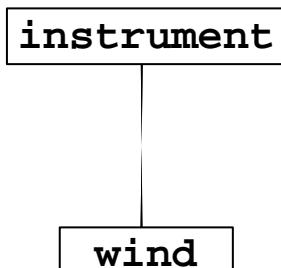
```

Интересното в този пример е **tune()** методът, който приема **Instrument** манипулятор. Обаче в **Wind.main()** **tune()** се вика като му се дава **Wind** манипулятор. Като знаем че Java е особено внимателен за проверката на типовете, изглежда странно че метод, който приема един тип приема с готовност и друг тип, докато не разберем че **Wind** обектът е също **Instrument** обект и няма метод който **tune()** може да извика за **Instrument** който не е също и **Wind**. Вътре в **tune()** кодът работи за **Instrument** и всичко извлечено от **Instrument** и актът на обръщане на **Wind** манипуляторът в **Instrument** манипулятор се нарича *upcasting*.

Защо “upcasting”?

Причината за термина е историческа и е свързана с начина на чертане на диаграмите на наследяването отгоре надолу, растейки надолу. (Разбира се, може да чертаете вашите диаграми както си искате.) Диаграмата на наследяването за **Wind.java** е тогава:

(диаграмата липсва в тази ревизия - б.пр.)



Кастингът от извлечения към базовия премества нагоре по диаграмата на наследяването, така че обикновено се говори за *upcasting*. Пкастингът е винаги безопасен понеже се отива от по-частен тип към по-общ. Тоест извлеченият клас е надмножество на базовия клас. Той може да съдържа повече методи от базовия клас, но трябва да съдържа най-малко методите на базовия клас. Единственото нещо което може да се случи при ъпкастинга е да се загубят методи, не да се придобият (за използване и от там — да се получат разминавания - бел.пр.). Поради това компилаторът въобще нищо не казва.

Може също да се направи обратното на ъпкастинг, наречено *downcasting*, но това довежда до дилема която е обект на глава 11.

ОТНОВО КОМПОЗИЦИЯ VS. НАСЛЕДЯВАНЕ

В ОО програмиране най-вероятният начин на работа е да пакетирате данни и методи в клас и да създавате обекти от този клас. От време на време ще използвате съществуващи класове чрез композиция. Още по-рядко ще използвате наследяване. Така че макар и наследяването да изисква най-много усилие при изучаването на ООП, това не значи, че ще го използвате навсякъде, където е възможно това да стане. Напротив, ще го използвате в единични случаи, само там, където е ясно, че наследяването е полезно. Един от най-добрите начини да се познае е да се прецени дали ще се налага ъпкастинг от извлечените класове към базовите. Ако трябва да се прави ъпкастинг, наследяването е необходимо, но ако ъпкастинг няма да е наложителен ще трябва по-добре да огледате дали е необходимо наследяване. Следващата глава (полиморфизъм) дава един от най-властните подтици за наследяване, но ако помните да се запитате “Трябва ли ми ъпкастинг?” ще имате добър инструмент за избор между композицията и наследяването.

Ключовата дума **final**

Ключовата дума **final** има малко различно значение в зависимост от контекста, но изобщо тя значи “Това не може да се променя.” Може да искате да предотвратите промените по две причини: дизайн и ефективност. Понеже тези две неща са доста различни, възможно е да се използва думата **final** неправилно.

Следващите секции разглеждат трите места където **final** може да се използва: за данни, методи и за клас.

Final данни

Много програмни езици имат начин да се заяви, че определени данни са “константи.” Константата е полезна по две причини:

1. Може да бъде константа по време на компилация която никога няма да се промени.
2. Може да бъде инициализирана по време на изпълнение и да не искате да се променя.

В случая на константа по време на компилация компилаторът може да използва константата навсякъде където трябва; тоест изчисленията се правят (веднъж -б.пр.) по време на компилация и не се правят допълнителни разходи по време на изпълнение. В Java този вид константи трябва да са примитиви и се отбелязват с ключовата дума **final**. Стойност трябва да се даде в момента на дефинирането на такава константа.

Поле което е и **static** и **final** има единствена частица от паметта, която не може да се променя.

Когато се използва **final** с обекти наместо с примитивни типове работата става малко смущаваща. С примитив **final** прави **стойността** константа, но с обектов манипулятор, **final** прави манипулятора константа. Манипуляторът трябва да бъде инициализиран с обект в момента на декларацията и манипуляторът никога не може да се промени да сочи друг обект. Обаче обектът може да се променя; Java не дава начин да се направи някой произволен обект константен. (Може да напишете, обаче, клас, чито обекти ефективно са константи.) Това ограничение включва масивите, които са също обекти.

Ето пример, който демонстрира полета **final**:

```
//: c06:FinalData.java
// The effect of final on fields

class Value {
    int i = 1;
```

```

}

public class FinalData {
    // Can be compile-time constants
    final int i1 = 9;
    static final int i2 = 99;
    // Typical public constant:
    public static final int i3 = 39;
    // Cannot be compile-time constants:
    final int i4 = (int)(Math.random()*20);
    static final int i5 = (int)(Math.random()*20);

    Value v1 = new Value();
    final Value v2 = new Value();
    static final Value v3 = new Value();
    //! final Value v4; // Pre-Java 1.1 Error:
    // no initializer
    // Arrays:
    final int[] a = { 1, 2, 3, 4, 5, 6 };

    public void print(String id) {
        System.out.println(
            id + ": " + i4 + " " +
            ", i5 = " + i5);
    }
}

public static void main(String[] args) {
    FinalData fd1 = new FinalData();
    //! fd1.i1++; // Error: can't change value
    fd1.v2.i++; // Object isn't constant!
    fd1.v1 = new Value(); // OK -- not final
    for(int i = 0; i < fd1.a.length; i++)
        fd1.a(i)++; // Object isn't constant!
    //! fd1.v2 = new Value(); // Error: Can't
    //! fd1.v3 = new Value(); // change handle
    //! fd1.a = new int(3);

    fd1.print("fd1");
    System.out.println("Creating new FinalData");
    FinalData fd2 = new FinalData();
    fd1.print("fd1");
    fd2.print("fd2");
}
} ///:~

```

Тъй като **i1** и **i2** са **final** примитиви със стойности известни по време на компилацията, те могат да се третират като константи по време на компилацията и не са различни по никакъв съществен начин. **i3** е по-типичен случай на дефиниране на такива константи: **public** за да бъдат видими извън пакета, **static** за да се подчертава че е единствена и **final** за това че е константа. Забележете че **final static** примитиви с константни начални стойности (т.е. константи по време на компилация) имат имена само от гоеми букви по конвенция. Също забележете че **i5** не може да бъде известно по време на компилация, така че името не е с големи букви.

Че нещо е **final** не значи непременно, че стойността му е известна по време на компилация. Това е демонстрирано чрез инициализирането на **i4** и **i5** по време на изпълнение чрез използване на генератор на случайни числа. Тази част на примера също показва разликата между правенето **final** стойност **static** и **не-static**. Тази разлика се проявява само при инициализация по време на изпълнение, понеже константите по време на компилация се

третират еднакво от компилатора. (И предполагаме се оптимизират предварително.) Разликата се вижда от изхода на програмата:

```
fd1: i4 = 15, i5 = 9
Creating new FinalData
fd1: i4 = 15, i5 = 9
fd2: i4 = 10, i5 = 9
```

Забележете че стойностите на **i4** за **fd1** и **fd2** са уникални, но стойността за **i5** не е променена при създаването на втори **FinalData** обект. Това е защото е **static** и се инициализира веднъж при натоварването, а не всеки път при създаването на обект.

Променливите **v1** до **v4** демонстрират какво значи **final** манипулятор. Както можете да видите в **main()**, само защото **v2** е **final** не означава, че не можете да промените стойността му. Не може обаче да пресвържете **v2** към нов обект точно защото е **final**. Това е, което **final** значи за манипулятор. Може да видите, че същото важи и за масив, който просто е друг тип манипулятор. (Не знам начин да направя самите манипулятори на масиви **final**.) Правенето на манипулятори **final** изглежда по-малко полезно от равенето на примитиви **final**.

Празни final

Java 1.1 позволява създаването на *blank finals*, които са полета декларирани като **final** но не им се дава инициализираща стойност. Във всички случаи те трябва да бъдат инициализирани преди да бъдат използвани и компилаторът осигурява това. Те дават много по-голяма гъвкавост в използването на ключовата дума **final** понеже, например, **final** поле в клас може сега да бъди различно за всеки обект и все пак да запази качеството си на "финален". Ето пример:

```
//: c06:BlankFinal.java
// "Blank" final data members

class Poppet { }

class BlankFinal {
    final int i = 0; // Initialized final
    final int j; // Blank final
    final Poppet p; // Blank final handle
    // Blank finals MUST be initialized
    // in the constructor:
    BlankFinal() {
        j = 1; // Initialize blank final
        p = new Poppet();
    }
    BlankFinal(int x) {
        j = x; // Initialize blank final
        p = new Poppet();
    }
    public static void main(String[] args) {
        BlankFinal bf = new BlankFinal();
    }
} ///:~
```

Принудени сте да дадете стойност на този род членове или по време на декларирането им, или във всеки конструктор. По този начин се гарантира, че всичко ще бъде инициализирано преди употребата.

Final аргументи

Java 1.1 позволява да се правят аргументи **final** чрез декларирането им като такива в аргументния списък. Това значи че вътре в метода не може да променяте това, към което сочи манипуляторът:

```
//: c06:FinalArguments.java
// Using "final" with method arguments

class Gizmo {
    public void spin() {}
}

public class FinalArguments {
    void with(final Gizmo g) {
        //! g = new Gizmo(); // Illegal -- g is final
        g.spin();
    }

    void without(Gizmo g) {
        g = new Gizmo(); // OK -- g not final
        g.spin();
    }

    // void f(final int i) { i++; } // Can't change
    // You can only read from a final primitive:
    int g(final int i) { return i + 1; }

    public static void main(String[] args) {
        FinalArguments bf = new FinalArguments();
        bf.without(null);
        bf.with(null);
    }
} ///:~
```

Забележете че може да се дава **null** манипулятор за аргумент който е **final** без компилаторът да се вайка, точно както за нефинален аргумент.

Методите **f()** и **g()** показват какво става когато примитивни аргументи са **final**: може да четете аргумента, но не може да го променяте.

Final методи

Има две причини за съществуването на **final** методи. Първата е да се "заключи" методът така че никой наследяваш клас да не може да му промени поведението. Това се прави по причини на дизайна когато искате да осигурите че ако класът се наследи методът няма да бъде подтиснат.

Втората причина е ефективността. Ако направите метод **final**, позволявате на компилатора да превърне извикванията му в *inline* извиквания. Когато компилаторът види извикване на **final** той може (по своя преценка) да пропусне нещата, които се правят при нормалния подход за извикване на методи (слагане на аргументите на стека, преход към кода на метода, преход обратно, почистване на стека и оправяне с върнатата стойност) и вместо тях може да сложи направо тялото на метода на мястото на извикването. Това елиминира допълнителните разходи за викането на метода. Разбира се, ако методът е голям, вашият код ще набъбне и няма да усетите никакво подобрение на скоростта, понеже съкращаването на работата е малко в сравнение с времето за изпълнение на самия метод. Предвидено е Java компилаторът да е в състояние да познае такива възможности и умно да прецени дали да направи даден **final** метод илайн. Обаче е по-добре да не се надържаме на възможностите на компилатора и

да правим методи **final** само ако са много малки или искаме явно да предотвратим възможността за подтискането им.

Всеки **private** в клас е имплицитно **final**. Понеже нямате достъп до **private** метод, не може да го подтиснете (ако и компилаторът не дава съобщение за грешка като се опитвате да го подтиснете, вие реално не го подтискате, а създавате нов метод). Може да добавите **final** спецификатора към **private** метод но това нищо не му дава в повече.

Final класове

Като кажете че цял клас е **final** (чрез предхождане на дефиницията му с ключовата дума **final**), заявявате че не щете да наследявате от този клас и не щете никой да го прави. С други думи, по никакви проектантски или от сигурността причини този клас никога не трябва да се променя или да се наследява от други класове. Алтернативно, може да е причината ефективността и искате работата с този клас да е толкова ефективна, колкото е възможно.

```
//: c06:Jurassic.java
// Making an entire class final

class SmallBrain {}

final class Dinosaur {
    int i = 7;
    int j = 1;
    SmallBrain x = new SmallBrain();
    void f() {}
}

//! class Further extends Dinosaur {}
// error: Cannot extend final class 'Dinosaur'

public class Jurassic {
    public static void main(String[] args) {
        Dinosaur n = new Dinosaur();
        n.f();
        n.i = 40;
        n.j++;
    }
} //:~
```

Забележете че данновите членове могат да бъдат **final** или не по ваш избор. Същите правила важат за **final** за данни членове без значение дали класът е **final**. Декларирането на клас като **final** просто предотврятва наследяването – нищо повече. Поради това предотврятване обаче всички методи на **final** клас са имплицитно **final**, понеже няма начин да се подтиснат. Така че компилаторът има същата възможност за подобряване на ефективността, както ако ги бяхте обявили **final**.

Може да добавите спецификатор **final** към метод на **final** клас, но нищо повече не се задава реално.

Final предпазливост

Може да изглежда добре да направите метод **final** докато проектирате клас. Може да чувствате че ефективността е много важна или че никой не трябва да го променя. Понякога това е така.

Но бъдете внимателни с предположенията. Изобщо е трудно да се предвиди как даден клас ще се използва в бъдеще. Ако определите метод като **final** бихте премахнали възможността вашият метод да се подтисне от друг програмист само защото не сте могли да си представите използването на вашия клас по неговия начин.

Стандартната Java библиотека е добър пример за това. В частност Java 1.0/1.1 **Vector** класът бе много използван и щеше да бъде още по-полезен ако, в името на ефективността, всички методи не бяха направени **final**. Лесно е да се предвиди, че ще е полезно да може да се наследява такъв полезен и общ клас, но кой знае защо проектантите му решили да забранят това. Това е иронично по две причини. Първо, **Stack** е наследен от **Vector**, което значи че **Stack** е **Vector**, което пък не е реално истина. Второ, много от най-важните методи на **Vector**, такива като **addElement()** и **elementAt()** са **synchronized**, което както ще видите в глава 14 вкарва значителни допълнителни разходи които по всяка вероятност ще унищожат подобренията вследствие на **final**. Това ни кара да даваме вяра на теорията, че програмистите са винаги лоши и се мъчат да познаят къде ще има оптимизация. Особено лошо е че такова лошо проектиране е направено в стандартна библиотека и всички ние ще трябва да се оправяме с него. (За щастие библиотеката за колекциите на Java 2 заменя **Vector** с **ArrayList**, който има много по-цивилизовано поведение.)

Интересно е също да се отбележи че **Hashtable**, друг важен клас на стандартна библиотека, няма **final** методи. Както се спомена на друго място в тази книга, очевидно е че различните класове са проектирани от съвършено различни хора. (Забележете краткостта на имената в **Hashtable** сравнени с онези във **Vector**.) Това е нещо от вид, който не трябва да е очевиден за потребителите на библиотеката. Когато нещата са недомислени това прави повече работа за потребителя. (Забележете че Java 2 библиотеката за колекциите заменя **Hashtable** със **HashMap**.)

Инициализация и товарене на класовете

В много от по-традиционните програмни езици програмите се товарят изведнъж като част от процеса на стартирането им. Това се следва от инициализация, а после програмата започва. Процесът на инициализация в тези езици трябва да бъде грижливо управляван така че **statics** да не причинява неприятности. C++, например, има проблеми ако един **static** очаква друг **static** да бъде валиден преди втория да бъде инициализиран.

Java няма този проблем поради различния си подход към товаренето. Понеже всичко в Java е обект много активности се удават по-лесно и тази е една от тях. Както ще научите в следващата глава, кодът за всеки обект е разположен в отделен файл. Този файл не се товари докато кодът не стане непосредствено необходим. Изобщо може да се каже, че кодът няма да се качва, докато не стане нужда да се конструира обект. Понеже може да има някои тънкости във връзка със **static** методите може също да се каже "Кодът на класа се товари в точката на първото използване."

Точката на първото използване е също където се инициализира **static**. Всичките **static** обекти и **static** кодови блокове ще се инициализират в текстова последователност (тоест в реда в който сте ги дефинирали в описанието на класа) в точката на товаренето. **static**, разбира се, се инициализират само веднъж.

Инициализация с наследяване

Полезно е да се погледне инициализационния процес в цялост, включително наследяването, за да се види цялата картина на това което става. Да вземем следващия код:

```

//: c06:Beetle.java
// The full process of initialization.

class Insect {
    int i = 9;
    int j;
    Insect() {
        prt("i = " + i + ", j = " + j);
        j = 39;
    }
    static int x1 =
        prt("static Insect.x1 initialized");
    static int prt(String s) {
        System.out.println(s);
        return 47;
    }
}

public class Beetle extends Insect {
    int k = prt("Beetle.k initialized");
    Beetle() {
        prt("k = " + k);
        prt('j = ' + j);
    }
    static int x2 =
        prt("static Beetle.x2 initialized");
    static int prt(String s) {
        System.out.println(s);
        return 63;
    }
    public static void main(String[] args) {
        prt("Beetle constructor");
        Beetle b = new Beetle();
    }
} ///:~

```

Изходът за тази програма е:

```

static Insect.x1 initialized
static Beetle.x2 initialized
Beetle constructor
i = 9, j = 0
Beetle.k initialized
k = 63
j = 39

```

Първото нещо което се случва като тръгне Java с **Beetle** е че лоудерът излиза и намира въпросния клас. В процеса на товаренето му лоудерът намира, че той има базов клас (това е, което ключовата дума **extends**казва), който тогава той товари. Това ще стане независимо дали ще правите обект от този базов клас. (Опитайте да изкоментирате създаването на клас за да го докажете на себе си.)

Ако базовият клас има базов клас, този втори базов клас тогава ще се натовари и така нататък. После **static** инициализация в кореновия базов клас (в този случай **Insect**) се изпълнява, после в съседния извлечен клас и т.н. Това е важно, защото инициализацията на статичните членове на наследения клас може да зависи от това дали членовете на базовия клас са инициализирани правилно.

В тази точка всички необходими класове са натоварени така че може да се създае обект. Първо всички примитиви в обекта се снабдяват със техните стойности по подразбиране и всички манипулатори получават стойност `null`. Тогава ще се извика конструкторът на базовия клас. В този случай извикването е автоматично, но също може да укажете извикване на конструктор (като първа операция в `Beetle()` конструктора) чрез `super`. Конструирането на базовия клас става в същия ред както на извлечения клас. След като конструкторът на базовия клас завърши работата си, променливите на екземпляра се инициализират по текстовия ред. Накрая се изпълнява останалата част от тялото на конструктора.

Резюме

И композицията и наследяването позволяват да се получат нови типове от съществуващи типове. Типично се използва композицията за използване на съществуващата реализация на даден тип и наследяване, когато искате повторно да използвате интерфейса. Тъй като извлеченият клас има интерфейса на базовия клас, може да се направи *upcast* към базовия, което е критично важно за полиморфизма, както ще научите в следващата глава.

Напук на силното ударение върху наследяването в ООП, когато започнете проектиране най-вече ще предпочтите композицията като начин и ще използвате наследяване само когато това е явно необходимо. (Както ще видите в следващата глава.) Композицията има тенденция да бъде по-гъвкава. В добавка, използвайки дона дената от наследяването хитрост, може да променяте точния тип, а с това и повезението, на членовете-обекти по време на изпълнение. Затова може да променяте поведението на композиран обект по време на изпълнение.

Въпреки че композицията и наследяването помагат за бързото разработване на проекти, изобщо ще искате да преразгледате йерархията си, преди да дадете на други програмисти да се занимават с нея. Вашата цел е всеки клас да има специфична употреба и да не бъде нито твърде голям (събирайки толкова функционалност, че да е съмнително дали ще се използва пак) нито дразнещо малък (та да не може да се използва без добавяне на функционалност). Като завършите класовете си те трябва да са лесни за многократно използване.

Упражнения

- Създайте два класа, **A** и **B**, с конструктори по подразбиране (празни аргументни списъци) които съобщават за себеси. Наследете нов клас наречен **C** от **A** и създайте член **B** вътре в **C**. Не създавайте конструктор за **C**. Създайте обект от клас **C** и наблюдавайте резултатите.
- Променете упражнение 1 така че **A** и **B** да имат конструктори с аргументи вместо конструктори по подразбиране. Напишете конструктор за **C** и направете всичката инициализация в конструктора на **C**.
- Вземете файла **Cartoon.java** и изкоментирайте конструктора на класа **Cartoon**. Обясните какво става.
- Вземете файла **Chess.java** и изкоментирайте конструктора на класа **Chess**. Обясните какво става.

7: Полиморфизъм

Полиморфизъмът е третата основна черта на един ООП език, след абстракцията на данните и наследяването.

Той дава друго измерение на разделянето на интерфейса от реализациите, разделя какво от как. Полиморфизъмът позволява подобрена организация и четимост на кода както и разширяеми програми, които могат да бъдат "разраснати" не само при първоначалното създаване на проекта, но и когато нови черти станат желани.

Капсулирането създава нови даннови типове чрез комбиниране на характеристики и поведение. Скриването на реализациите разделя реализациите от интерфейса чрез правенето на детайлите **private**. Този вид механична организация има очевиден смисъл за всеки, който се е занимавал с процедурно програмиране. Но полиморфизъмът се занимава с разделяне в термините на типове. В предната глава видяхте, че наследяването прави възможно третирането на обект като неговия си тип или типа на базовия клас. Тази възможност е критично важна понеже позволява много типове (извлечени от един базов тип) да се третират като че са от един тип и един и същ отрязък код да работи различно в различните случаи. Полиморфното извикване на методи позволява да се изрази разликата от друг, подобен тип, доколкото и двата са извлечени от един и същ базов тип. Това различаване се изразява чрез разликите в поведението на методите, които може да извикате чрез базовия клас.

В тази глава ще учене за полиморфизма (също наричан *динамично свързване* или *късно свързване* или *свързване по време на изпълнение*) започвайки от основното, с примери които махат от погледа всичко освен полиморфното поведение.

Upcasting

В глава 6 видяхме как обект може да се използва с неговия тип или с типа на базовия клас. Вземането на обектов манипулятор и третирането му като манипулятор от типа на базовия клас се нарича *upcasting* поради начина на изобразяване на дърветата на наследяването на хартия.

Видяхме също и възникващ проблем, който съществува и в следната програма: (Виж страница 63 ако има проблеми с пускането на програмата.)

```
//: c07:Music.java
// Inheritance & upcasting
package c07;

class Note {
    private int value;
    private Note(int val) { value = val; }
    public static final Note
        middleC = new Note(0),
        cSharp = new Note(1),
        cFlat = new Note(2);
} // Etc.

class Instrument {
    public void play(Note n) {
```

```

        System.out.println("Instrument.play()");
    }

}

// Wind objects are instruments
// because they have the same interface:
class Wind extends Instrument {
    // Redefine interface method:
    public void play(Note n) {
        System.out.println("Wind.play()");
    }
}

public class Music {
    public static void tune(Instrument i) {
        // ...
        i.play(Note.middleC);
    }
    public static void main(String[] args) {
        Wind flute = new Wind();
        tune(flute); // Upcasting
    }
} //:~

```

Методът **Music.tune()** приема **Instrument** манипулатор, но също и каквото и да е извлечено от **Instrument**. В **main()** може да се види това да става с **Wind** манипулатора даден на **tune()**, без да е необходим каст. Това е приемливо; интерфейсът в **Instrument** трябва да съществува в **Wind**, понеже **Wind** е наследен от **Instrument**. Ъпкастингът от **Wind** към **Instrument** може да "стесни" този интерфейс, но не може да го направи по-малък от интерфейса на **Instrument**.

Зашо ъпкастинг?

Тази програма може да ви изглежда странна. Защо трябва нарочно да се забрави типът на обекта? Това се случва при ъпкастинга и много по-праволинейно действие изглежда **tune()** да вземе просто **Wind** манипулатор като свой аргумент. Това изважда наяве най-важното: Ако го направите така, ще трябва да пишете нов **tune()** за всеки сорт **Instrument** във вашата система. Да кажем че последваме тази обосновка и добавим **Stringed** и **Brass** инструменти:

```

//: c07:Music2.java
// Overloading instead of upcasting

class Note2 {
    private int value;
    private Note2(int val) { value = val; }
    public static final Note2
        middleC = new Note2(0),
        cSharp = new Note2(1),
        cFlat = new Note2(2);
} // Etc.

class Instrument2 {
    public void play(Note2 n) {
        System.out.println("Instrument2.play()");
    }
}

```

```

class Wind2 extends Instrument2 {
    public void play(Note2 n) {
        System.out.println("Wind2.play()");
    }
}

class Stringed2 extends Instrument2 {
    public void play(Note2 n) {
        System.out.println("Stringed2.play()");
    }
}

class Brass2 extends Instrument2 {
    public void play(Note2 n) {
        System.out.println("Brass2.play()");
    }
}

public class Music2 {
    public static void tune(Wind2 i) {
        i.play(Note2.middleC);
    }

    public static void tune(Stringed2 i) {
        i.play(Note2.middleC);
    }

    public static void tune(Brass2 i) {
        i.play(Note2.middleC);
    }

    public static void main(String[] args) {
        Wind2 flute = new Wind2();
        Stringed2 violin = new Stringed2();
        Brass2 frenchHorn = new Brass2();
        tune(flute); // No upcasting
        tune(violin);
        tune(frenchHorn);
    }
} //:~

```

Това работи, но има голям недостатък: Трябва да се пишат специфични за типа методи за всеки **Instrument2** клас който се добави. На първо място това значи повече програмиране, но също значи че ако добавите нови методи като **tune()** или нов вид **Instrument**, ще има много работа за свършване. Като добавим факта, че компилаторът няма да издаде никакво съобщение ако забравите да пренатоварите методите си целият процес на работа с методите се вижда явно неуправляем.

Не би ли било много по-хубаво ако веднъж пишете метод който взима типа на базовия клас за аргумент, а не на специфичен клас? Тоест, не би ли било най-хубаво да забравите за извлечениите класове и да пишете методи само за базовия клас?

Точно това е което полиморфизъмът позволява да се прави. Обаче повечето програмисти (които са програмирали процедурно преди) малко имат проблеми с начина на работа на полиморфизма.

Особеността

Трудността с **Music.java** може да се види чрез пускане на програмата. Изходът е **Wind.play()**. Това е точно желаният изход, но не изглежда да има смисъл да работи по този начин. Погледнете **tune()** метода:

```
public static void tune(Instrument i) {  
    // ...  
    i.play(Note.middleC);  
}
```

Той приема **Instrument** манипулятор. Така че как е възможно компилаторът да знае че **Instrument** манипуляторът сочи **Wind** в този случай а не **Brass** или **Stringed**? Компилаторът не може да знае. За да се постигне по-дълбоко разбиране на въпроса първо трябва да погледнем отблизо свързването.

Свързване при извикването на метод

Свързването на извикване на метод с тялото на метода се нарича *binding*. Когато то се направи преди програмата да е стартирана (от компилатора и линкера, ако има такъв), това се нарича *ранно свързване*. Може да не сте чували термина преди понеже той никога не е съществувал като алтернатива в процедурното програмиране. С компилаторите имат само един начин за викане на функции и това е ранното свързване.

Смущаващата част от горната програма се върти около ранното свързване, понеже компилаторът не би могъл да знае кой метод да извика след като приема **Instrument** манипулятор.

Решението се нарича *късно свързване*, което значи че свързването става по време на изпълнение и е основано на типа на обекта. Късното свързване също се нарича *динамично свързване* или *свързване по време на изпълнение*. Когато в езика има късно свързване, трябва да има и съответен механизъм за разпознаване на типа и викане на подходящия метод по време на изпълнение. Тоест компилаторът пак не знае типа на обекта, но механизъмът на извикването на методи го намира и скача в необходимото тяло на метод. Механизъмът на късното свързване се мени от език на език, но е лесно да се съобрази, че някакъв вид запис на информация в обектите би трябвало да съществува.

В Java се използва късно свързване освен ако методът е деклариран като **final**. Това значи че обикновено не трябва да се замисляте за типа на свързването – то става автоматично.

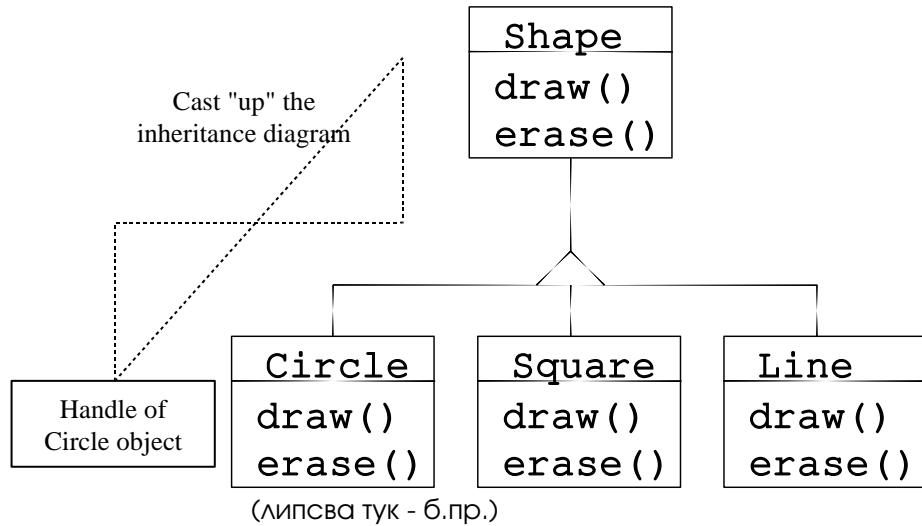
Зашо бихте декларирали метод като **final**? Както беше отбелязано в предната глава, това предотвратява изменянето му от когото и да било. Може би по-важно, това ефективно “изключва” динамичното свързване или по-скоро казва на компилатора че то не е нужно. Това позволява на компилатора да генерира по-ефективен код за извикванията на методи които са **final**.

Постигане на точното поведение

След като знаете че свързването в Java полиморфно и късно, вие можете да направите вашия код да говори на базовия клас и да сте сигурни, че това ще става с всякакви извлечени класове. Или, казано с други думи “пращате съобщение на обект и го оставяте да реши какво точно требва да направи.”

Класическият пример в ООП е примерът “форма”. Това се използва всеобщо поради лесната му визуализация, но за нещастие може да смути начинаещия програмист и да го накара да мисли, че ОО програмирането е точно за програмиране на графика, което разбира се не е така.

Примерът има базов клас наречен **Shape** и различни извлечени типове: **Circle**, **Square**, **Triangle** и т.н. Причината примерът да е толкова подходящ е че е лесно да се каже "кръгът е вид форма" и това да бъде разбрано. Диаграмата на наследяването показва зависимостите:



Ъпкастингът може да се наложи в ред простичък като този:

```
| Shape s = new Circle();
```

Тук **Circle** обект се създава и манипулаторът непосредствено се присвоява на **Shape**, което би изглеждало грешка (присвояване един тип на друг) и все пак всичко е наред понеже **Circle** е **Shape** по наследство. Така че компилаторът се съгласява с операторите и не издава съобщение за грешка.

Когато викате един от методите на базовия клас (които са били подтиснати в идвлечения клас):

```
| s.draw();
```

отново бихте могли да очаквате че **draw()** на **Shape** се вика понеже това е, най-сетне, манипулятор на **Shape** токо че откъде да знае компилаторът да прави нещо друго? И пак правилният (метод-б.пр.) **Circle.draw()** се вика поради късното свързване (полиморфизма).

Следващия пример показва това по малко различен начин:

```
//: c07:Shapes.java
// Polymorphism in Java

class Shape {
    void draw() {}
    void erase() {}
}

class Circle extends Shape {
    void draw() {
        System.out.println("Circle.draw()");
    }
    void erase() {
        System.out.println("Circle.erase()");
    }
}

class Square extends Shape {
```

```

void draw() {
    System.out.println("Square.draw()");
}
void erase() {
    System.out.println("Square.erase()");
}
}

class Triangle extends Shape {
void draw() {
    System.out.println("Triangle.draw()");
}
void erase() {
    System.out.println("Triangle.erase()");
}
}

public class Shapes {
public static Shape randShape() {
switch((int)(Math.random() * 3)) {
default: // To quiet the compiler
case 0: return new Circle();
case 1: return new Square();
case 2: return new Triangle();
}
}
public static void main(String[] args) {
Shape[] s = new Shape[9];
// Fill up the array with shapes:
for(int i = 0; i < s.length; i++)
s[i] = randShape();
// Make polymorphic method calls:
for(int i = 0; i < s.length; i++)
s[i].draw();
}
} //:~

```

Базовият клас **Shape** основава общ интерфейс за всичко, което е наследено от **Shape** – тоест всички форми могат да бъдат чертани и трити. Извлечениите класове подискат тези дефиниции за да може да се осигури специфично необходимото поведение за всяка форма.

Главният клас **Shapes** съдържа **static** метод **randShape()** който произвежда манипулятор към случайно избран **Shape** обект всеки път когато го викате. Забележете че ъпкастинг става с всеки от **return** операторите, който взима манипулятор **Circle**, **Square**, или **Triangle** и го изпраща извън метода като връщан тип, **Shape**. Така че когато и да извикате този метод, никога нямате шанс да видите какъв точно тип е, понеже получавате обратно просто манипулятор **Shape**.

main() съдържа масив от **Shape** манипулятори запълнен чрез извиквания на **randShape()**. В този момент вие знаете че имате **Shapes** (**Форми** -б.пр.), но не знаете нищо по специфично от това (а също и компилаторът не знае). Обаче като се движите през масива и викате **draw()** за всеки един, коректното специфично за типа поведение магически се проявява, както може да видите от примера на изхода:

```

Circle.draw()
Triangle.draw()
Circle.draw()

```

```

Circle.draw()
Circle.draw()
Square.draw()
Triangle.draw()
Square.draw()
Square.draw()

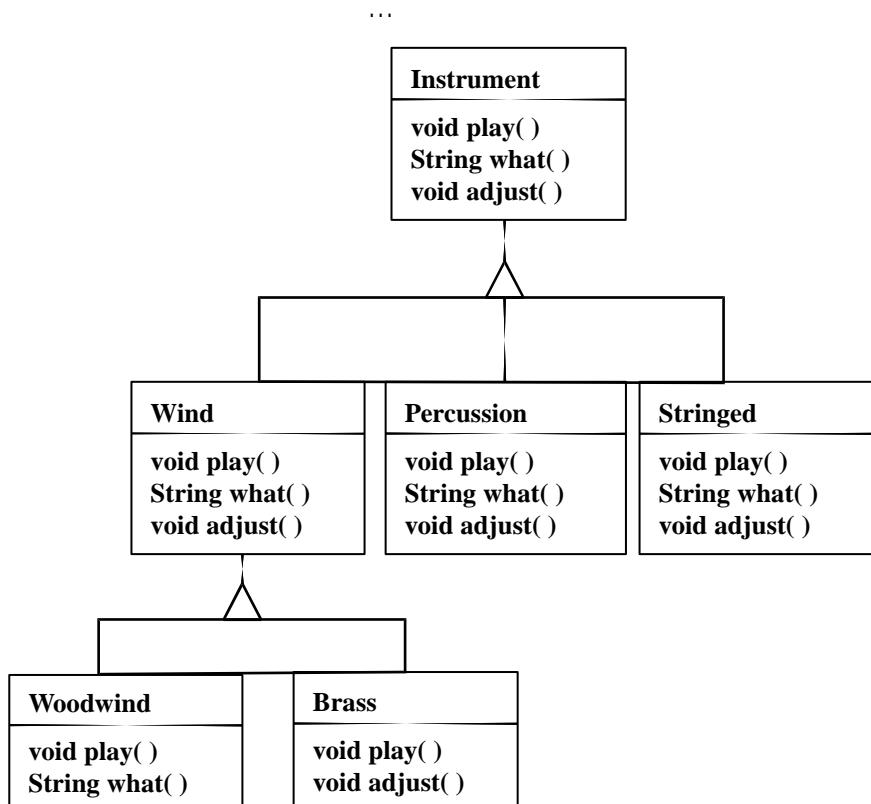
```

Разбира се тъй като формите са случаен избор всеки път, изходът от вашите пускания все ще е различен. Доводът да се избере случаен избор е да се покаже, че компилаторът няма никакви допълнителни знания, които биха му помогнали предварително да вземе решение. Всичките викания на `draw()` са чрез динамично свързване.

Разширяемост

Сега нека да се върнем към примера с музикалния инструмент. Поради полиморфизма може да добавяте колкото си искате типове без да изменяте метода `tune()`. В добре проектирана ОО програма повечето или всичките ви методи ще следват примера на `tune()` и ще комуникират само с интерфейса на базовия клас. Такава програма е разширяема понеже може да се добавя функционалност чрез наследяване но абекти от базовия клас. Методите които манипулират интерфейса на базовия клас не е необходимо въобще да се пипат за да работят и с новите класове.

Да видим какво ще стане ако вземем примера с инструментите и добавим нови методи към базовия клас и няколко нови класа. Ето диаграмата:



Всички нови класове работят перфектно със стария, непроменен `tune()` метод. Даже ако `tune()` е в отделен файл и нови методи са добавени в интерфейса на `Instrument`, `tune()` работи коректно без рекомпилиация. Ето реализацията на горната програма:

```

//: c07:Music3.java
// An extensible program
import java.util.*;

```

```

class Instrument3 {
    public void play() {
        System.out.println("Instrument3.play()");
    }
    public String what() {
        return "Instrument3";
    }
    public void adjust() {}
}

class Wind3 extends Instrument3 {
    public void play() {
        System.out.println("Wind3.play()");
    }
    public String what() { return "Wind3"; }
    public void adjust() {}
}

class Percussion3 extends Instrument3 {
    public void play() {
        System.out.println("Percussion3.play()");
    }
    public String what() { return "Percussion3"; }
    public void adjust() {}
}

class Stringed3 extends Instrument3 {
    public void play() {
        System.out.println("Stringed3.play()");
    }
    public String what() { return "Stringed3"; }
    public void adjust() {}
}

class Brass3 extends Wind3 {
    public void play() {
        System.out.println("Brass3.play()");
    }
    public void adjust() {
        System.out.println("Brass3.adjust()");
    }
}

class Woodwind3 extends Wind3 {
    public void play() {
        System.out.println("Woodwind3.play()");
    }
    public String what() { return "Woodwind3"; }
}

public class Music3 {
    // Doesn't care about type, so new types
    // added to the system still work right:
    static void tune(Instrument3 i) {
        // ...
        i.play();
    }
}

```

```

static void tuneAll(Instrument3[] e) {
    for(int i = 0; i < e.length; i++)
        tune(e[i]);
}
public static void main(String[] args) {
    Instrument3[] orchestra = new Instrument3[5];
    int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind3();
    orchestra[i++] = new Percussion3();
    orchestra[i++] = new Stringed3();
    orchestra[i++] = new Brass3();
    orchestra[i++] = new Woodwind3();
    tuneAll(orchestra);
}
} //:~

```

Новите методи са **what()**, който връща **String** манипулятор с описание на класа и **adjust()**, който дава някакъв начин за нагласяване на всеки инструмент.

В **main()** когато слагате нещо вътре в **Instrument3** масива автоматично става ъпкастинг към **Instrument3**.

Може да видите че метода **tune()** е блажено невеж относно промените на кода станали около него, и все пак работи коректно. Това е точно нещото, което се очаква да даде полиморфизъмът. Промените във вашия код не засягат частите на програмата, които не трябва да се засягат. Казано с други думи полиморфизъмът е най-важният инструмент който помага на програмиста да "отдели нещата които се променят от тези които ще останат така."

Подтискане vs. претоварване

Нека да погледнем по друг начин на нашия пример. В следната програма интерфейсът на **play()** е променен в процеса на подтискането му, което значи че той не е подтиснат (метода), а е претоварен. Компилаторът позволява да се претоварват методи затова не се оплаква. Поведението обаче вероятно не съвпада с желаното. Ето примера:

```

//: c07:WindError.java
// Accidentally changing the interface

class NoteX {
    public static final int
        MIDDLE_C = 0, C_SHARP = 1, C_FLAT = 2;
}

class InstrumentX {
    public void play(int NoteX) {
        System.out.println("InstrumentX.play()");
    }
}

class WindX extends InstrumentX {
    // OOPS! Changes the method interface:
    public void play(NoteX n) {
        System.out.println("WindX.play(NoteX n)");
    }
}

```

```

public class WindError {
    public static void tune(InstrumentX i) {
        // ...
        i.play(NoteX.MIDDLE_C);
    }
    public static void main(String[] args) {
        WindX flute = new WindX();
        tune(flute); // Not the desired behavior!
    }
} //:~

```

Тук изпъква и друг смущаващ аспект. В **InstrumentX** методът **play()** взема **int** с идентификатор **NoteX**. Тоест въпреки че **NoteX** е име на клас, то също може да се използува като идентификатор без оплаквания. Но в **WindX** **play()** взима **NoteX** манипулятор който има за идентификатор **n**. (Въпреки че може даже да се напише **play(NoteX NoteX)** без грешка.) По този начин сякаш програмистът е възнамерявал да подтисне **play()** но е направил малка печатна грешка. Компилаторът, обаче, е предположил че претоварване наместо подтискане е било намерението. Забележете, че ако следвате стандартната за имената в Java конвенция идентификаторът на аргумента би бил **noteX**, което би го отличавало от име на клас.

В **tune InstrumentX i** е изпратеното от **play()** съобщение с един от членовете на **NoteX** — (**MIDDLE_C**) като аргумент. Тъй като **NoteX** съдържа **int** дефиниции, това значи че **int** версията на сега претоварения **play()** метод се вика, а понеже той не е подтиснат се използва версията му в базовия клас.

Изходът е:

```
| InstrumentX.play()
```

Това сигурно не е полиморфно извикване на метод. Като веднъж сте разбрали какво става, може да оправите нещата много лесно, но представете си колко трудно би могло да бъде това, ако този ред бе погребан в голяма програма.

Абстрактни класове и методи

Във всичките примери с инструменти методите на базовия клас **Instrument** бяха винаги “празни” методи. Щеше да е нередно, грешно, ако тези методи бяха някога викани. Така е защото целта на **Instrument** е да създаде общ интерфейс за всичките класове извлечени от него.

Единствената причина да се създаде такъв общ интерфейс е че така той може да се изрази различно за различните извлечени типове. Съставя се основна форма, така че може да се каже какво е общо между всичките типове. Друг начин да се изкаже това е назоваването на **Instrument** абстрактен базов клас (или просто **абстрактен клас**). Създавате абстрактен клас когато искате да манипулирате множество от обекти чрез неговия общ интерфейс. Всички методи на извлечен клас които имат същата сигнатура като на съответния метод на базовия клас ще бъдат викани автоматично когато трябва чрез механизма на късното свързване. (Обаче, както се видя в предната секция, ако името е същото но сигнатурата различна ще се получи фактически не подтискане, а претоварване, което надали е желания резултат.)

Ако имате абстрактен клас като **Instrument**, обектите които бихте създали от него почти винаги нямат смисъл. Тоест **Instrument** е само за да изрази интерфейса, а не конкретна реализация, така че създаване на обекти от **Instrument** няма смисъл и вероятно вие ще искате да пресовратите създаването им от потребителя. Това може да се изпълни като се направят всичките методи на **Instrument** да извеждат съобщения за грешка, но това забавя обратната връзка чак до времето на изпълнение и изисква изтощително тестване от страна на потре-

бителя (на класа - бел.пр.). Винаги е по-добре да се хващат проблемите още по време на компилация.

Java има механизъм за случая наречен *абстрактен метод*. Това е незавършен метод; само декларация без тяло на метода. Ето синтаксисът на декларацията на абстрактен метод:

```
| abstract void X();
```

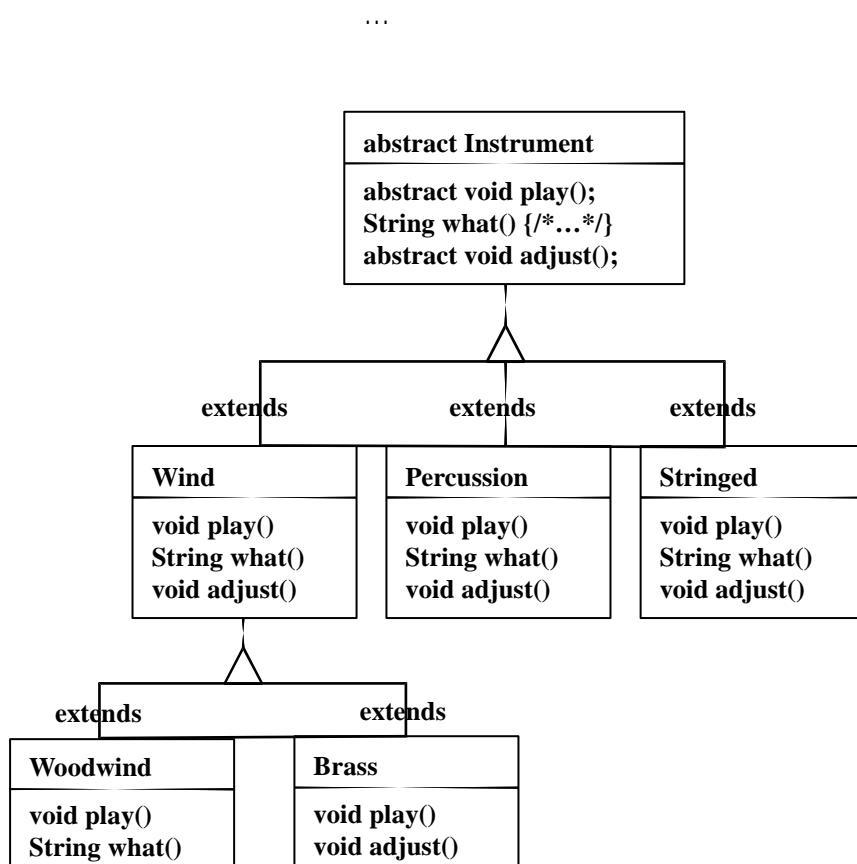
Клас съдържащ абстрактни методи се нарича *абстрактен клас*. Ако класът съдържа един или повече абстрактни методи той задължително трабва да бъде квалифициран с **abstract**. (Иначе компилаторът ще издаde съобщение за грешка.)

Ако абстрактният клас е незавършен, какво се очаква да направи компилаторът ако някой се опитва да създаде обект от него? Не може безопасно да се създаде обект от абстрактен клас, токо че ще получите съобщение за грешка от компилатора. По този начин компилаторът осигурява чистотата на абстрактния клас и няма нужда да се беспокоите от погрешно използване.

Ако наследите от абстрактен клас и искате да се създават обекти от новия тип, трябва да напишете методи които подтискат всички абстрактни методи. Ако не го направите (и може да изберете този вариант) извлеченият клас също е абстрактен и компилаторът ще ви застави да квалифицирате този клас с ключовата дума **abstract**.

Възможно е да се декларира клас с **abstract** без да се включват **abstract** методи. Това е полезно когато не искате да има **abstract** методи и все пак искате да предотвратите създаване на обекти от него клас.

Класът **Instrument** лесно може да се превърне в абстрактен. Само някои от методите ще бъдат абстрактни, понеже не се изисква всичките да бъдат. Ето как изглежда:



Ето оркестърският пример променен да използва **abstract** класове и методи:

```

//: c07:Music4.java
// Abstract classes and methods
import java.util.*;

abstract class Instrument4 {
    int i; // storage allocated for each
    public abstract void play();
    public String what() {
        return "Instrument4";
    }
    public abstract void adjust();
}

class Wind4 extends Instrument4 {
    public void play() {
        System.out.println("Wind4.play()");
    }
    public String what() { return "Wind4"; }
    public void adjust() {}
}

class Percussion4 extends Instrument4 {
    public void play() {
        System.out.println("Percussion4.play()");
    }
    public String what() { return "Percussion4"; }
    public void adjust() {}
}

class Stringed4 extends Instrument4 {
    public void play() {
        System.out.println("Stringed4.play()");
    }
    public String what() { return "Stringed4"; }
    public void adjust() {}
}

class Brass4 extends Wind4 {
    public void play() {
        System.out.println("Brass4.play()");
    }
    public void adjust() {
        System.out.println("Brass4.adjust()");
    }
}

class Woodwind4 extends Wind4 {
    public void play() {
        System.out.println("Woodwind4.play()");
    }
    public String what() { return "Woodwind4"; }
}

public class Music4 {
    // Doesn't care about type, so new types
    // added to the system still work right:
    static void tune(Instrument4 i) {

```

```

// ...
i.play();
}
static void tuneAll(Instrument4[] e) {
    for(int i = 0; i < e.length; i++)
        tune(e[i]);
}
public static void main(String[] args) {
    Instrument4[] orchestra = new Instrument4[5];
    int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind4();
    orchestra[i++] = new Percussion4();
    orchestra[i++] = new Stringed4();
    orchestra[i++] = new Brass4();
    orchestra[i++] = new Woodwind4();
    tuneAll(orchestra);
}
} ///:~

```

Може да видите че няма промени освен в базовия клас.

Полезно е да се създават **abstract** класове и методи понеже те правят явна абстрактността на класа и казват и на потребителя и на компилатора какво се цели.

Интерфейси

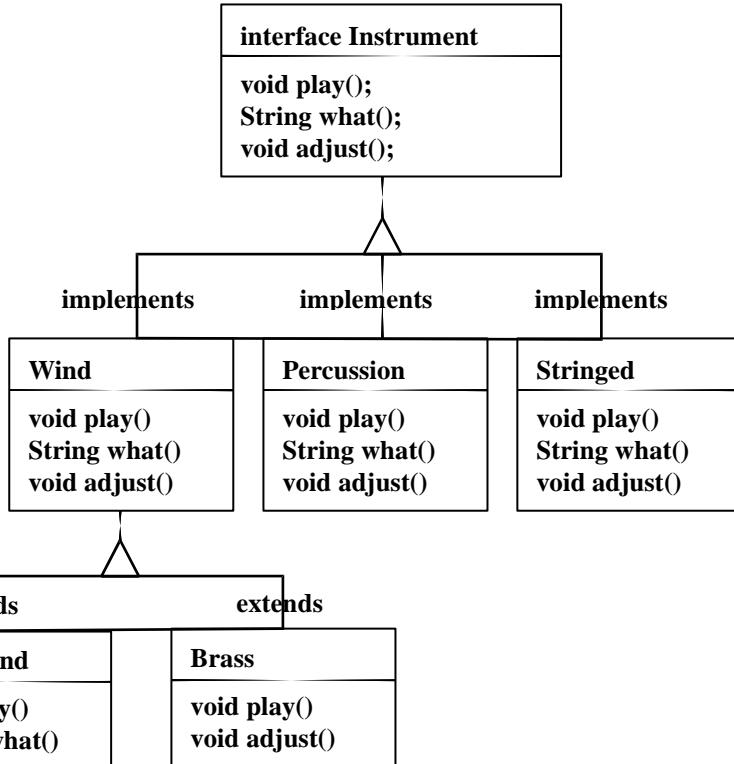
Ключовата дума **interface** придвижква концепцията за абстрактност една стъпка напред. Може да си я мислим като "чисто" абстрактен клас. Тя позволява на създателя да определи формата на класа: имената на методите, аргументните списъци и връщаните стойности, но не и телата на методите. Един **interface** може също да съдържа данни членове от примитивни типове, но те са имплицитно **static** и **final**. **interface** дава само форма, не реализация.

interface казва: "Така ще изглеждат класовете реализиращи този конкретен интерфейс." Така всеки код, който използва конкретен **interface** знае кои методи могат да се викат с него **interface** и това е всичко. Така че **interface** се използва да се установи "протокол" между класове. (Някои ООП езици използват думата *protocol* за същата цел.)

За да се създае **interface**, използвайте ключовата дума **interface** вместо ключовата дума **class**. Както за клас може да добавите ключовата дума **public** преди ключовата дума **interface** (но само ако този **interface** е дефиниран във файл със същото име) или да не го добавите, оставяйки "приятелски" статус.

За да се направи клас да отговаря на конкретен **interface** (или група от **interface**-и) използвайте ключовата дума **implements**. Вие казвате "interface-ът е за как изглежда а ето как работи." Отделно от това има голямо сходство с наследяването. Диаграмата за примера с инструментите показва това:

...



Веднъж като реализирате **interface**, тази реализация става обикновен клас който може да бъде разширяван по обичайния начин.

Може да изберете явно да декларирайте методите в **interface** като **public**. Но те са **public** и ако не го направите. Така че когато **implement**-ирате един **interface**, методите от **interface** трябва да са дефинирани като **public**. Иначе ще станат по подразбиране "приятелски" и по този начин ще ограничите достъпността им при наследяване, което не е позволено от Java компилатора.

Може да видите това в променена версия на примера с **Instrument**. Забележете че всеки метод в **interface** е само декларация, което е единственото нещо, позволявано от компилатора тук. В добавка никой от методите в **Instrument5** не е деклариран като **public**, но все пак те са автоматично **public**:

```

//: c07:Music5.java
// Interfaces
import java.util.*;

interface Instrument5 {
    // Compile-time constant:
    int i = 5; // static & final
    // Cannot have method definitions:
    void play(); // Automatically public
    String what();
    void adjust();
}

class Wind5 implements Instrument5 {
    public void play() {
        System.out.println("Wind5.play()");
    }
    public String what() { return "Wind5"; }
    public void adjust() {}
}

```

```

class Percussion5 implements Instrument5 {
    public void play() {
        System.out.println("Percussion5.play()");
    }
    public String what() { return "Percussion5"; }
    public void adjust() {}
}

class Stringed5 implements Instrument5 {
    public void play() {
        System.out.println("Stringed5.play()");
    }
    public String what() { return "Stringed5"; }
    public void adjust() {}
}

class Brass5 extends Wind5 {
    public void play() {
        System.out.println("Brass5.play()");
    }
    public void adjust() {
        System.out.println("Brass5.adjust()");
    }
}

class Woodwind5 extends Wind5 {
    public void play() {
        System.out.println("Woodwind5.play()");
    }
    public String what() { return "Woodwind5"; }
}

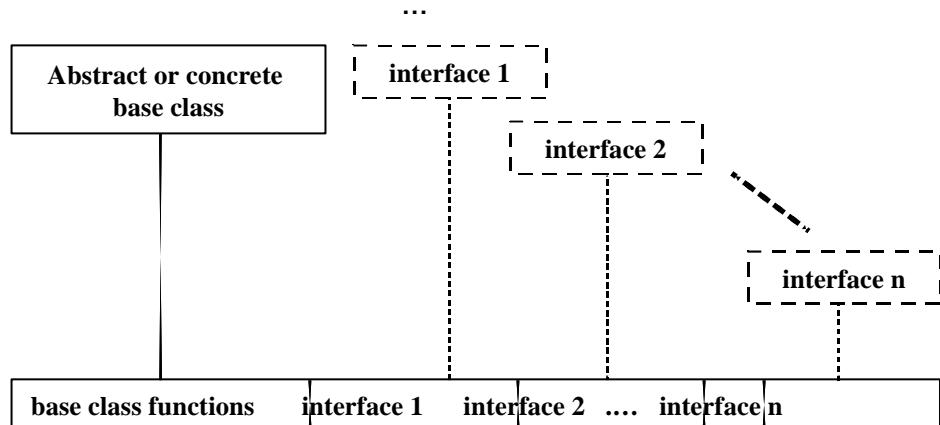
public class Music5 {
    // Doesn't care about type, so new types
    // added to the system still work right:
    static void tune(Instrument5 i) {
        // ...
        i.play();
    }
    static void tuneAll(Instrument5[] e) {
        for(int i = 0; i < e.length; i++)
            tune(e[i]);
    }
    public static void main(String[] args) {
        Instrument5[] orchestra = new Instrument5[5];
        int i = 0;
        // Upcasting during addition to the array:
        orchestra[i++] = new Wind5();
        orchestra[i++] = new Percussion5();
        orchestra[i++] = new Stringed5();
        orchestra[i++] = new Brass5();
        orchestra[i++] = new Woodwind5();
        tuneAll(orchestra);
    }
} ///:~

```

Останалата част работи по стария начин. Няма значение дали провите ъпкаст към "обикновен" клас с име **Instrument5**, **abstract**-ен клас наречен **Instrument5**, или към **interface** наречен **Instrument5**. Поведението е същото. Фактически може да видите в метода **tune()** че няма никакво свидетелство че **Instrument5** е "обикновен" клас, **abstract**-ен клас или **interface**. Това е целта: Всеки подход дава на програмиста различен начин за управление на създаването и използването на обекти.

"Многократно наследяване" в Java

interface не е просто "по-чиста" форма на **abstract**-ен клас. Той има по-важно предназначение от това. Понеже **interface** въобще няма реализация – тоест няма памет асоциирана с **interface** – няма и нищо, което да пречи много **interface**-и да бъдат комбинирани. Това е ценно понеже по някой път е нужно да може да се каже "х е а и б и с." В C++ комбинирането на много класове се нарича **многократно наследяване** и носи малко ограничаващ действията багаж понеже всеки клас си има реализация. В Java може да се направи същото, но само един от класовете може да има реализация, така че проблемът от C++ не се появява в Java когато се комбинират много интерфейси:



В извлечен клас не сте заставени да имате базов клас който е или **abstract** или "конкретен" (такъв който няма **abstract**-ни методи). Ако **наследите от не-interface**, може да наследите само от един. Всички останали базови елементи трябва да бъдат **interface**-и. Слагате всички имена за наследяване след ключовата дума **implements** и ги разделяте със запетай. Може да имате колкото искате **interface**-и и всеки става независим тип, към който може да се прави кастинг. Следния пример показва конкретен клас комбиниран с няколко **interface**-а за да се получи нов клас:

```

//: c07:Adventure.java
// Multiple interfaces
import java.util.*;

interface CanFight {
    void fight();
}

interface CanSwim {
    void swim();
}

interface CanFly {
    void fly();
}

class ActionCharacter {

```

```

public void fight() {}

}

class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {
    public void swim() {}
    public void fly() {}
}

public class Adventure {
    static void t(CanFight x) { x.fight(); }
    static void u(CanSwim x) { x.swim(); }
    static void v(CanFly x) { x.fly(); }
    static void w(ActionCharacter x) { x.fight(); }
    public static void main(String[] args) {
        Hero i = new Hero();
        t(i); // Treat it as a CanFight
        u(i); // Treat it as a CanSwim
        v(i); // Treat it as a CanFly
        w(i); // Treat it as an ActionCharacter
    }
} ///:~

```

Може да се види че **Hero** комбинира конкретния клас **ActionCharacter** с интерфейсите **CanFight**, **CanSwim** и **CanFly**. Като се комбинира конкретен клас с интерфейси по този начин, конкретният клас трябва да е пръв, после интерфейсите. (Иначе компилаторът дава грешка.)

Забележете, че сигнатурата на **fight()** е същата в **interface CanFight** и класа **ActionCharacter** и че **fight()** не е дефиниран в **Hero**. Правилото за **interface** е че може да наследявате от него (както ще видите след малко), но тогава получавате друг **interface**. Ако искате да създадете обект от нов тип той трябва да бъде клас с всички необходими дефиниции. Въпреки че **Hero** не дава експлицитна дефиниция за **fight()**, тя идва от **ActionCharacter** автоматично така че е възможно създаването на обекти от **Hero**.

В клас **Adventure** може да видите четири метода които взимат за аргументи различни интерфейси и конкретния клас. Когато се създава обект от **Hero** той може да се подаде на всеки от тези методи, което значи че ще се направи ъпкаст към всеки от интерфейсите **interface** по ред. Поради начина по който интерфейсите са проектирани в Java това работи без засечки и без никакво усилие от страна на програмиста.

Помнете че централната причина за съществуването на интерфейсите бе посочена в горния пример: да може да се прави ъпкаст към повече от един базов тип. Обаче втората причина да се използват интерфейси е същата като за **abstract** базови класове: да се предотврати възможността клиент-програмистът да направи обекти от този клас и да се посочи, че това е само интерфейс. Това довежда до въпроса: **interface** ли ще използвате или **abstract-ен** клас? **interface** дава ползите от **abstract** класа и ползите от **interface**, токо че ако е възможно да декларирате вашия клас без никакви методи и член-променливи винаги ще предпочитате **interface**-и пред **abstract**-ни класове. Фактически ако знаете че нещо ще става базов клас първо ще се опитате да го направите като **interface**, а само ако се наложи да имате дефиниции на методи и/или член-променливи ще го преработите в **abstract** клас.

Разширяване на интерфейс с наследяване

Лесно може да се добавят декларации на методи в **interface** чрез наследяване и също може да се комбинират няколко **interfaces** в нов **interface** с наследяване. В двата случая се получава нов **interface**, както в следния пример:

```
//: c07:HorrorShow.java
// Extending an interface with inheritance

interface Monster {
    void menace();
}

interface DangerousMonster extends Monster {
    void destroy();
}

interface Lethal {
    void kill();
}

class DragonZilla implements DangerousMonster {
    public void menace() {}
    public void destroy() {}
}

interface Vampire
    extends DangerousMonster, Lethal {
    void drinkBlood();
}

class HorrorShow {
    static void u(Monster b) { b.menace(); }
    static void v(DangerousMonster d) {
        d.menace();
        d.destroy();
    }
    public static void main(String[] args) {
        DragonZilla if2 = new DragonZilla();
        u(if2);
        v(if2);
    }
} ///:~
```

DangerousMonster е просто разширение на **Monster** квада дава нов **interface**. Това е реализирано в **DragonZilla**.

Синтаксисът използван във **Vampire** работи само когато се наследяват интерфейси. Нормално се използва **extends** със само един клас, но тъй като **interface** може да се направи от много други интерфейси, **extends** може да се отнася за няколко други интерфейси от които се прави нов **interface**. Както може да се види **interface** имената просто са разделени със запетай.

Групиране на константи

Понеже всички полета в **interface** са автоматично **static** и **final**, **interface**-ът е удобен инструмент за задаване на групи константи, много подобно на **enum** в С или C++. Например:

```
//: c07:Months.java
// Using interfaces to create groups of constants
package c07;

public interface Months {
    int
        JANUARY = 1, FEBRUARY = 2, MARCH = 3,
        APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
        AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
        NOVEMBER = 11, DECEMBER = 12;
} //:~
```

Забележете Java стила на използване само на големи букви (с подчертаващо тире за разделяне на отделните думи в един идентификатор) за **static final** примитивите, които имат статични инициализатори – тоест, за константите по време на компилация.

Полетата в един **interface** са автоматично **public**, така че не е необходимо специално да се указва това.

Сега може да използвате константите извън пакета им импортирайки **c07.*** или **c07.Months** както бихте импортирали всеки друг пакет и споменавайки стойностите с изрази като **Months.JANUARY**. Разбира се, получавате **int** така че нямате допълнителната сигурност на типа от C++ **enum**, но тази (често използвана) техника сигурно е подобрение спрямо твърдото записване на числа във вашите програми. (Което често се нарича “магически числа” и дава много труден за поддръжка код.)

Ако искате допълнителна сигурност за типа, може да направите класа така:¹

```
//: c07:Month2.java
// A more robust enumeration system
package c07;

public final class Month2 {
    private String name;
    private Month2(String nm) { name = nm; }
    public String toString() { return name; }
    public final static Month2
        JAN = new Month2("January"),
        FEB = new Month2("February"),
        MAR = new Month2("March"),
        APR = new Month2("April"),
        MAY = new Month2("May"),
        JUN = new Month2("June"),
        JUL = new Month2("July"),
        AUG = new Month2("August"),
        SEP = new Month2("September"),
        OCT = new Month2("October"),
        NOV = new Month2("November"),
        DEC = new Month2("December");
    public final static Month2[] month = {
        JAN, JAN, FEB, MAR, APR, MAY, JUN,
```

¹ This approach was inspired by an e-mail from Rich Hoffarth.

```

    JUL, AUG, SEP, OCT, NOV, DEC
};

public static void main(String[] args) {
    Month2 m = Month2.JAN;
    System.out.println(m);
    m = Month2.month(12);
    System.out.println(m);
    System.out.println(m == Month2.DEC);
    System.out.println(m.equals(Month2.DEC));
}
} //:~

```

Класът е наречен **Month2** понеже вече има **Month** в стандартната библиотека на Java. Той е **final** клас с **private** конструктор така че никой не може да наследява от него или да прави обекти. Единствените екземпляри са **final static** създадените в самия клас: **JAN, FEB, MAR** и т.н. Тези обекти са също използвани в масива **month**, което позволява да избирате месеците по номер вместо по име. (Забележете допълнителния **JAN** в масива за да се даде отместване единици, така че December да е 12-ти месец.) В **main()** може да видите сигурността на типа: **m** е **Month2** обект така че може да се присвои само на **Month2**. В предишния пример **Months.java** имаше само **int** стойности, така че на **int** променливата която щеше да представя месеците можеха да се дадат произволни стойности, което не бе много сигурно.

Този подход също позволява да се използва **==** или **equals()** взаимозаменямо, както е показано в края на **main()**.

Инициализиране на полета в интерфейсите

Полетата декларирани в интерфейси са автоматично **static** и **final**. Те не могат да бъдат "blank finals," но могат да се инициализират с неконстантни изрази. Например:

```

//: c07:RandVals.java
// Initializing interface fields with
// non-constant initializers
import java.util.*;

public interface RandVals {
    int rint = (int)(Math.random() * 10);
    long rlong = (long)(Math.random() * 10);
    float rfloat = (float)(Math.random() * 10);
    double rdouble = Math.random() * 10;
} //:~

```

Понеже полетата са **static**, те се инициализират когато класът се натоварва за пръв път, при първия достъп до кое да е поле. Ето прост тест:

```

//: c07:TestRandVals.java

public class TestRandVals {
    public static void main(String[] args) {
        System.out.println(RandVals.rint);
        System.out.println(RandVals.rlong);
        System.out.println(RandVals.rfloat);
        System.out.println(RandVals.rdouble);
    }
} //:~

```

Полетата, разбира се, не са част от интерфейса, а са запомнени в статичната памет отделена за този интерфейс.

Вътрешни класове

В Java 1.1 е възможно да се сложи дефиниция на клас в друга дефиниция на клас. Това се нарича *вътрешен клас*. Вътрешният клас е полезна черта, понеже позволява да се групират класовете които логически са обединени и да се управлява видимостта на единия в другия. Важно е да се разбере обаче, че вътрешните класове доста се отличават от композицията.

Често когато учите за тях нуждата от вътрешните класове не е непосредствено очевидна. В края на тази секция, след като се опише целия синтаксис и семантиката, ще намерите пример който изяснява ползите от вътрешните класове.

Създавате вътрешен клас точно както очаквате да се прави: чрез слагане на дефиниция на клас в класа, с който работите в момента: (Виж стр. 63 ако имате трудности с пускането на тази програма.)

```
//: c07(parcel1:Parcel1.java
// Creating inner classes
package c07.parcel1;

public class Parcel1 {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) {
            label = whereTo;
        }
        String readLabel() { return label; }
    }
    // Using inner classes looks just like
    // using any other class, within Parcel1:
    public void ship(String dest) {
        Contents c = new Contents();
        Destination d = new Destination(dest);
    }
    public static void main(String[] args) {
        Parcel1 p = new Parcel1();
        p.ship("Tanzania");
    }
} ///:~
```

Вътрешните класове, когато се използват вътре в **ship()**, точно както кои да са други класове. Единствената разлика на практика е че имената са вместени в **Parcel1**. Ще видите много понататък, че това не е единствената разлика.

По-типично е външният клас да има метод, който връща манипулатор към вътрешния клас, както тук:

```
//: c07(parcel2:Parcel2.java
// Returning a handle to an inner class
package c07.parcel2;
```

```

public class Parcel2 {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) {
            label = whereTo;
        }
        String readLabel() { return label; }
    }
    public Destination to(String s) {
        return new Destination(s);
    }
    public Contents cont() {
        return new Contents();
    }
    public void ship(String dest) {
        Contents c = cont();
        Destination d = to(dest);
    }
    public static void main(String[] args) {
        Parcel2 p = new Parcel2();
        p.ship("Tanzania");
        Parcel2 q = new Parcel2();
        // Defining handles to inner classes:
        Parcel2.Contents c = q.cont();
        Parcel2.Destination d = q.to("Borneo");
    }
} ///:~

```

Ако искате да направите обект от вътрешен клас където и да е освен в **не-static** метод на външния клас трябва да специфицирате името на метода като *OuterClassName.InnerClassName*, както се вижда в **main()**.

Вътрешни класове и ъпкастинг

До тук вътрешните класове не изглеждат много драматично. Най-после, ако искате скриване, Java вече има перфектен механизъм за скриване – само оставяте класа да бъде “приятелски” (видим само вътре в пакета) без да правите вътрешен клас.

Обаче вътрешните класове се изявяват ако започнете ъпкастинг към базови класове, в частност към **interface**. (Ефектът от произвеждане на интерфейсов манипулатор от обекта който го използва е в основата си същото като ъпкастинг към базов клас.) Така е защото вътрешният клас после може да бъде напълно невидим и недостъпен за който и да било, което е удобно за скриване на реализацията. Всичко което вземате обратно е манипулатор към базов клас или **interface** и е възможно даже да не можете да намерите точния тип, както е показано тук:

```

//: c07(parcel3:Parcel3.java
// Returning a handle to an inner class
package c07.parcel3;

abstract class Contents {
    abstract public int value();
}

```

```

interface Destination {
    String readLabel();
}

public class Parcel3 {
    private class PContents extends Contents {
        private int i = 11;
        public int value() { return i; }
    }
    protected class PDestination
        implements Destination {
        private String label;
        private PDestination(String whereTo) {
            label = whereTo;
        }
        public String readLabel() { return label; }
    }
    public Destination dest(String s) {
        return new PDestination(s);
    }
    public Contents cont() {
        return new PContents();
    }
}

class Test {
    public static void main(String[] args) {
        Parcel3 p = new Parcel3();
        Contents c = p.cont();
        Destination d = p.dest("Tanzania");
        // Illegal -- can't access private class:
        //!! Parcel3.PContents c = p.new PContents();
    }
} ///:~

```

Сега **Contents** и **Destination** представлят интерфейсите достъпни за клиент-програмиста. (**interface**, запомнете, автоматично прави всички свои членове **public**.) За удобство всичките са в един файл, но обикновено **Contents** и **Destination** биха били и двете **public** в техни собствени файлове.

Нещо ново е добавено в **Parcel3**: вътрешният клас **PContents** е **private** така че никой освен **Parcel3** няма достъп до него. **PDestination** е **protected**, така че никой освен **Parcel3**, класовете в **Parcel3** пакета (тъй като **protected** също дава пакетен достъп; тоест, **protected** е също "приятелски"), и наследниците на **Parcel3** няма достъп до **PDestination**. Това значи че клиент-програмистът има ограничено знание и достъп до тези членове. Фактически не можете даже да направите даункаст към **private** вътрешен клас (или **protected** вътрешен клас ако не сте наследник), понеже нямате достъп до името, както може да видите в **class Test**. Така **private** вътрешния клас дава начин да се избегнат всякакви зависимости от типа и напълно да се скрие реализацията. В добавка разширяването на **interface** е безполезно от гледна точка на приложния програмист, понеже той не може да получи достъп до никакъв друг метод освен тези които са част от публичния **interface** клас. Това също дава възможност на Java компилатора да произведе по-ефективен код.

Нормалните (невътрешни) класове не могат да бъдат направени **private** или **protected** – само **public** или "приятелски."

Забележете че **Contents** не е необходимо да е **abstract** клас. Бихте могли също да използвате и обикновен клас, но най-типичната начална точка за такъв дизайн е **interface**.

Вътрешни класове в методи и обхвати

Това които видяхте до тук може да служи за компас при използването на вътрешни класове. Изобщо кодът който ще пишете включвайки вътрешни класове ще бъде с "прости" вътрешни класове които са малки и лесни за разбиране. Обаче дизайнът на вътрешните класове е много завършен и ще иза множество други неща, по-скрити, пътища, които може да използвате, ако искате: вътрешни класове могат да бъдат създадени в методи и даже в произволен обхват. Има две причини да се прави това:

1. Както се показва преди, при реализацията на някакъв интерфейс за да може да се създаде и върне манипулатор.
2. Когато решавате сложен проблем и искате да създадете клас за да си помогнете, но не искате той да е достъпен за публиката.

В следващите примери предишният код ще се промени за да използва:

1. Клас дефиниран вътре в метод
2. Клас дефиниран в обхват вътре в метод
3. Анонимен клас реализиращ **interface**
4. Анонимен клас разширяващ клас който има конструктор не по подразбиране
5. Анонимен клас който изпълнява инициализация на полета
6. Анонимен клас който изпълнява конструиране чрез инициализация на екземпляра (анонимните вътрешни класове не могат да имат конструктори)

Това ще стане в пакета **innerscopes**. Първо общите интерфейси от предишния код ще се дефинират в техни собствени файлове за да се използват във всички примери:

```
//: c07:innerscopes:Destination.java
package c07.innerscopes;

interface Destination {
    String readLabel();
} ///:~
```

Беше изтъкнато че **Contents** би могъл да бъде **abstract** клас, така че тук той ще бъде в по-натурална форма, като **interface**:

```
//: c07:innerscopes:Contents.java
package c07.innerscopes;

interface Contents {
    int value();
} ///:~
```

Въпреки че е обикновен клас с реализация, **Wrapping** също се използва като общ "интерфейс" за извлечените от него класове:

```
//: c07:innerscopes:Wrapping.java
package c07.innerscopes;

public class Wrapping {
```

```

private int i;
public Wrapping(int x) { i = x; }
public int value() { return i; }
} //:~

```

Ще забележите по-горе че **Wrapping** има конструктор който изисква аргумент, за да станат малко по-интересни нещата.

Първият пример показва създаването на целия клас в обхвата на метод (вместо в обхвата на друг клас):

```

//: c07:innerscopes:Parcel4.java
// Nesting a class within a method
package c07.innerscopes;

public class Parcel4 {
    public Destination dest(String s) {
        class PDestination
            implements Destination {
            private String label;
            private PDestination(String whereTo) {
                label = whereTo;
            }
            public String readLabel() { return label; }
        }
        return new PDestination(s);
    }
    public static void main(String[] args) {
        Parcel4 p = new Parcel4();
        Destination d = p.dest("Tanzania");
    }
} //:~

```

Класът **PDestination** е част от **dest()** а не от **Parcel4**. (Също забележете че бихте могли да използвате **PDestination** за вътрешен клас на всеки клас от същата поддиректория без конфликт на имената.) Затова **PDestination** не може да бъде достъпен отвън на **dest()**. Забележете ъпкастинга който става в оператора за връщане – нищо не излиза от **dest()** освен манипулатор на базовия клас **Destination**. Разбира се фактът че името на класа **PDestination** е сложено вътре в **dest()** не значи че **PDestination** не е валиден обект когато **dest()** завърши.

Следващия пример позволява как може да вложите вътрешен клас в произволен обхват:

```

//: c07:innerscopes:Parcel5.java
// Nesting a class within a scope
package c07.innerscopes;

public class Parcel5 {
    private void internalTracking(boolean b) {
        if(b) {
            class TrackingSlip {
                private String id;
                TrackingSlip(String s) {
                    id = s;
                }
                String getSlip() { return id; }
            }
            TrackingSlip ts = new TrackingSlip("slip");
        }
    }
}

```

```

        String s = ts.getSlip();
    }
    // Can't use it here! Out of scope:
    // TrackingSlip ts = new TrackingSlip("x");
}
public void track() { internalTracking(true); }
public static void main(String[] args) {
    Parcel5 p = new Parcel5();
    p.track();
}
} //:~

```

Класът **TrackingSlip** е вмествен вътре в обхвата на **if** оператор. Това не значи че класът е третиран в зависимост от условие – той се компилира заедно с всичко друго. Обаче не е достъпен извън обхвата където е дефиниран. Във всичко друго е като всеки обикновен клас.

Следващият пример изглежда малко странно:

```

//: c07:innerscopes:Parcel6.java
// A method that returns an anonymous inner class
package c07.innerscopes;

public class Parcel6 {
    public Contents cont() {
        return new Contents() {
            private int i = 11;
            public int value() { return i; }
        }; // Semicolon required in this case
    }
    public static void main(String[] args) {
        Parcel6 p = new Parcel6();
        Contents c = p.cont();
    }
} //:~

```

Методът **cont()** комбинира създаването на връщана стойност с дефиниция на клас която представя върнатата стойност! В добавка класът е анонимен – няма име. За да станат нещата малко по-зле, изглежда като че сме започнали да създаваме **Contents** обект:

```

    return new Contents()

```

но тогава, преди да стигнете до точката със запетая, вие казвате, “Но чакай, мисля че влизам в дефиниция на клас”:

```

    return new Contents() {
        private int i = 11;
        public int value() { return i; }
    };

```

Този странен синтаксис значи “създай обект от анонимен клас който е наследен от **Contents**.“ Манипулаторът върнат от **new** е автоматично ъпкастнат към **Contents** манипулатор. Синтаксисът за анонимен вътрешен клас е съкратено записване на:

```

class MyContents extends Contents {
    private int i = 11;
    public int value() { return i; }
}
return new MyContents();

```

В анонимния вътрешен клас **Contents** е създаден чрез използване на конструктор по подразбиране. Следния пример показва какво да се прави ако базовият клас иска аргумент:

```
//: c07:innerscopes:Parcel7.java
// An anonymous inner class that calls the
// base-class constructor
package c07.innerscopes;

public class Parcel7 {
    public Wrapping wrap(int x) {
        // Base constructor call:
        return new Wrapping(x) {
            public int value() {
                return super.value() * 47;
            }
        }; // Semicolon required
    }
    public static void main(String[] args) {
        Parcel7 p = new Parcel7();
        Wrapping w = p.wrap(10);
    }
} ///:~
```

Тоест, приставате подходящ аргумент на конструктора на базовия клас, тук **x**-тът даден на **new Wrapping(x)**. Анонимният клас не може да има конструктор, където нормално се вика **super()**.

И в двата предишни примера точка-запетаята не означава края на тялото на класа (както прави в C++). Вместо това означава края на израза, който съдържа вътрешния клас. Това е идентично с използването на точка-запетаята навсякъде другаде.

Какво ще стане ако се опитате да инициализирате нещо в обект от анонимен вътрешен клас? Поради анонимността няма име, с което да се нарече конструктора, така че няма конструктор. Можете, обаче, да изпълните инициализация в точката на определяне на полетата:

```
//: c07:innerscopes:Parcel8.java
// An anonymous inner class that performs
// initialization. A briefer version
// of Parcel5.java.
package c07.innerscopes;

public class Parcel8 {
    // Argument must be final to use inside
    // anonymous inner class:
    public Destination dest(final String dest) {
        return new Destination() {
            private String label = dest;
            public String readLabel() { return label; }
        };
    }
    public static void main(String[] args) {
        Parcel8 p = new Parcel8();
        Destination d = p.dest("Tanzania");
    }
} ///:~
```

Ако при дефинирането на анонимен вътрешен клас пожелаете да използвате обект който е дефиниран извън анонимния вътрешен клас компилаторът изисква той да бъде **final**. Това е защото аргументът на **dest()** е **final**. Ако забравите ще получите грешка при компилация.

Доколкото просто присвоявате поле, горният подход е точен. Ами ако трябва да изпълните неща, които се правят в конструктори? В Java 1.1 с *инициализация на екземпляр* можете, фактически, да зъдъдете конструктор за анонимен вътрешен клас:

```
//: c07:innerscopes:Parcel9.java
// Using "instance initialization" to perform
// construction on an anonymous inner class
package c07.innerscopes;

public class Parcel9 {
    public Destination
        dest(final String dest, final float price) {
            return new Destination() {
                private int cost;
                // Instance initialization for each object:
                {
                    cost = Math.round(price);
                    if(cost > 100)
                        System.out.println("Over budget!");
                }
                private String label = dest;
                public String readLabel() { return label; }
            };
        }
    public static void main(String[] args) {
        Parcel9 p = new Parcel9();
        Destination d = p.dest("Tanzania", 101.395F);
    }
} ///:~
```

Вътре в инициализатора на екземпляра може да видите код, който не може да бъде изпълнен при инициализацията на полета (т.е. **if** оператора). Така че ефективно инициализатора на екземпляра е конструктора на анонимния вътрешен клас. Разбира се, той е ограничен; не може да претоварвате инициализаторите на екземпляра така че имате един конструктор от този вид.

Връзката към външния клас

До тук сякаш вътрешните класове са схема за скриване на имена и организация на код, която е полезна, но не неизбежна. Обаче има и друго нещо. Когато създавате вътрешен клас обектите от него имат връзка към външния клас който ги създава, а така те имат достъп до полетата на външния клас – без никакви специални квалификации. В добавка вътрешните класове имат право на достъп до всички елементи на външния клас.² Следващия пример демонстрира това:

```
//: c07:Sequence.java
// Holds a sequence of Objects

interface Selector {
    boolean end();
    Object current();
```

² This is very different from the design of *nested classes* in C++, which is simply a name-hiding mechanism. There is no link to an enclosing object and no implied permissions in C++.

```

void next();
}

public class Sequence {
    private Object[] o;
    private int next = 0;
    public Sequence(int size) {
        o = new Object[size];
    }
    public void add(Object x) {
        if(next < o.length) {
            o[next] = x;
            next++;
        }
    }
    private class SSelector implements Selector {
        int i = 0;
        public boolean end() {
            return i == o.length;
        }
        public Object current() {
            return o[i];
        }
        public void next() {
            if(i < o.length) i++;
        }
    }
    public Selector getSelector() {
        return new SSelector();
    }
    public static void main(String[] args) {
        Sequence s = new Sequence(10);
        for(int i = 0; i < 10; i++)
            s.add(Integer.toString(i));
        Selector sl = s.getSelector();
        while(!sl.end()) {
            System.out.println((String)sl.current());
            sl.next();
        }
    }
}
} //:~

```

Sequence е просто масив с фиксирана дължина от **Object** с клас който го обгръща. Викате **add()** за добавите нов **Object** в края на последователността (ако има място). За да се намери всеки обект в **Sequence** има интерфейс наречен **Selector**, който позволява да видите дали сте на **end()** (края-б.пр.), да видите **current()** (текущия-б.пр.) **Object** и да отидете на следващия **next()** **Object** в **Sequence** (последователността). Понеже **Selector** е **interface**, много други обекти може да приложат **interface** по техни си начини и много методи може да вземат **interface** като аргумент, с оглед да се осигури родов код.

Тук **SSelector** е частен клас който дава функционалността на **Selector**. В **main()**, може да се види създаването на **Sequence**, следвано от събиране на определен брой **String** обекти. Тогава се прави **Selector** с извикване на **getSelector()** и това се използва за да се движим през **Sequence** и избираме всеки елемент.

В началото **SSelector** изглежда като друг вътрешен клас. Разгледайте го обаче отблизо. Забележете че всеки от методите **end()**, **current()** и **next()** се отнася към **o**, който

манипулатор не е част от **SSelector**, а е **private** поле в обгръщащия клас. Обаче вътрешният метод има достъп до полетата на външния клас като че са негови. Излиза че това е много удобно, както личи в горния пример.

Така вътрешният клас има достъп до членовете на обгръщащия го клас. Как става това? Вътрешният клас трябва да помни връзка към породилия го външин клас. Тогава като споменете член на обгръщащия клас този (скрит) указател се използва за достъп до члена. За щастие компилаторът върши всичките подробности заради вас, но може сега да разберете, че вътрешен клас може да се създаде само асоцииран с обект от обгръщащия клас. Процесът на конструирането изисква инициализация на манипулатор към обгръщащия клас и компилаторът ще се оплаква, ако няма достъп до него. Повечето пъти всичко това става без намесата на програмиста.

static вътрешни класове

За да се разбере значението на **static** когато се приложи към вътрешен клас трябва да се припомни че вътрешният клас неявно разполага с указател към обгръщащия го клас. Това не е така, обаче, когато вътрешният клас е **static**. **static** вътрешен клас значи:

1. Не е необходимо да има създаден обект от обгръщащия клас за да се създаде обект от **static** вътрешен клас.
2. Не може да се ползва обгръщащия обект извънре на **static** вътрешен клас.

Има ограничения: **static** членовете могат да бъдат само във обгръщащия клас, така че вътрешният клас не може да има **static** данни или **static** вътрешни класове.

Ако не е необходимо да създавате обект от външния клас за да създавате обект от вътрешния клас, може всичко да направите **static**. За да направите това, трябва също и вътрешните класове да са **static**:

```
//: c07(parcel10:Parcel10.java
// Static inner classes
package c07.parcel10;

abstract class Contents {
    abstract public int value();
}

interface Destination {
    String readLabel();
}

public class Parcel10 {
    private static class PContents
        extends Contents {
            private int i = 11;
            public int value() { return i; }
    }
    protected static class PDestination
        implements Destination {
            private String label;
            private PDestination(String whereTo) {
                label = whereTo;
            }
            public String readLabel() { return label; }
        }
    public static Destination dest(String s) {
```

```

        return new PDestination(s);
    }
    public static Contents cont() {
        return new PContents();
    }
    public static void main(String[] args) {
        Contents c = cont();
        Destination d = dest("Tanzania");
    }
} //:~

```

В **main()** не е необходим обект от **Parcel10** заместо това използвате нормалния синтаксис за избор на **static** член за да извикате методите, които връщат манипулатори към **Contents** и **Destination**.

Нормално не може да слагате никакъв (изпълним-б.пр.) код в **interface**, но **static** вътрешен клас може да бъде част от **interface**. Тъй като класът е **static** това не нарушава правилата за интерфейсите – **static** вътрешния клас само се слага в пространството на имената на интерфейса:

```

//: c07:IInterface.java
// Static inner classes inside interfaces

interface IInterface {
    static class Inner {
        int i, j, k;
        public Inner() {}
        void f() {}
    }
} //:~

```

По-рано в книгата съветвах да се слага **main()** във всеки клас с цел тестване на класа. Един недостатък на това е допълнителният код, който трябва да се влачи. Ако това е проблем, може да използвате **static** вътрешен клас да съдържа вашия код:

```

//: c07: TestBed.java
// Putting test code in a static inner class

class TestBed {
    TestBed() {}
    void f() { System.out.println("f()"); }
    public static class Tester {
        public static void main(String[] args) {
            TestBed t = new TestBed();
            t.f();
        }
    }
} //:~

```

Това генерира отделен клас наречен **TestBed\$Tester** (за стартиране на програмата пишете **java TestBed\$Tester**). Може да използвате този клас за тестване, но не е необходимо да го включвате в крайната версия.

Обръщания към обект от външния клас

Ако е необходимо да произведете манипулятор към външния обект пишете името на външния клас следвано от точка и **this**. Например в класа **Sequence.Sselector** всеки може да направи и запомни манипулятор към външния клас **Sequence** като се напише **Sequence.this**. Това което

се получава е автоматично с точния тип. (Всичко е известно и проверено по време на компилация, така че няма допълнителни разходи по време на изпълнение.)

Понякога е нужно да се каже на обекти да създадат обекти от някой от техните вътрешни класове. За да се направи това е необходимо да се даде манипулятор към този външен клас на израза с **new**, подобно на това:

```
//: c07:parcel11.Parcel11.java
// Creating inner classes
package c07.parcel11;

public class Parcel11 {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) {
            label = whereTo;
        }
        String readLabel() { return label; }
    }
    public static void main(String[] args) {
        Parcel11 p = new Parcel11();
        // Must use instance of outer class
        // to create an instances of the inner class:
        Parcel11.Contents c = p.new Contents();
        Parcel11.Destination d =
            p.new Destination("Tanzania");
    }
} ///:~
```

За да се създаде обект от вътрешния клас направо не се следва същата форма да се обръщаме към името на външния клас **Parcel11** както може да се очаква, а вместо това се използва обект от външния клас за да се направи обект от вътрешния клас:

```
Parcel11.Contents c = p.new Contents();
```

И така, не е възможно да се създаде обект от вътрешен клас докато не се създаде обект от външния клас. Това е защото вътрешният клас е твърде свързан с външния клас, който го е създал. Обаче ако направите **static** вътрешен клас не е необходимо да има указател към външен обект.

Наследяване от вътрешни класове

Понеже конструкторът на вътрешният клас трява да използва манипулятор към външния клас, нещата са малко по-усложнени ако наследявате от вътрешен клас. Проблемът е в "тайния" манипулятор към обгръщащия обект който трябва да бъде инициализиран и вече в извлечения клас не остава манипулятор към външен клас. Решението е да се използва синтаксис който да прави връзката явна:

```
//: c07:InheritInnner.java
// Inheriting an inner class

class WithInnner {
    class Inner {}
}
```

```

public class InheritInnner
    extends WithInnner.Innner {
//! InheritInnner() {} // Won't compile
InheritInnner(WithInnner wi) {
    wi.super();
}
public static void main(String[] args) {
    WithInnner wi = new WithInnner();
    InheritInnner ii = new InheritInnner(wi);
}
} ///:~

```

Може да се види, че **InheritInnner** разширява само вътрешния клас, не и външния. Но когато се дойде до изпълнението на конструктор, този по подразбиране не става и не може просто да дадете манипулятор на външния. Освен това трябва да се използва синтаксисът:

```
| enclosingClassHandle.super();
```

вътре в конструктора. Това дава необходимия манипулятор и тогава програмата се компилира.

Могат ли вътрешни класове да се подтискат?

Какво става, ако създадете вътрешен клас, наследите от външния клас и предефинирате вътрешния клас? Тоест, възможно ли е да се подтисне вътрешен клас? Това изглежда като мощна концепция, но "подтискането" на вътрешен клас като друг метод на външния клас не прави нищо:

```

//: c07:BigEgg.java
// An inner class cannot be overriden
// like a method

class Egg {
    protected class Yolk {
        public Yolk() {
            System.out.println("Egg.Yolk()");
        }
    }
    private Yolk y;
    public Egg() {
        System.out.println("New Egg()");
        y = new Yolk();
    }
}

public class BigEgg extends Egg {
    public class Yolk {
        public Yolk() {
            System.out.println("BigEgg.Yolk()");
        }
    }
    public static void main(String[] args) {
        new BigEgg();
    }
} ///:~

```

Конструкторът по подразбиране се прави автоматично от компилатора и вика конструктора на базовия клас. Може да се помисли че след като **BigEgg** се създава "подтиснатата" версия на **Yolk** би се използвала, но това не е така. Изходът е:

```
New Egg()
Egg.Yolk()
```

Този пример показва, че не става никаква допълнителна магия, ако наследите външния клас. Обаче е възможно явно да наследите от вътрешния клас:

```
//: c07:BigEgg2.java
// Proper inheritance of an inner class

class Egg2 {
    protected class Yolk {
        public Yolk() {
            System.out.println("Egg2.Yolk()");
        }
        public void f() {
            System.out.println("Egg2.Yolk.f()");
        }
    }
    private Yolk y = new Yolk();
    public Egg2() {
        System.out.println("New Egg2()");
    }
    public void insertYolk(Yolk yy) { y = yy; }
    public void g() { y.f(); }
}

public class BigEgg2 extends Egg2 {
    public class Yolk extends Egg2.Yolk {
        public Yolk() {
            System.out.println("BigEgg2.Yolk()");
        }
        public void f() {
            System.out.println("BigEgg2.Yolk.f()");
        }
    }
    public BigEgg2() { insertYolk(new Yolk()); }
    public static void main(String[] args) {
        Egg2 e2 = new BigEgg2();
        e2.g();
    }
} //:~
```

Сега **BigEgg2.Yolk** явно **extends Egg2.Yolk** и подтиска методите му. Методът **insertYolk()** позволява **BigEgg2** да направи ъпкаст на един от неговите **Yolk** обекти към **y** манипулатора в **Egg2**, така че когато **g()** извика **y.f()** подтиснатата версия на **f()** се използва. Изходът е:

```
Egg2.Yolk()
New Egg2()
Egg2.Yolk()
BigEgg2.Yolk()
BigEgg2.Yolk.f()
```

Второто извикване на `Egg2.Yolk()` е извикването в конструктора на базовия клас на `BigEgg2.Yolk` конструктора. Може да видите, че подтиснатата версия на `f()` се използва когато се вика `g()`.

Идентификатори на вътрешния клас

Тъй като всеки клас произвежда `.class` файл който съдържа всичката информация как да се създаде обект от този тип (тази информация произвежда мета-клас наречен **Class** обект), може да познаете че вътрешните класове трябва също да създават `.class` файлове да съдържат информация за **техните Class** обекти. Имената на тези файлове/класове имат строга формула: името на обхващащия клас, следвано от '`$`', следван от името на вътрешния клас. Например `.class` файловете създавани от `InheritInnner.java` включват:

```
| InheritInnner.class  
| WithInnner$Innner.class  
| WithInnner.class
```

Ако вътрешните класове са анонимни компилаторът започва просто да генерира числа за техни идентификатори. Ако вътрешни класове са вместени във вътрешни класове, техните имена са просто добавени след '`$`' и идентификатор(ите) на външни класове).

Въпреки че тази схема на генерация на имена е прости и праволинейна, тя също е добра и устиява в много ситуации.³ Понеже това е стандартна схема за имена в Java, генерираните файлове са автоматично независими от платформата. (Забележете че Java компилаторът променя вътрешните класове по всички начини, щото те да работят.)

Защо вътрешни класове: рамки на управлението

До тук видяхте много синтаксис и семантика, обясняващи как работят вътрешните класове, но това не отговаря на въпроса защо съществуват те. Защо Sun си създаде толкова безпокойства вмъквайки толкова фундаментална черта на езика в Java 1.1? Отговорът е нещо, което аз ще споменавам като *control framework*.

Application framework е клас или множество от класове, проектиран(о) да се решава определен проблем. За да се приложи приложната рамка се наследяват един или няколко класа и се подтискат колкото е нужно на брой методи. Кодът който се пише за подтиснатите методи променя поведението с цел да се реши точно необходимия (по-друг от първоначалния - б.пр.) проблем. Рамката за управление е частен случай на приложна рамка, проектирана с цел да се реагира на събития; система, която в поведението си се ръководи главно от събития се нарича *задвижвана от събития система*. Един от най-важните проблеми в потребителското програмиране е графичният потребителски интерфейс (GUI), който почти изцяло е задвижван от събития. Както ще видите в глава 13, Swing библиотеката на Java е управляваща рамка, която елегантно решава GUI проблема използвайки вътрешни класове.

За да видим как вътрешните класове позволяват лесно създаване и използване на управляващи рамки да вземем една, чиято задача е да изпълнява нещо, когато се случи събитието "готовност." Макар че "готовност" би могло да значи много неща, в нашия случай то се определя по часовника. Ова което се получава е управляваща рамка, която не е определено какво управлява (и би могла да управлява всичко-б.пр.). Първо, има интерфейс който описва всяко възможно управляващо събитие. Той е **abstract** клас наместо истински

³ On the other hand, '`$`' is a meta-character to the Unix shell and so you'll sometimes have trouble when listing the `.class` files. This is a bit strange coming from Sun, a Unix-based company. My guess is that they weren't considering this issue, but instead thought you'd naturally focus on the source-code files.

interface понеже поведението по подразбиране е управление по часовник, така че част от реализацията може да бъде включена тук:

```
//: c07:controller:Event.java
// The common methods for any control event
package c07.controller;

abstract public class Event {
    private long evtTime;
    public Event(long eventTime) {
        evtTime = eventTime;
    }
    public boolean ready() {
        return System.currentTimeMillis() >= evtTime;
    }
    abstract public void action();
    abstract public String description();
} //:~
```

Конструкторът просто прихваща времето когато искате да стартира **Event**, докато **ready()** казва когато стане време да се пуска. Разбира се, **ready()** би могло да бъде подтиснато в извлечен клас за да може действието на **Event** да се основе на нещо друго, различно от времето.

action() е методът, който се вика когато **Event** е **ready()** и **description()** дава текстова информация за **Event**.

Следващия файл съдържа фактическата управляваща рамка, която управлява и пуска събитията. Първият клас фактически е "помощен" чиято задача е да съдържа **Event** обекти. Би могъл да се замести с която и да е подходяща колекция и в глава 8 ще откриете такива, които правят трика без писането на този допълнителен код:

```
//: c07:controller:Controller.java
// Along with Event, the generic
// framework for all control systems:
package c07.controller;

// This is just a way to hold Event objects.
class EventSet {
    private Event[] events = new Event[100];
    private int index = 0;
    private int next = 0;
    public void add(Event e) {
        if(index >= events.length)
            return; // (In real life, throw exception)
        events[index++] = e;
    }
    public Event getNext() {
        boolean looped = false;
        int start = next;
        do {
            next = (next + 1) % events.length;
            // See if it has looped to the beginning:
            if(start == next) looped = true;
            // If it loops past start, the list
            // is empty:
            if((next == (start + 1) % events.length)
                && looped)
```

```

        return null;
    } while(events(next) == null);
    return events(next);
}
public void removeCurrent() {
    events(next) = null;
}
}

public class Controller {
    private EventSet es = new EventSet();
    public void addEvent(Event c) { es.add(c); }
    public void run() {
        Event e;
        while((e = es.getNext()) != null) {
            if(e.ready()) {
                e.action();
                System.out.println(e.description());
                es.removeCurrent();
            }
        }
    }
}
} //://:~
```

EventSet съдържа до 100 **Events**. (В “реална” колекция от глава 8 нямаше да се главоболите с максималната дължина, понеже те си я сменят сами). **index** се използва за пазене сведения за следващото свободно място, **next** се използва при търсене на следващия **Event** в списъка за да се види какво ще се прави по-нататък. Това е важно през време на извикването на **getNext()**, понеже **Event** обектите се махат от листа (чрез **removeCurrent()**) след като са стартирани, така че **getNext()** ще намери дупки в списъка като се движи през него.

Забележете че **removeCurrent()** не просто слага някакъв флаг за да отбележи че обектите са използвани. Вместо това слага манипулятор да бъде **null**. Това е важно, понеже ако боклучарят види че обектът се използва той няма да го почисти. Ако очаквате вашите манипулятори да станат излишни (както тук), добре е да ги приравните на **null** когато вече не трябват за да дадете възможност на боклучаря да ги чисти.

Controller е мястото, където става истинската работа. Там се използва **EventSet** за владеене на неговите **Event** обекти, а **addEvent()** позволява да се добавят събития в този списък. Но важен метод е **run()**. Този метод цели премахването на **Event** обекти, които са изпълнени. За всеки за който намери **ready()** вика **action()** метода, извежда **description()** и после маха **Event** от списъка.

Забележете че до този момент не се знае какво точно прави **Event**. И това е гвоздеят на програмата; как се “отделят нещата които се променят от тези, които остават така.” Или, използвайки моя термин, “векторът на промяната” са различните действия свързани с различните **Event** обекти, а различните действия се изразяват чрез създаване на различни подкласове на **Event**.

Тук е мястото, където вътрешните класове влизат в играта. Те позволяват две неща:

1. Да се изрази цялата реализация на рамката в единствен клас, капсулирайки с това всичко, което е уникално за реализацията. Вътрешните класове се използват за да се изразят многото различни видове **action()** необходими да се реши проблема. В добавка следващия пример използва **private** вътрешни класове така че реализацията е напълно скрита и може да се променя безнаказано.

- Чрез вътрешните класове се избегва реализацията да стане тромава, понеже лесно има достъп до всеки член на външния клас. Без тази възможност кодът може да стане толкова неприятен, че да се видите принудени да търсите друг начин.

Да вземем конкретна рамка проектирана да работи с оранжерия.⁴ Всяка дейност е напълно различна: светлини, вода, включване и изключване на термостати, зумери и рестартиране на системата. Но управляващата рамка е проектирана лесно да изолира този тъй различен код. За всеки тип дейност се наследява нов **Event** вътрешен клас и се пише управляващ код в **action()**.

Както е типично за приложна рамка **GreenhouseControls** е наследен от **Controller**:

```
//: c07:controller:GreenhouseControls.java
// This produces a specific application of the
// control system, all in a single class. Inner
// classes allow you to encapsulate different
// functionality for each type of event.
package c07.controller;

public class GreenhouseControls
    extends Controller {
    private boolean light = false;
    private boolean water = false;
    private String thermostat = "Day";
    private class LightOn extends Event {
        public LightOn(long eventTime) {
            super(eventTime);
        }
        public void action() {
            // Put hardware control code here to
            // physically turn on the light.
            light = true;
        }
        public String description() {
            return "Light is on";
        }
    }
    private class LightOff extends Event {
        public LightOff(long eventTime) {
            super(eventTime);
        }
        public void action() {
            // Put hardware control code here to
            // physically turn off the light.
            light = false;
        }
        public String description() {
            return "Light is off";
        }
    }
    private class WaterOn extends Event {
        public WaterOn(long eventTime) {
            super(eventTime);
        }
        public void action() {
```

⁴ For some reason this has always been a pleasing problem for me to solve; it came from *C++ Inside & Out*, but Java allows a much more elegant solution.

```

// Put hardware control code here
water = true;
}
public String description() {
    return "Greenhouse water is on";
}
}
private class WaterOff extends Event {
    public WaterOff(long eventTime) {
        super(eventTime);
    }
    public void action() {
        // Put hardware control code here
        water = false;
    }
    public String description() {
        return "Greenhouse water is off";
    }
}
private class ThermostatNight extends Event {
    public ThermostatNight(long eventTime) {
        super(eventTime);
    }
    public void action() {
        // Put hardware control code here
        thermostat = "Night";
    }
    public String description() {
        return "Thermostat on night setting";
    }
}
private class ThermostatDay extends Event {
    public ThermostatDay(long eventTime) {
        super(eventTime);
    }
    public void action() {
        // Put hardware control code here
        thermostat = "Day";
    }
    public String description() {
        return "Thermostat on day setting";
    }
}
// An example of an action() that inserts a
// new one of itself into the event list:
private int rings;
private class Bell extends Event {
    public Bell(long eventTime) {
        super(eventTime);
    }
    public void action() {
        // Ring bell every 2 seconds, rings times:
        System.out.println("Bing!");
        if(--rings > 0)
            addEvent(new Bell(
                System.currentTimeMillis() + 2000));
    }
}

```

```

public String description() {
    return "Ring bell";
}
}

private class Restart extends Event {
    public Restart(long eventTime) {
        super(eventTime);
    }
    public void action() {
        long tm = System.currentTimeMillis();
        // Instead of hard-wiring, you could parse
        // configuration information from a text
        // file here:
        rings = 5;
        addEvent(new ThermostatNight(tm));
        addEvent(new LightOn(tm + 1000));
        addEvent(new LightOff(tm + 2000));
        addEvent(new WaterOn(tm + 3000));
        addEvent(new WaterOff(tm + 8000));
        addEvent(new Bell(tm + 9000));
        addEvent(new ThermostatDay(tm + 10000));
        // Can even add a Restart object!
        addEvent(new Restart(tm + 20000));
    }
    public String description() {
        return "Restarting system";
    }
}

public static void main(String[] args) {
    GreenhouseControls gc =
        new GreenhouseControls();
    long tm = System.currentTimeMillis();
    gc.addEvent(gc.new Restart(tm));
    gc.run();
}
} ///:~

```

Забележете че **light**, **water**, **thermostat** и **rings** всичките принадлежат на външния клас **GreenhouseControls**, а вътрешните класове нямат проблеми с достъпа до тези полета. Също, повечето от **action()** методите също включват някакво управление на хардуер, което най-вероятно ще въвлече изпълнението на не-Java код.

Повечето от **Event** класовете изглеждат подобни, но **Bell** и **Restart** са специални. **Bell** бие (включва зумер, камбана или нещо такова - б.пр.) и ако не е било достатъчно, добавя още един **Bell** обект към списъка на събитията, така че ще бие пак по-късно. Забележете как вътрешните класове почти изглеждат като множествено наследяване: **Bell** има методите на **Event** и също има всичките методи на външния клас **GreenhouseControls**.

Restart е отговорен за стартирането на системата, затова слага всички необходими събития. Разбира се, по-гъвкав начин да се реализиратова е да се избегне твърдото кодиране и те да се четат от файл. (Едно упражнение в глава 10 иска да модифицирате този код за да се постигне това.) Тий като **Restart()** е просто друг **Event** обект може просто да се добави **Restart** обект в **Restart.action()** така че системата редовно да се рестартира. И всичко каквото трябва да се направи в **main()** е да се създаде **GreenhouseControls** обект и да се добави **Restart** обект за пускането му.

Този пример трябва да издигне много в очите ви значението на вътрешните обекти, особено като са използвани в управляваща рамка. Обаче в последната част на глава 13 ще видите как елегантно те се използват за описание на графичен интерфейс. Като свършите въпросната секция ще бъдете напълно убедени.

Конструктори и полиморфизъм

Обикновено с конструкторите е по-различно от другите методи. Това е така и в случая на полиморфизъм. Макар че конструкторите не са полиморфни (въпреки че може да има нещо като "виртуален конструктор," както ще видите в глава 11), важно е да се разбере как действат конструкторите в големи йерархии и при полиморфизъм. Това разбиране ще ви предпази от навлизане в неприятни положения.

Ред на извикване на конструкторите

Редът на извикване на конструкторите беше накратко изложен в глава 4, но това беше преди да въведем наследяването и полиморфизма.

В конструктора на извлечения клас винаги се вика конструктор на базов клас, докато всичките конструктори на базови класове са извикани. Това има смисъл, понеже конструкторът има специална задача: да гледа дали обектът е построен правилно. Извлеченият клас има достъп само до собствените си членове, той няма достъп до тези на базовия клас (чиито членове типично са **private**). Само конструкторът на базовия клас има правата и достатъчно знания за да може да инициализира членовете на базовия клас. Така че е важно да се извикат всички конструктори, иначе обектът няма да се конструира правилно. Това е причината компилаторът да заставя да има извикване на конструктор за всяка порция на извлечения клас. Той тихичко ще извика конструктор по позразбиране ако не сте задали конструктор в тялото на класа. Ако няма конструктор по подразбиране, компилаторът ще се оплаква. (В случая когато класът няма конструктори компилаторът автоматично синтезира конструктор по подразбиране.)

Нека да разгледаме пример, който демонстрира ефектите от композицията, наследяването и полиморфизма върху реда на извикване на конструкторите:

```
//: c07:Sandwich.java
// Order of constructor calls

class Meal {
    Meal() { System.out.println("Meal()"); }
}

class Bread {
    Bread() { System.out.println("Bread()"); }
}

class Cheese {
    Cheese() { System.out.println("Cheese()"); }
}

class Lettuce {
    Lettuce() { System.out.println("Lettuce()"); }
}

class Lunch extends Meal {
    Lunch() { System.out.println("Lunch()"); }
}
```

```

class PortableLunch extends Lunch {
    PortableLunch() {
        System.out.println("PortableLunch()");
    }
}

class Sandwich extends PortableLunch {
    Bread b = new Bread();
    Cheese c = new Cheese();
    Lettuce l = new Lettuce();
    Sandwich() {
        System.out.println("Sandwich()");
    }
    public static void main(String[] args) {
        new Sandwich();
    }
} //:~

```

Този пример създава сложен клас от други класове, всеки клас има конструктор който се обявява сам. Важният клас е **Sandwich**, който отразява три нива на наследяване (четири, ако смятате и неявното от **Object**) и три член-обекти. Когато е създаден **Sandwich** обект в **main()**, изходът е:

```

Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Sandwich()

```

Това показва че редът на извикване на конструкторите в сложен обект е:

1. Вика се конструкторът на базовия клас. Тази стъпка се повтаря рекурсивно така, че конструкторът на кореновия клас се вика първо, после този на първия извлечен клас и т.н., докато се достигне последния извлечен клас.
2. Инициализаторите на членовете се викат по реда на декларирането.
3. Тялото на конструктора на извлечения клас се вика.

Редът на извикване на конструкторите е важен. Огато наследявате, знаете всичко за базовия клас и имате достъп до всички **public** и **protected** членове на базовия клас. Това значи, че трябва със сигурност всички членове на базовия клас да съществуват и да са инициализирани когато сте в извлечения клас. В нормален метод конструкцията вече е станала, така че всички членове от всички части на обекта са построени. В конструктора, обаче, трябва да е възможнък до сте сигурни, че всички членове, които ви трябват, ще са построени. Единствения начин да стане това е като се вика първо конструктора на базовия клас. Ака когато сте в конструктора на извлечения клас всичките членове, които може да използвате от базовия клас са вече инициализирани. "Знанието че всички членове са валидни" вътре в конструктора е също причината че, навсякъде където е възможно, ще инициализирате всички член-обекти (т.е. обекти сложени в класа чрез композиция) в точката на дефинирането им в класа (като **b**, **c**, и **l** в горния пример). Ако следвате тази практика, ще помогнете да се осигури всички членове на базовия клас и член-обектите на текущия обект да са инициализирани. За нещастие това не може да стане във всички случаи, както ще видите в следващата секция.

Наследяването и `finalize()`

Когато използвате композиция за създаване на нов клас никога не се грижите за финализирането на обекти от този клас. Всеки член е независим обект и като такъв се обработва от боклучаря и се финализира без значение дали се е случило да е член на вашия клас. С наследяването, обаче, трябва да се наследи `finalize()` в извлечения клас ако има специално почистване да се върши към събирането на боклука. Когато подтиснете `finalize()` в наследения клас, важно е да се помни да се извика версията от базовия клас на `finalize()`, иначе няма да стане финализацията на базовия клас. Следващия пример показва нещата:

```
//: c07:Frog.java
// Testing finalize with inheritance

class DoBaseFinalization {
    public static boolean flag = false;
}

class Characteristic {
    String s;
    Characteristic(String c) {
        s = c;
        System.out.println(
            "Creating Characteristic " + s);
    }
    protected void finalize() {
        System.out.println(
            "finalizing Characteristic " + s);
    }
}

class LivingCreature {
    Characteristic p =
        new Characteristic("is alive");
    LivingCreature() {
        System.out.println("LivingCreature()");
    }
    protected void finalize() {
        System.out.println(
            "LivingCreature finalize");
        // Call base-class version LAST!
        if(DoBaseFinalization.flag)
            try {
                super.finalize();
            } catch(Throwable t) {}
    }
}

class Animal extends LivingCreature {
    Characteristic p =
        new Characteristic("has heart");
    Animal() {
        System.out.println("Animal()");
    }
    protected void finalize() {
        System.out.println("Animal finalize");
        if(DoBaseFinalization.flag)
```

```

    try {
        super.finalize();
    } catch(Throwable t) {}
}

class Amphibian extends Animal {
    Characteristic p =
        new Characteristic("can live in water");
    Amphibian() {
        System.out.println("Amphibian()");
    }
    protected void finalize() {
        System.out.println("Amphibian finalize");
        if(DoBaseFinalization.flag)
            try {
                super.finalize();
            } catch(Throwable t) {}
    }
}

public class Frog extends Amphibian {
    Frog() {
        System.out.println("Frog()");
    }
    protected void finalize() {
        System.out.println("Frog finalize");
        if(DoBaseFinalization.flag)
            try {
                super.finalize();
            } catch(Throwable t) {}
    }
    public static void main(String[] args) {
        if(args.length != 0 &&
           args[0].equals("finalize"))
            DoBaseFinalization.flag = true;
        else
            System.out.println("not finalizing bases");
        new Frog(); // Instantly becomes garbage
        System.out.println("bye!");
        // Must do this to guarantee that all
        // finalizers will be called:
        System.runFinalizersOnExit(true);
    }
} ///:~

```

Класът **DoBaseFinalization** просто съдържа флаг който казва на всеки клас от йерархията кога да вика **super.finalize()**. Този флаг се слага според командния ред, така че може да видите поведението с и без финализация на базовия клас.

Всеки клас в йерархията също съдържа член-обект от тип **Characteristic**. Ще видите, че независимо как се викат финализаторите на базовия клас, **Characteristic** член-обектите винаги се финализират.

Всеки подтиснат **finalize()** трябва да има достъп най-малко до **protected** членовете понеже **finalize()** методът в класа **Object** е **protected** и компилаторът няма да позволи да се намали достъпът при наследяването. ("Приятелски" е с по-малък достъп от **protected**.)

В `Frog.main()` флагът `DoBaseFinalization` е конфигуриран и единствен `Frog` обект се създава. Помнете, че събирането на боклука и в частност финализацията биха могли и да не се случат, така че за да се направи да станат `System.runFinalizersOnExit(true)` добавя допълнителна работа във връзка с гарантирането на това. Без финализацията на базовия клас изходът е:

```
not finalizing bases
Creating Characteristic is alive
LivingCreature()
Creating Characteristic has heart
Animal()
Creating Characteristic can live in water
Amphibian()
Frog()
bye!
Frog finalize
finalizing Characteristic is alive
finalizing Characteristic has heart
finalizing Characteristic can live in water
```

Може да видите че, разбира се, не се викат финализатори на базовите класове на `Frog`. Но ако добавите "finalize" на командния ред получавате:

```
Creating Characteristic is alive
LivingCreature()
Creating Characteristic has heart
Animal()
Creating Characteristic can live in water
Amphibian()
Frog()
bye!
Frog finalize
Amphibian finalize
Animal finalize
LivingCreature finalize
finalizing Characteristic is alive
finalizing Characteristic has heart
finalizing Characteristic can live in water
```

Макар и редът в който обектите да се финализират да е същия в който се създават, технически редът на финализация не се задава. С базовите класове, обаче, имате пълен контрол върху реда на финализацията. Най-добрият ред е показаният тук, което е обрънатия ред на инициализацията. Следвайки формата използвана за деструкторите в C++ първо ще финализирате извлечения клас, после базовия. Това е защото във финализацията на извлечения клас би могло да се викат методи от базовия, които трябва да са още живи за това и не бива да се извършва финализация на базовия клас предварително.

Поведение на полиморфни методи в конструкторите

Иерархията на извикване на конструкторите изважда наяве интересна дилема. Какво става, ако сте в конструктора и извикате динамично свързан метод на конструирания в момента обект? В обикновен обект може да си въобразим каквото става – динамично свързаното извикване се решава по време на изпълнение, понеже извикващия обек не може да знае дали методът е негов или на извлечен клас. Може за смисленост да се счита, че така става и в случая на конструктор.

Това обаче не е точно. Ако викате динамично свързан метод вътре в конструктор, подтиснатата дефиниция на метода се използва. Обаче ефектът може да бъде неочекван и може да докара трудни за улавяне грешки.

Концептуално задачата на конструктора е да докара обекта в живота (което едва ли е тривиално като постижение). През време на работа на конструктора целия обект може да е само частично построен – знае се само, че базовият клас е бил инициализиран, но не може да се знае кои класове са наследени. Динамичното свързване на метод обаче, търси “напред” или “навън” в йерархията на наследяването. То вика метод в извлечен клас. Ако превите това вътре в конструктор, викате член който манипулира членове които може още да не са инициализирани – сигурна рецепта за катастрофа.

Може да видите проблема в следния пример:

```
//: c07:PolyConstructors.java
// Constructors and polymorphism
// don't produce what you might expect.

abstract class Glyph {
    abstract void draw();
    Glyph() {
        System.out.println("Glyph() before draw()");
        draw();
        System.out.println("Glyph() after draw()");
    }
}

class RoundGlyph extends Glyph {
    int radius = 1;
    RoundGlyph(int r) {
        radius = r;
        System.out.println(
            "RoundGlyph.RoundGlyph(), radius = "
            + radius);
    }
    void draw() {
        System.out.println(
            "RoundGlyph.draw(), radius = " + radius);
    }
}

public class PolyConstructors {
    public static void main(String[] args) {
        new RoundGlyph(5);
    }
} ///:~
```

В **Glyph** методът **draw()** е **abstract**, така че е проектиран да бъде подтиснат. Разбира се ние сме принудени да го подтиснем в **RoundGlyph**. Но конструктора на **Glyph** вика този метод, извикването свършва в **RoundGlyph.draw()**, което изглежда да е желаното. Но погледнете изхода:

```
Glyph() before draw()
RoundGlyph.draw(), radius = 0
Glyph() after draw()
RoundGlyph.RoundGlyph(), radius = 5
```

Когато конструктора на **Glyph** вика **draw()**, стойността на **radius** даже не е началната стойност по подразбиране 1. Тя е нула. Това вероятно ще причини точка или нищо да се изобрази на екрана, а вие ще се чуците защо програмата не работи.

Редът на инициализация описан в предната секция не е съвсем пълен, а точно липсващата част е ключът към мистерията. Фактическият процес на инициализация е:

1. Паметта алокирана за обект се инициализира с нула преди да се направи каквото и да е друго.
2. Викат се конструкторите на базови класове както бе описано преди. В тази точка подтиснатия **draw()** метод се вика, (да, преди конструкторът на **RoundGlyph** да се извика), като стойността на **radius** е нула, поради т.1.
3. Инициализаторите на членове се викат по реда на декларирането.
4. Тялото на конструктора на извлечения клас се вика.

По-добре е всичко да се инициализира с нула (или каквото нулата значи за конкретния вид данни) отколкото да е просто боклук. Това включва манипуляторите на обекти които са вградени в обект чрез композиция. Така че ако забравите да инициализирате такъв манипулятор щяхте да получите изключение по време на изпълнение. Всичко обаче става нула, която обикновено е издайническа стойност, видяна в изход от програма.

От друга страна, трябва множко да се уплашите от изхода на тази програма. Направихте логични предположения и все пак програмата е мистериозно неверна, без оплаквания от компилатора. (C++ има по-рационално поведение в такива ситуации.) Бъгове от този род лесно могат да бъдат погребани и да отнемат много време за откриването им.

Като резултат, ето добро правило за конструкторите: “Прави минималното за да стане обекта и ако е възможно, избягвай викането на методи.” Единствените безопасни за викане в конструктори методи са тези които са **final** в базовия клас. (Това също се отнася за **private** които са автоматично **final**.) Те не могат да бъдат подтиснати и не могат да донесат такива изненади.

Проектиране с наследяване

Като научи за полиморфизма на човек може да му се стори че всичко трябва да става с наследяване щом полиморфизмът е толкова умен инструмент. Това може да утежни проектите; фактически ако направо изберете наследяването за начин от един клас да се получи друг клас може да стане ненужно сложно.

По-добрият подход е да се използва композиция първо когато не е ясно какво трябва да се използва. Композицията не вкарва проекта в йерархии на наследяване. Но композицията е по-пъкава понеже позволява динамично да се избира типа (и чрез това-поведението) когато се използва композиция, докато наследяването изисква точният тип да е известен по време на компилацията. Следващия пример илюстрира това:

```
//: c07:Transmogrify.java
// Dynamically changing the behavior of
// an object via composition.

interface Actor {
    void act();
}

class HappyActor implements Actor {
    public void act() {
```

```

        System.out.println("HappyActor");
    }
}

class SadActor implements Actor {
    public void act() {
        System.out.println("SadActor");
    }
}

class Stage {
    Actor a = new HappyActor();
    void change() { a = new SadActor(); }
    void go() { a.act(); }
}

public class Transmogrify {
    public static void main(String[] args) {
        Stage s = new Stage();
        s.go(); // Prints "HappyActor"
        s.change();
        s.go(); // Prints "SadActor"
    }
} //:~

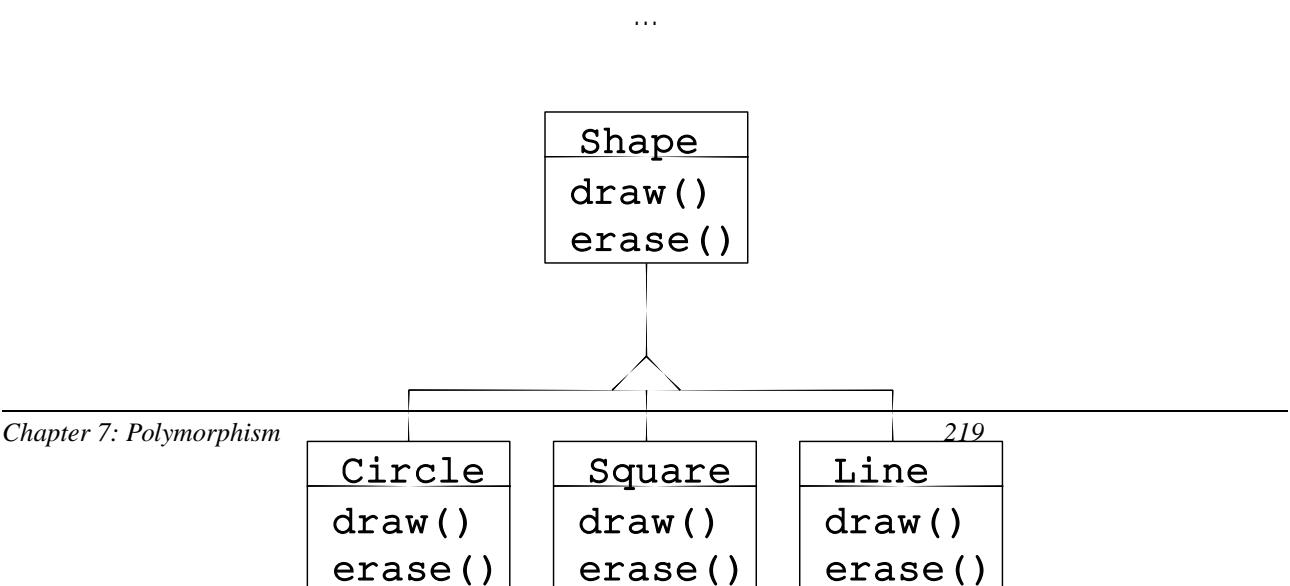
```

Stage съдържа манипулатор на **Actor**, който се инициализира с **HappyActor** обект. Това значи че **go()** произвежда частно поведение. Но доколкото манипулаторът може да бъде пресвързан към друг обект по време на изпълнение, манипулатор на **SadActor** обект може да бъде заместен в **a** и тогава поведението давано от **go()** се променя. Така се постига динамична гъвкавост по време на изпълнение. В контраст не може да решите да наследявате динамично по време на изпълнение; това трява да е напълно определено по време на компилация.

Общо правило е “Използвай наследяване за изразяване на разлики в поведението и член-променливи за изразяване вариациите в състоянието.” В горния пример и двете са използвани: два различни класа са наследени за да се изрази разликата в **act()** метода и **Stage** използва композиция за да позволи промяна в състоянието си. В този случай промяната в състоянието води промяна в поведението.

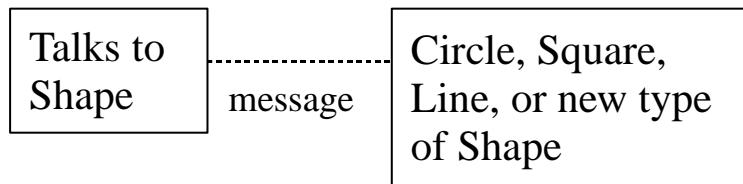
Чисто наследяване vs. разширяване

Когато изучаваме наследяването може да ни се стори че най-добрия начин да го осъществим е “чистия” подход. Тоест само методите които фигурират в кореновия клас или **interface** да бъдат подтискани в извлечения клас, както е на следната диаграма:



Това може да се означи като чиста "е" зависимост понеже интерфейса на клас я определя. Наследяването гарантира, че извлечените класове ще имат не по-малко от интерфейса на базовия клас. Ако проследите горната диаграма, извлечените класове също ще имат *не повече* от интерфейса на базовия клас.

Това може да бъде названо *чиста субституция*, понеже извлечените класове могат точно да заместят базовия клас, никога няма нужда от допълнителна информация за субкласовете когато ги използвате:

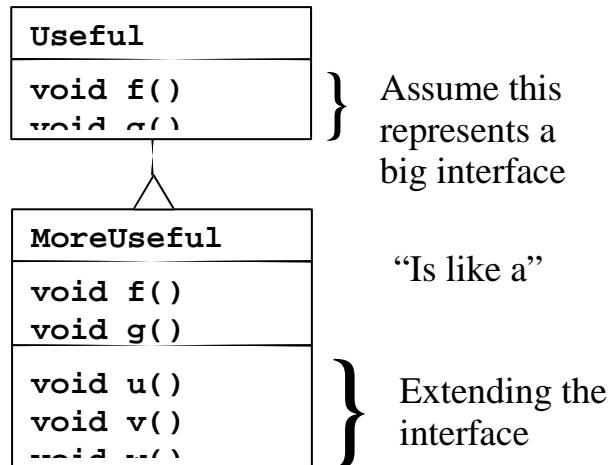


"Is a"

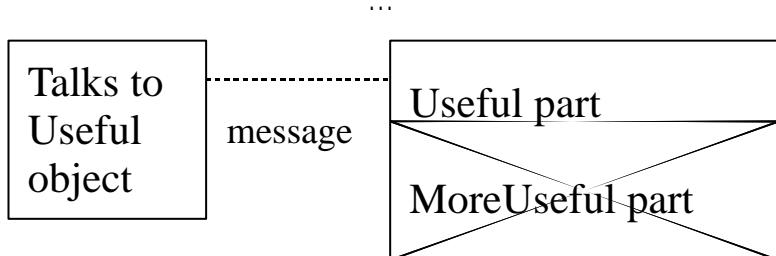
Тоест базовият и извлеченият клас могат да получават едни и същи съобщения, понеже имат един и същ интерфейс. Всичко което е необходимо е ъпкаст и никога не трябва да знаете точния тип на обекта, с който се работи. Всичко става благодарение на полиморфизма.

Ако гледаме по този начин изглежда, че чистата "е" е единствения смислен начин да се направят нещата, всеки друг дизайн индицира междуинно мислене и по определение е "фалшив". Това също е капан. Щом стъпите на тази плоскост на мислене ще се огледате наоколо и ще откриете че разширяването на интерфейса (което, за нещастие, ключовата дума **extends** изглежда да подпомага) е перфектното решение на всеки частен проблем. Това може да бъде окачествено като "прилича-на" зависимост защото извлеченият клас е като базовия клас – има същия основен интерфейс – но има други черти които изискват допълнителни методи за реализацията си:

...



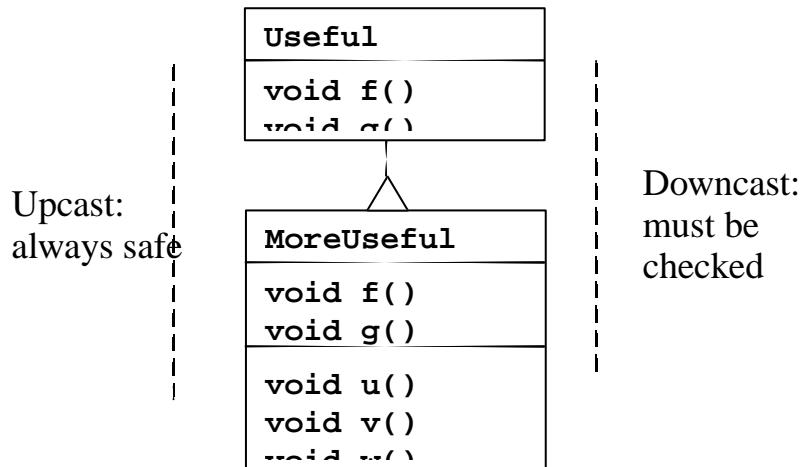
Докато това също е полезен и смислен подход (в зависимост от ситуацията) той има и недостатък. Разширената част на интерфейса на извлечения клас не е достъпна от базовия клас, така че като се направи ъпкаст не може да се викат новите методи:



Ако не се прави ъпкаст в този случай, това няма да ви беспокои, но често ще попадате в ситуации където ще трябва да преоткривате точния тип за да може да използвате съответните методи. Следващата секция показва как се прави това.

Даункастинг и идентификация на типа по време на изпълнение

Тъй като се губи специфичната за типа информация чрез *upcast* (преместване нагоре по йерархията), има смисъл да се възстанови тази информация – тоест да се отиде надолу по йерархията – и се използва *downcast*. Знае се че ъпкастът е винаги безопасен; базовият клас не може да има по-голям интерфейс от извлечения клас, така че каквито и да са съобщения подавани чрез интерфейса на базовия клас ще бъдат приети. Но с даункаста не се знае че формата (например) е фактически кръг. Би могла да е квадрат, триъгълник или някакъв друг тип.



За да се реши проблема трябва да има начин да се гарантира че даункастът е коректен, така че да не може да се направи каст към неправилен тип и после да се изпрати съобщение, което обектът не може да приеме. Това би било твърде небезопасно.

В някои езици (като C++) трябва да се изпълни специална операция, за да се получи безопасен кастинг, но в Java всеки каст се проверява! Така че даже и да изглежда че се прави само единичен каст в скоби, по време на изпълнение се проверява дали типът е този който се очаква. Ако не е, получава се **ClassCastException**. Тази дейност за проверка на типовете по време на изпълнение е наречена *run-time type identification* (RTTI). Следващият пример демонстрира поведението на RTTI:

```

//: c07:RTTI.java
// Downcasting & Run-Time Type
// Identification (RTTI)
import java.util.*;

class Useful {
    public void f() {}
    public void g() {}
}

class MoreUseful extends Useful {
    public void f() {}
    public void g() {}
    public void u() {}
    public void v() {}
    public void w() {}
}

public class RTTI {
    public static void main(String[] args) {
        Useful() x = {
            new Useful(),
            new MoreUseful()
        };
        x(0).f();
        x(1).g();
        // Compile-time: method not found in Useful:
        //!! x(1).u();
        ((MoreUseful)x(1)).u(); // Downcast/RTTI
        ((MoreUseful)x(0)).u(); // Exception thrown
    }
} ///:~

```

Както в диаграмата **MoreUseful** разширява интерфейса на **Useful**. Но понеже то е наследено, може да се направи тъпкаст към **Useful**. Може да видите това да става в инициализацията на масива **x** в **main()**. Понеже и двата обекта в масива са от клас **Useful**, може да се изпратят **f()** и **g()** методите към двете, а ако се опитате да извикате **u()** (който съществува само в **MoreUseful**) ще получите грешка по време на изпълнение.

Ако искате да използвате разширения интерфейс на обект от клас **MoreUseful** може да опитате с даункаст. Ако типът е коректен, всичко ще стане. Иначе ще получите **ClassCastException**. Не е необходимо да пишете специален код във връзка с това, понеже то индицира програмна грешка която се проявява където и да е в програмата.

Има повече за RTTI отколкото прост кааст. Например има начин да се види точния тип преди опита за даункаст. Цялата глава 11 е посветена на изучаването на идентификацията на типовете по време на изпълнение в Java.

Резюме

Полиморфизъм значи "различни форми." В ООП имате същото лице (общия интерфейс на базовия клас) и различни форми използващи това лице: различните версии на динамично свързваните методи.

Видяхте в тази глава че не е възможно да се разбере и даже да се създаде пример за полиморфизъм без използването на абстракция на данните и наследяване. Полиморфизъмът е черта която не може да се разглежда изолирано (като **switch** оператора например), а работи само в концерт, като част от "голямата картина" на зависимостите между класовете. Хората често се смущават от други, не ОО черти на Java, като претоварването на методи, които понякога биват представяни като обектно ориентирани. Не се лъжете: ако няма късно свързване няма полиморфизъм.

За да се използва полиморфизъм и с това ОО черти ефективно във вашите програми трябва да си разширите кръгозора не извън само методите и съобщенията на отделен клас, но също и общо между класовете и техните взаимозависимости. Въпреки че това изисква значителни усилия, тази борба си заслужава, понеже дава по-бърза разработка на програмите, по-добра организация на кода, разширяеми програми, по-лесна поддръжка на кода.

Упражнения

- Създайте йерархия на наследяване от **Rodent**: **Mouse**, **Gerbil**, **Hamster** и т.н.. В базовия клас дайте методи които са общи за **Rodent**-ите, подтиснете ги в извлечени класове за да осигурите специфичното за дадения **Rodent** поведение. Създайте масив **Rodents**, попълнете го със специфични типове **Rodents**, извикайте методите на базовия клас да видите какво ще стане.
- Променете упражнение 1 така че **Rodent** да е **interface**.
- Оправете проблема в **WindError.java**.
- В **GreenhouseControls.java** добавете **Event** вътрешни класове които включват и изключват вентилаторите.

8: Притежаване на обектите

Твърде прости са програмите, които имат фиксиран брой обекти с известни времена на живот.

Изобщо програмите ви ще създават обекти според критерии, които ще са известни чак по време на изпълнение на програмите. Няма да знаете докато програмата се пусне броя и даже точния тип на нужните обекти. За да се реши общия програмен проблем трябва да може да създавате произволно число обекти, по всяко време, навсякъде. Така че не може да се разчита на създаване на именуван манипулятор за всеки от обектите:

```
| MyObject myHandle;
```

понеже никога не се знае точно колко броя ще трябват.

За да се реши този доста основен проблем, Java има няколко основни начина за владеене на обекти (или по-скоро - на манипулятори). Вграденият тип е масив, който беше коментиран преди и ще се спрем още на него в тази глава. Също помощната библиотека на Java има **класове-колекции** (също известни като **контейнерни класове**, но терминът "контейнер" е използван от Swing GUI библиотеката така че тук ще се използва "колекция") които дават по-съвършен начин за владеене и даже за манипулиране на обекти. Останалата част от тази глава ще се занимава с тези неща.

Масиви

Повечето необходимо въведение в масивите се съдържа в последната част на глава 4, която показва как се дефинира и инициализира масив. Фокусът на тази глава е владеенето на обекти, а масивът е просто един начин да се направи това. Но има множество начини да се направи това, така че какво откроява масивите?

Има две неща, които отличават масивите сред другите коллекции: ефективността и типа. Масивът е най-ефикасният начин който Java дава за запомняне и достъп до обекти (фактически - до манипулятори). Масивът е прости линейна последователност, което прави достъпа до елементи бърз, но се плаща за тази скорост: когато създавате масив, дължината му е зададена и не може да се променя по време на живота му (той е обект - б.пр.). Може да искате да създавате масив с определена дължина и после, ако не ви стигне мястото, да създавате нов и да преместите ванипуляторите от стария масив в новия. Това е поведението на класа **ArrayList** който ще бъде изучен по-късно в тази глава. Поради допълнителната работа заради тази гъвкавост с дължината, обаче, **ArrayList** е забележимо по-малко ефективен от масивите.

Класът **vector** в C++ знае типа на обектите които съдържа, но това е допълнителен недостатък в сравнение с масивите в Java: Операторът на **vector** в C++ **operator()** не прави проверка за излизане от обхвата, така че може да минете зад края. (Възможно е, обаче, да питате колко е голям **vector** и методът **at()** прави проверка да не се излеза извън границите.) В Java има проверка на границите независимо дали работите с масив или коллекция – ще се получи **RuntimeException** ако излезете от границите. Както ще научите в глава 9, този тип изключение

индицира програмистка грешка и затова не е необходимо да проверявате за него в програмата. Впрочем, причината в C++ **vector** да не проверява за прескачане на границите при всеки достъп е скоростта – в Java имаме постоянни допълнителни разходи на ресурси както за масивите, така и за колекциите.

Другите родови класове за колекции, които ще се изучават в тази глава, **List**, **Set** и **Map**, работят с обектите като че последните нямат специфичен тип. Тоест те ги третират като да са **Object**, кореновият клас на класовете в Java. Това работи чудесно от една гледна точка: необходимо е да построите само една колекция, който и да е Java обект ще е подходящ за нея. (Освен примитивите – те могат да бъдат слагани като константи в колекциите чрез обгръщащите класове в Java или като променяме величини като ги включите в свой клас.) Това е второто място където масивът е най-добър за родови колекции: когато създавате масив го създавате да съдържа определен тип. Това значи че имате проверка по време на компилация да не сложите в него неправилен тип, или да не събъркато типа който извлечвате. Разбира се, Java няма да ви позволи да изпротите неподходящо съобщение към обект, както (ще провери типа-б.пр.) по време на компилация, така и по време на изпълнение. Така че едното не е по-рисковано от другото; просто едното е по-бързо, по-добре е компилаторът да се обади, тогава и вероятността крайният потребител да получи изключение е по-малка.

Заради ефективността и проверката на типовете е най-добре да използвате масиви ако е възможно. Ако обаче имате да решавате по-общ проблем масивите са твърде ограничаващи. След погледа към масивите останалата част от тази глава ще бъде посветена на колекциите, доставяни с Java.

Масивите са първокласни обекти

Независимо от типа на масива с който работите идентификаторът на масива е фактически манипулятор към обект създаден на хийпа. Обектът в хийпа може да бъде създаден или имплицитно, като част от синтаксиса на инициализацията, или явно с **new** израз. Част от тобект на хийпа (фактически единственото поле или метод който е достъпен) е членът само за четене **length** който казва колко елемента могат да бъдат запомнени. Синтаксисът '**0**' е единственият друг достъп който може да има до елементите на масив.

Следващият пример показва различни начини за инициализиране на масиви и как манипуляторите на масиви могат да бъдат присвоявани на други масиви-обекти. Също се показва че масивите от примитиви и масивите от обекти са почти еднакви откъм използване. Единствената разлика е че часивите от обекти съдържат манипулятори докато тези от примитиви съдържат направо стойности. (Виж стр. 63 ако има проблеми с изпълнението на тази програма.)

```
//: c08:ArraySize.java
// Initialization & re-assignment of arrays
package c08;

class Weeble {} // A small mythical creature

public class ArraySize {
    public static void main(String[] args) {
        // Arrays of objects:
        Weeble() a; // Null handle
        Weeble() b = new Weeble(5); // Null handles
        Weeble() c = new Weeble(4);
        for(int i = 0; i < c.length; i++)
            c(i) = new Weeble();
        Weeble() d = {
            new Weeble(), new Weeble(), new Weeble()
        };
    }
}
```

```

// Compile error: variable a not initialized:
//!System.out.println("a.length=" + a.length);
System.out.println("b.length = " + b.length);
// The handles inside the array are
// automatically initialized to null:
for(int i = 0; i < b.length; i++)
    System.out.println("b(" + i + ")=" + b(i));
System.out.println("c.length = " + c.length);
System.out.println("d.length = " + d.length);
a = d;
System.out.println("a.length = " + a.length);
// Java 1.1 initialization syntax:
a = new Weeble() {
    new Weeble(), new Weeble()
};
System.out.println("a.length = " + a.length);

// Arrays of primitives:
int[] e; // Null handle
int[] f = new int[5];
int[] g = new int[4];
for(int i = 0; i < g.length; i++)
    g(i) = i*i;
int[] h = { 11, 47, 93 };
// Compile error: variable e not initialized:
//!System.out.println("e.length=" + e.length);
System.out.println("f.length = " + f.length);
// The primitives inside the array are
// automatically initialized to zero:
for(int i = 0; i < f.length; i++)
    System.out.println("f(" + i + ")=" + f(i));
System.out.println("g.length = " + g.length);
System.out.println("h.length = " + h.length);
e = h;
System.out.println("e.length = " + e.length);
// Java 1.1 initialization syntax:
e = new int[] { 1, 2 };
System.out.println("e.length = " + e.length);
}
} ///:~

```

Ето изхода от тази програма:

```

b.length = 5
b(0)=null
b(1)=null
b(2)=null
b(3)=null
b(4)=null
c.length = 4
d.length = 3
a.length = 3
a.length = 2
f.length = 5
f(0)=0
f(1)=0
f(2)=0

```

```
f(3)=0  
f(4)=0  
g.length = 4  
h.length = 3  
e.length = 3  
e.length = 2
```

Масивът **a** е просто манипулятор сочещ **null** и компилаторът предотвратява опити да се прави с него нещо преди да е провилно инициализиран. Масивът **b** се инициализира да сочи масив от **Weeble** манипулятори, но никога не се слагат там **Weeble** обекти. Може обаче да питате колко е дължината на масива, тъй като **b** сочи законен обект. Това извежда наясък недостатък: не може да намерите колко елемента има в масива, понеже **length** казва само колко елемента може да бъдат сложени в масива; тоест, дължината на обекта-масив, не колко елемента съдържа. Обаче когато се създава обект неговия манипулятор се инициализира с **null** така че можете да видите дали даден конкретен масив има елементи като проверите дали манипуляторът му сочи **null**. По подобен начин всеки масив от примитиви се инициализира с нула за числените типове, **null** за **char** и **false** за **boolean**.

Масивът **c** показва създаване на масив от обекти и присвояването на **Weeble** обекти на всичките слотове на масива. Масивът **d** показва синтаксиса на “агрегираната инициализация” с който се получава създаване на обект-масив (неявно с **new** на хийпа, точно като масива **c**) и инициализиране с обекти **Weeble**, всичкото в един ред.

Изразът

```
| a = d;
```

как може да се вземе манипулятор присъединен към един масив-обект и да ме се присвои друг обект-масив, точно както може да се направи с всеки един манипулятор на обект. Сега както **a** така и **d** сочат един и същ масив върху хийпа.

Java 1.1 добавя нов синтаксис за инициализация на масиви, която може да си представим като “динамична агрегатна инициализация.” Агрегатната инициализация от Java 1.0 използвана за **d** трябва да бъде използвана в точката на дефиниция на **d**, но със синтаксиса на Java 1.1 може да създавате и инициализирате масиви навсякъде. Например да предположим че **hide()** е метод който приема масив от **Weeble** обекти. Бихте могли да напишете:

```
| hide(d);
```

но в Java 1.1 може също динамично да създадете масива който ще дадете като аргумент:

```
| hide(new Weeble() { new Weeble(), new Weeble() });
```

Този нов синтаксис дава по-удобен начин за писане на кода в някои ситуации.

Втората част от примера показва че масив от примитиви работи точно както масив от обекти освен че масивът от примитиви съдържа директно стойности.

Колекции от примитиви

Колекциите класове могат да съдържат само манипулятори към обекти. Масив, обаче, може да бъде създаден да съдържа директно стойности, точно както може да съдържа и манипулятори към масиви. Възможно е да се използват “обхващащи” класове като **Integer**, **Double** и т.н. за да се сложат стойности на примитиви вътре в колекция, но както ще видите по-късно в тази глава в примера **WordCount.java** обхващащите класове са малко полезни в масиви. Дали да сложите примитиви в масив или да ги обградите в класове и да ги сложите в колекции е въпрос на ефективност. Много по-ефективно е да сложите стойностите в масив отколкото класове в колекция.

Разбира се, ако вашата колекция трябва сама да може да се разширява, макар и да е от примитиви, трябва да използвате обгръщащите класове. Може да си мислите че трябва да има специализиран тип **Vector** за всеки от примитивните типове данни, но Java не прави това за нас. Някакъв вид шаблони могат да дадат един ден по-добър начина за решаване на този проблем в Java.¹

Връщане на масив

Да предположим, че пишете метод и искате да върнете не едно нещо, а цял куп неща. Езици като C и C++ не улесняват това, защото не може да се върне направо масив, само указател към масив. Това докарва проблеми защото е трудно да се следи времето на живот на масив, което лесно води до изтичания на памет.

Java приема подобен подход, но просто се "връща масив." В действителност, разбира се, се връща манипулятор към масив, но с Java нямате отговорност за този масив – той ще бъде налице колкото ви трябва, а боклучарят ще почисти когато му дойде времето.

Като пример да вземем връщането на масив от тип **String**:

```
//: c08:IceCream.java
// Returning arrays from methods

public class IceCream {
    static String[] flav = {
        "Chocolate", "Strawberry",
        "Vanilla Fudge Swirl", "Mint Chip",
        "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie"
    };
    static String[] flavorSet(int n) {
        // Force it to be positive & within bounds:
        n = Math.abs(n) % (flav.length + 1);
        String[] results = new String(n);
        boolean[] picked =
            new boolean[flav.length];
        for (int i = 0; i < n; i++) {
            int t;
            do
                t = (int)(Math.random() * flav.length);
            while (picked(t));
            results(i) = flav(t);
            picked(t) = true;
        }
        return results;
    }
    public static void main(String[] args) {
        for(int i = 0; i < 20; i++) {
            System.out.println(
                "flavorSet(" + i + ") = ");
            String[] fl = flavorSet(flav.length);
            for(int j = 0; j < fl.length; j++)
                System.out.println("\t" + fl(j));
        }
    }
} ///:~
```

¹ This is one of the places where C++ is distinctly superior to Java, since C++ supports *parameterized types* with the **template** keyword.

Методът `flavorSet()` създава масив от `String` наречен `results`. Дължината на този масив е `n`, определена от аргумента който се дава на метода. После продължава случайното избиране измежду `flav` и слагането им в `results`, което накрая се връща. Връщането на масив е точно като връщането на всеки един обект – връща се манипулятор. Не е важно че масивът бе създаден в `flavorSet()`, нито ако е бил създаден където и да е другаде, в този смисъл. Боклукарят се грижи за махането на масива когато вече не е нужен, а той ще стои докато е нужен.

Между другото, забележете че `flavorSet()` избира по случаен начин като осигурява да няма избиране два пъти на един елемент. Това се прави в `do` цикъл който продължава избора докато намери че последното избрано не присъства в масива `picked`. (Разбира се, сравняването на `String` също би могло да бъде използвано, но сравняването на `String` е неефективно.) Ако изборът е успешен, добавя се. (`i` се инкрементира).

`main()` извежда 20 пълни множества вкус, така че може да видите че `flavorSet()` използва случаино избиране всеки път. Това е по-лесно да се наблюдава ако се пренасочи изходът към файл. И, докато гледате файла, помнете, не сте наистина гладни. (Вие само искате сладоледа, вие не се нуждате от него в действителност.)

Колекции

Да сумираме до тук: вашият пръв, най-ефективен начин за владеене на обекти е масив, а и сте принудени да използвате масив ако владяното е примитиви. До края на главата ще разгледаме по-общия случай, когато не се знае към момента на написването на програмата колко обекта ще са нужни, та трябва по-усложнен начин за владеенето на обектите. Java има три типа **класове-колекции** за решаване на този проблем: `List`, `Set` и `Map`. Изненадващо много проблеми могат да се решат с тези три инструменти.

Сред другите им характеристики – `Set`, например, може да съдържа само един обект с дадена стойност, а `Map` е асоциативен масив което позволява да се асоциира обект с някой друг обект – е и тази, че колекторните класове на Java автоматично сами ще си променят дължината при нужда. По този начин може да се владеят произволен брой обекти и не е необходимо да се грижим за това по време на написването на програмите.

Неудобство: неизвестен тип

“Неудобството” да се използват колекциите на Java е, че когато сложите обект в тях губите информацията за типа му. Това става защото по времето на написването на класа-колекция програмистът не е имал представа за конкретния тип класове, които ще образуват колекцията, а освен това ако се направи да се поддържа само вашият тип колекцията няма вече да е инструмент за всеобща употреба. Така че колекцията съдържа манипулатори на обекти от тип `Object`, каквито са разбира се всички обекти в Java, понеже това е кореновият клас. (Това не включва примитивните класове, понеже те не са наследени от нищо.) Това е много добро решение, освен в следните случаи:

1. Тъй като информацията за типа се изхвърля когато сложите обект в колекцията, всякали типове обекти могат да се сложат там, даже и да сте възнамерявали да сложите, да речем, само котки. Някой много лесно може да сложи куче в колекцията.
2. Тъй като информацията за типа е изгубена, единственото известно нещо в колекцията е че се съхранява манипулятор към `Object`. Трябва да се направи каст към типа треди да се използва манипулатора.

Откъм предимствата: Java не дава да се използват погрешно обектите които сте сложили в колекцията. Ако хвърлите куче в колекцията от котки, после минете през колекцията опитвайки

се да третирате всичко като котки, ще получите изключение когато стигнете до кучето. В същия смисъл, ако се опитате да направите каст на манипулятор към куче да стане на котка ще получите изключение по време на изпълнение.

Ето пример:

```
//: c08:CatsAndDogs.java
// Simple collection example
import java.util.*;

class Cat {
    private int catNumber;
    Cat(int i) {
        catNumber = i;
    }
    void print() {
        System.out.println("Cat #" + catNumber);
    }
}

class Dog {
    private int dogNumber;
    Dog(int i) {
        dogNumber = i;
    }
    void print() {
        System.out.println("Dog #" + dogNumber);
    }
}

public class CatsAndDogs {
    public static void main(String[] args) {
        ArrayList cats = new ArrayList();
        for(int i = 0; i < 7; i++)
            cats.add(new Cat(i));
        // Not a problem to add a dog to cats:
        cats.add(new Dog(7));
        for(int i = 0; i < cats.size(); i++)
            ((Cat)cats.get(i)).print();
        // Dog is detected only at run-time
    }
} ///:~
```

Може да се види, че използването на `ArrayList` е праволинейно: създавате го, слагате обекти вътре чрез `add()`, а по-късно ги вадите чрез `get()`. (Забележете че `Vector` има метод `size()` за да може да видите колко елемента са прибавени и да не стъпите неочеквано извън края, предизвиквайки изключение по време на изпълнението.)

Класовете `Cat` и `Dog` са различни – нямат нищо общо помежду си, освен че са `Objects`. (Ако не кажете явно от кой клас наследявате, наследявате от `Object`.) Класът `Vector` който идва от `java.util`, държи `Objects`, така че не само може да слагате `Cat` в тази колекция използвайки метода на `Vector add()`, но може също да добавите обекти `Dog` без оплаквания и грешки по време на изпълнение. Когато се опитате да извадите неща които мислите че са `Cat` обекти използвайки метода на `Vector get()`, получавате манипулятор към `Object` който трябва да се превърне в такъв към `Cat`. Тогава трябва да сложите целия израз в скоби за да стане превръщането проди да се използва методът `print()` на `Cat`, иначе ще получите синтаксична

грешка. По време на изпълнение, когато се опитате да направите каст на **Dog** обект към **Cat**, ще получите изключение.

Това е повече от просто досадно. Това е нещо, което може да доведе до трудни за откриване грешки. Ако една (или няколко) части от програма вмъкват обекти в колекция, а видите че само отделна част е изхвърлила изключение за неправилен тип на обект в колекцията, трябва да намерите къде е станало неправилното вмъкване. Това се прави чрез преглеждане на кода, който е почти най-лошия инструмент за откриване на грешки. От друга страна, удобно е да се започне с някаква стандартизирана колекция, напук на осъдицата и тромавостта.

Понякога работи правилно при всички случаи

Понякога изглежда че нещата стават както трябва без кастинг към оригиналния тип. Първият случай е доста специален: Класът **String** има малко по-голяма помощ от страна на компилатора за да работи гладко. Щом компилаторът очаква **String** обект и не получава такъв, той автоматично ще извика метода **toString()** който е определен в **Object** и може да бъде подтиснат от всеки Java клас. Този метод дава желания **String** обект, който после се използва където е нужно.

Така всичко което трябва да направите за да извеждат вашите обекти е да подтиснете метода **toString()** както е показано в следващия пример:

```
//: c08:WorksAnyway.java
// In special cases, things just seem
// to work correctly.
import java.util.*;

class Mouse {
    private int mouseNumber;
    Mouse(int i) {
        mouseNumber = i;
    }
    // Magic method:
    public String toString() {
        return "This is Mouse #" + mouseNumber;
    }
    void print(String msg) {
        if(msg != null) System.out.println(msg);
        System.out.println(
            "Mouse number " + mouseNumber);
    }
}

class MouseTrap {
    static void caughtYa(Object m) {
        Mouse mouse = (Mouse)m; // Cast from Object
        mouse.print("Caught one!");
    }
}

public class WorksAnyway {
    public static void main(String[] args) {
        ArrayList mice = new ArrayList();
        for(int i = 0; i < 3; i++)
            mice.add(new Mouse(i));
        for(int i = 0; i < mice.size(); i++) {
```

```

// No cast necessary, automatic call
// to Object.toString():
System.out.println(
    "Free mouse: " + mice.get(i));
MouseTrap.caughtYa(mice.get(i));
}
}
} //:~

```

You can see the redefinition of **toString()** in **Mouse**. In the second **for** loop in **main()** you find the statement:

```
| System.out.println("Free mouse: " + mice.get(i));
```

След знака '+' компилаторът очаква да види **String** обект. **get()** дава **Object**, така че за да получи желания **String** компилаторът неявно вика **toString()**. За нещастие тези магии стават само със **String**; няма ги за никой друг тип.

Вторият подход за скриване на кастинга е заложен в **Mousetrap**. Методът **caughtYa()** приема не-**Mouse**, но **Object**, който после превръща в **Mouse**. Това е твърде самонадеяно, разбира се, понеже чрез приемане на **Object** нищо не би могло да бъде подадено към метода. Ако обаче кастингът не е коректен – ако сте подали неправилен тип – ще получите изключение по време на изпълнение. Това не е толкова добре като проверката по време на компилация, но все още бива. Забележете че при използването на метода:

```
| MouseTrap.caughtYa(mice.get(i));
```

Не е необходим каст.

Правене на **Vector** чувствителен към типа

Може да не искате да изоставите този въпрос точно сега. По-добро решение е да наследите **Vector**, така че да приема само вашия тип и да произвежда само вашия тип:

```

//: c08:GopherList.java
// A type-conscious ArrayList
import java.util.*;

class Gopher {
    private int gopherNumber;
    Gopher(int i) {
        gopherNumber = i;
    }
    void print(String msg) {
        if(msg != null) System.out.println(msg);
        System.out.println(
            "Gopher number " + gopherNumber);
    }
}

class GopherTrap {
    static void caughtYa(Gopher g) {
        g.print("Caught one!");
    }
}

class GopherList {
    private ArrayList v = new ArrayList();
    public void add(Gopher m) {

```

```

    v.add(m);
}
public Gopher get(int index) {
    return (Gopher)v.get(index);
}
public int size() { return v.size(); }
public static void main(String[] args) {
    GopherList gophers = new GopherList();
    for(int i = 0; i < 3; i++)
        gophers.add(new Gopher(i));
    for(int i = 0; i < gophers.size(); i++)
        GopherTrap.caughtYa(gophers.get(i));
}
} //:~

```

Това е като в предния пример, освен че новият клас **GopherVector** има **private** член от тип **Vector** (наследяването от **Vector** има тенденция да разочарова, по причини които ще видите по-късно), и методи точно като **Vector**. Обаче той не приема и не произвежда родови **Objects**, само **Gopher** обекти.

Понеже **GopherVector** ще приема само **Gopher**, ако бяхте написали:

```
| gophers.add(new Pigeon());
```

щяхте да получите грешка *по време на компилация*. Този подход, много по-досаден от програмистка гледна точка, ще ви каже незабавно ако не използвате правилен тип.

Забележете че не е необходим каст за използване на **get()** – то си е **Gopher**.

Параметризирани типове

Този вид проблеми не е изолиран – има множество случаи, когато трябва да получите нови типове от налични типове, в които е необходимо да се разполага с информация за типа по време на компилация. Това е концепцията за параметризиран тип. В C++ това се поддържа директно от езика с шаблон. В определен момент Java резервира ключовата дума **generic** за да я използва някой ден за параметризираните типове, но не е сигурно, че това никога ще се случи.

Итератори

Във всеки клас-колекция трябва да има начин да се пъхат вътре класове и да се вадят от там. Най-сетне това е основната задача на колекцията – да съдържа неща. В масива **ArrayList add()** начинът да се вмъхват обекти, а **get()** е един начин да се вадят. **ArrayList** е доста гъвкав – може да изберете всяко нещо по всяко време, да избирате няколко неща едновременно с различни индекси.

Ако се мисли на по-високо ниво има неудобство: необходимо е да се знае точния тип на колекцията за да може да се използва тя. Това по начало може да не изглежда зле, но ако започнете с използване на **ArrayList**, а после за ради ефективност решите, че трябва да преминете към **LinkedList**? Или бихте желали да напишете парче код което не трябва да знае и да се грижи с каква колекция работи.

Концепцията за *итератор* може да се използва за достигане на това по-високо ниво на абстракция. Това е обект който може да се движи през редицата обекти и да работи с тях без приложният програмист да трябва да знае нещо за структурата на въпросната редица. В добавка итераторът обикновено е това, което се нарича “лек обект”; тоест лесен за създаване. Поради последната причина често ще срещате на пръв поглед странни

ограничения за итераторите; например някои итератори могат да ровят само в едната посока.

В Java **Iterator** е пример за итератор с този тип ограничения. Не може много да се направи с него освен:

1. Да се иска от колекцията да даде **Iterator** чрез метод наречен **iterator()**. Този **Iterator** Този итератор ще може да даде първия елемент на колекцията при последващо извикване на метода **next()**.
2. Да време следващия обект чрез **next()**.
3. Да види дали има още обекти чрез **hasNext()**.
4. Да махне последния елемент върнат от итератора чрез **remove()**.

Това е всичко. Проста реализация на итератор, но все още мощна. За да видим как работи, да повторим **CatsAndDogs.java** програмата от тази глава. В оригиналната версия методът **get()** беше използван да се избере всеки елемент, но в тази версия ще използваме итератор:

```
//: c08:CatsAndDogs2.java
// Simple collection with Iterator
import java.util.*;

class Cat2 {
    private int catNumber;
    Cat2(int i) {
        catNumber = i;
    }
    void print() {
        System.out.println("Cat number " +catNumber);
    }
}

class Dog2 {
    private int dogNumber;
    Dog2(int i) {
        dogNumber = i;
    }
    void print() {
        System.out.println("Dog number " +dogNumber);
    }
}

public class CatsAndDogs2 {
    public static void main(String[] args) {
        ArrayList cats = new ArrayList();
        for(int i = 0; i < 7; i++)
            cats.add(new Cat2(i));
        // Not a problem to add a dog to cats:
        cats.add(new Dog2(7));
        Iterator e = cats.iterator();
        while(e.hasNext())
            ((Cat2)e.next()).print();
        // Dog is detected only at run-time
    }
} ///:~
```

Вижда се, че единствената промяна е в последните няколко реда. Вместо:

```
for(int i = 0; i < cats.size(); i++)
    ((Cat)cats.get(i)).print();
```

се използва **Iterator** за стъпка през редицата:

```
while(e.hasNext())
    ((Cat2)e.next()).print();
```

С **Iterator** няма нужда да се беспокоите за броя на елементите в колекцията. Това се прави заради вас чрез **hasNext()** и **next()**.

Като друг пример да видим разработването на метод за извеждане за обща употреба:

```
//: c08:HamsterMaze.java
// Using an Iterator
import java.util.*;

class Hamster {
    private int hamsterNumber;
    Hamster(int i) {
        hamsterNumber = i;
    }
    public String toString() {
        return "This is Hamster #" + hamsterNumber;
    }
}

class Printer {
    static void printAll(Iterator e) {
        while(e.hasNext())
            System.out.println(
                e.next().toString());
    }
}

public class HamsterMaze {
    public static void main(String[] args) {
        ArrayList v = new ArrayList();
        for(int i = 0; i < 3; i++)
            v.add(new Hamster(i));
        Printer.printAll(v.iterator());
    }
} ///:~
```

Поглед по-отблизо до метода:

```
static void printAll(Iterator e) {
    while(e.hasNext())
        System.out.println(
            e.next().toString());
}
```

Забележете че няма информация за типа на последователността. Всичко което има е **Iterator**, а това е всичко, което трябва да знаете за последователността: така може да се вземе следващия обект, така се разбира къде сте спрямо края. Идеята за вземане на колекция от обекти и преминаване през нея за да се свърши работа е мощна и ще личи до края на книгата.

Този конкретен пример е даже повече родов, понеже използва вездесъщия **toString()** метод (вездесъщ само защото е част от класа **Object**). Друг начин да се извика извеждане (макар и малко по-малко ефективен, ако въобще забележите разликата) е:

```
| System.out.println("") + e.next());
```

който използва "автоматично превръщане към **String**" което е вградено в Java. Когато компилаторът види **String**, следван от '+', той очаква друг **String** да следва и вика **toString()** автоматично. (В Java 1.1 първият **String** не е необходим; всеки обект ще бъде преобразуван към **String**.) Може също да направите каст, което има ехекта на извикване на **toString()**:

```
| System.out.println((String)e.next());
```

Изобщо, обаче, ще трябва да се прави повече от викане на методи на **Object** така че отново ще се натъкнете на нуждата от преобразуване на типове. Трябва да предполагате че имате **Iterator** за последователност ота от типа, с който работите, и да преобразувате към него тип (получавайки изключение по време на изпълнение ако не е както трябва).

ТИПОВЕ КОЛЕКЦИИ

Стандартните библиотеки на Java 1.0 и 1.1 идват с възможния минимум класове-колекции, но те вероятно са достатъчни за большинството ваши програмни проекти. (Както ще видите на края на тази глава Java 2 има радикално преработена и попълнена библиотека.)

ArrayList

ArrayList е доста прост за използване, както вече видяхме. Въпреки че повечето време се използва **add()** за вмъкване на обекти, **get()** за измъкване по един и **elements()** за да се получи **Iterator** към редицата, има също и други методи, които могат да бъдат полезни. Както обикновено с Java библиотеките, ние няма да ги използваме или да говорим за всичките, но трябва да погледнете в електронната документация за да получите представа какво могат.

Скапване на Java

Стандартните колекции в Java съдържат метод **toString()** така че могат да произведат **String** представяне на самите себе си, включително съдържаните в тях обекти. Вътре в **ArrayList**, например, **toString()**-тът върви по елементите на **ArrayList** и вика **toString()** за всеки. Да речем че искате да изведете адреса на вашия клас. Изглежда смислено просто да се използва **this** (C++ програмистите в частност са предразположени към този начин):

```
//: c08:CrashJava.java
// One way to crash Java
import java.util.*;

public class CrashJava {
    public String toString() {
        return "CrashJava address: " + this + "\n";
    }
    public static void main(String[] args) {
        ArrayList v = new ArrayList();
        for(int i = 0; i < 10; i++)
            v.add(new CrashJava());
        System.out.println(v);
    }
} //:~
```

Оказва се че ако просто създадете **CrashJava** обект и го изведете, ще получите безкрайна редица от изключения. Ако обаче сложите обектите **CrashJava** в **ArrayList** и го изведете този **ArrayList** както е показано тук, той не може да се справи и не получавате даже изключение; Java просто се скапва. (Но най-малкото не повлича и операционната система.) Това беше тествано с Java 1.1.

Това което става е автоматично превръщане към **String**ове. Като напишете:

```
| "CrashJava address: " + this
```

компилаторът вижда **String** следван от '+' и нещо, което не е **String**, така че се опитва да превърне **this** в **String**. Прави това чрез извикване на **toString()**, което дава рекурсивно извикване. Когато това се случи вътре в **ArrayList**, изглежда че стекът се препълва преди механизъмът за следенето му да има време да се намеси.

Ако наистина искате да изведете адреса на обекта в подобен случай, решението е да извикате **Object toString()** метода, който прави точно това. Така че заместо **this** ще напишете **super.toString()**. (Това работи само ако направо сте наследили от **Object** или никой от родителските класове не е подтиснал метода **toString()**).

BitSet

BitSet е в действителност **Vector** за битове и се използва за обработка на много информация от типа "включено-изключено". Той е ефективен само от гледна точка на дължината; ако ви трябва ефикасен достъп трябва да знаете че е малко по-бавен от случая с обикновените данни типове.

В добавка минималната дължина на **BitSet** е дължината на **long**: 64 бита. Това значи че ако работите с нещо по-малко, да кажем 8 бита, **BitSet** ще бъде малко прахоснически, така че е по-добре да създадете собствен клас за работа с вашите флагове.

В нормален **Vector** колекцията ще се разширява с добавянето на нови елементи. **BitSet** също прави това – някак си. Тоест понякога го прави, а понякога не, което прави да изглежда, че реализацията на **BitSet** в Java версия 1.0 просто е лошо направена. (Това е оправено в Java 1.1.) Следващият пример показва как работи **BitSet** и демонстрира бъга на версия 1.0:

```
//: c08:Bits.java
// Demonstration of BitSet
import java.util.*;

public class Bits {
    public static void main(String[] args) {
        Random rand = new Random();
        // Take the LSB of nextInt():
        byte bt = (byte)rand.nextInt();
        BitSet bb = new BitSet();
        for(int i = 7; i >=0; i--)
            if((1 << i) & bt != 0)
                bb.set(i);
            else
                bb.clear(i);
        System.out.println("byte value: " + bt);
        printBitSet(bb);

        short st = (short)rand.nextInt();
        BitSet bs = new BitSet();
        for(int i = 15; i >=0; i--)
```

```

if(((1 << i) & st) != 0)
    bs.set(i);
else
    bs.clear(i);
System.out.println("short value: " + st);
printBitSet(bs);

int it = rand.nextInt();
BitSet bi = new BitSet();
for(int i = 31; i >=0; i--)
    if(((1 << i) & it) != 0)
        bi.set(i);
    else
        bi.clear(i);
System.out.println("int value: " + it);
printBitSet(bi);

// Test bitsets >= 64 bits:
BitSet b127 = new BitSet();
b127.set(127);
System.out.println("set bit 127: " + b127);
BitSet b255 = new BitSet(65);
b255.set(255);
System.out.println("set bit 255: " + b255);
BitSet b1023 = new BitSet(512);
// Without the following, an exception is thrown
// in the Java 1.0 implementation of BitSet:
//   b1023.set(1023);
//   b1023.set(1024);
System.out.println("set bit 1023: " + b1023);
}

static void printBitSet(BitSet b) {
    System.out.println("bits: " + b);
    String bbits = new String();
    for(int j = 0; j < b.size() ; j++)
        bbits += (b.get(j) ? "1" : "0");
    System.out.println("bit pattern: " + bbits);
}
} ///:~

```

За случайното формиране на **byte**, **short** и **int** се използва генератор на случайни числа и всяко се трансформира в съответния битов низ в **BitSet**. Това работи чудесно понеже **BitSet** е 64 бита, така че никое не причинява нарастващо на дължината. Но в Java 1.0, когато **BitSet** е повече от 64 бита, се проявява странно поведение. Ако сетнете бит който е с едно след алокираната от **BitSet** в момента памет, той ще се разшири весело. Но ако се опитате да сетнете битове които не са точно до границата, ще получите изключение, понеже **BitSet** няма да се разшири правилно в Java 1.0. Примерът показва **BitSet** от 512 бита. Конструкторът алокира памет за два пъти повече битове. Ако се опитате да сетнете бит 1024 или по-голям номер без първо да сетнете бит 1023, ще изхвърлите изключение в Java 1.0. За щастие това е оправено в Java 1.1, но избягвайте използването на **BitSet** ако пишете код за Java 1.0.

Stack

Stack понякога се споменава като “last-in, first-out” (LIFO) колекция. Тоест каквото сте “вкарали” в **Stack** последно първо го “изкарвате”. Както във всички други колекции в Java,

това което се вкарва и изкарва са **Object**, така че трябва да преобразувате което изкарвате.

Лошото е че наместо да се използва **Vector** като градивен блок за изграждане на **Stack**, **Stack** е наследен от **Vector**. Така че има всичкото поведение и характеристики на **Vector** плюс някои допълнителни черти на **Stack**. Трудно е да се каже дали проектаните са намерили точно този начин за необходимим да се направят нещата или това просто е наивно проектирано.

Ето проста демонстрация на **Stack** която чете ред по ред от масив и ги пъха във вид на **String**:

```
//: c08:Stacks.java
// Demonstration of Stack Class
import java.util.*;

public class Stacks {
    static String[] months = {
        "January", "February", "March", "April",
        "May", "June", "July", "August", "September",
        "October", "November", "December" };
    public static void main(String[] args) {
        Stack stk = new Stack();
        for(int i = 0; i < months.length; i++)
            stk.push(months[i] + " ");
        System.out.println("stk = " + stk);
        // Treating a stack as a Vector:
        stk.addElement("The last line");
        System.out.println(
            "element 5 = " + stk.elementAt(5));
        System.out.println("popping elements:");
        while(!stk.empty())
            System.out.println(stk.pop());
    }
} //:~
```

Всеки ред в масива **months** се пъха в **Stack** с **push()**, а по-късно се извлича от там с **pop()**. За да се подчертаят, операции на **Vector** също се изпълняват със **Stack** обект. Това е възможно поради факта, че чрез наследяването **Stack** е **Vector**. Така всички операции които може да се изпълнят с **Vector** могат също да се изпълнят със **Stack**, като например **elementAt()**.

Map

Vector позволява да се избира от последователността посредством номер, така че може да се асоциира с определено количество обекти. Какво обаче ще стане, ако искате да изберете между обектите по някакъв друг критерий? **Stack** е един такъв пример: критерият при него е "това нещо което последно е влязло в стека." Мощно развитие на идеята за "избор от последователност" е алтернативно наричано карта, а речник или асоциативно поле. Концептуално тя прилича на Вектора, но наместо да избирате обект по номер го избирате чрез друг обект! Това често е клетков процес в програмите.

Концепцията се появява в Java като **abstract**ния клас **Dictionary**. Интерфейсът му е праволинеен: **size()** казва колко елемента има вътре, **isEmpty()** е **true** ако няма елементи, **put(Object key, Object value)** добавя стойност (нещото което искате), асоциира го с ключ (нещото чрез което ще търсите). **get(Object key)** дава стойността съответстваща на определен ключ, а **remove(Object key)** маха двойката ключ-стойност от списъка. Има Итератори: **keys()** дава **Iterator** за ключовете, а **elements()** дава **Iterator** за всички стойности. Това е всичко за **Dictionary**.

Dictionary не е ужасяващо труден за прилагане. Ето един прост подход, който използва два **Vectora**, един за ключовете и един за стойностите:

```
//: c08:AssocArray.java
// Simple version of a Map
import java.util.*;

public class AssocArray extends AbstractMap {
    private ArrayList keys = new ArrayList();
    private ArrayList values = new ArrayList();
    public int size() { return keys.size(); }
    public boolean isEmpty() {
        return keys.isEmpty();
    }
    public Object put(Object key, Object value) {
        int index = keys.indexOf(key);
        if (index == -1) { // Key not found
            keys.add(key);
            values.add(value);
            return null;
        } else { // Key already in table; replace
            Object returnval = values.get(index);
            values.set(index, value);
            return returnval;
        }
    }
    public Object get(Object key) {
        int index = keys.indexOf(key);
        // indexOf() Returns -1 if key not found:
        if(index == -1) return null;
        return values.get(index);
    }
    public Object remove(Object key) {
        int index = keys.indexOf(key);
        if(index == -1) return null;
        keys.remove(index);
        Object returnval = values.get(index);
        values.remove(index);
        return returnval;
    }
    public Set keySet() {
        return new HashSet(keys);
    }
    public Collection values() {
        return values;
    }
    public Set entrySet() {
        Set set = new HashSet();
        // Iterator it = keys.iterator();
        // while(it.hasNext()) {
        //     Object k = it.next();
        //     Object v = values.get(values.indexOf(k));
        //     set.add(new Map.Entry(k, v));
        // }
        return set;
    }
    // Test it:
}
```

```

public static void main(String[] args) {
    AssocArray aa = new AssocArray();
    for(char c = 'a'; c <= 'z'; c++)
        aa.put(String.valueOf(c),
               String.valueOf(c)
               .toUpperCase());
    char[] ca = { 'a', 'e', 'i', 'o', 'u' };
    for(int i = 0; i < ca.length; i++)
        System.out.println("Uppercase: " +
                           aa.get(String.valueOf(ca(i))));

}
} //:~

```

Първото нещо което личи в дефиницията на **AssocArray** е че той **extends Dictionary**. Това значи че **AssocArray** е от типа **Dictionary**, така че може да му поръчвате същите неща като на **Dictionary**. Ако направите собствен **Dictionary**, както е сторено тук, всичко което трябва да се направи е да се попълнят методите в **Dictionary**. (И трябва да подтиснете всички методи, понеже всички са – с изключение на конструктора – абстрактни.)

Vectorите **keys** и **values** са свързани с общ индексен номер. Тоест ако извикате **put()** с ключ “roof” и стойност “blue” (предполага се, че асоциирате различните части на къща с цветовете, в които те са боядисани) и вече има 100 елемента в **AssocArray**, тогава “roof” ще бъде 101-я елемент на **keys** и “blue” ще бъде 101 елемент на **values**. Ако погледнете **get()** когато му давате “roof” като ключ, той произвежда индексен номер с **keys.indexOf()**, а после дава стойността чрез вземане по индексния номер от вектора **values**.

Тестът в **main()** е прост; това е карта (в тази секция “карта” има смисъл на “съответствие” -бел.пр.) на малки букви към големи, което очевидно би могло да се направи по много други по-ефективни начини. Но тук показва че **AssocArray** е работоспособен.

Стандартният Java съдържа два различни типа **Map**ове: **HashMap** и **TreeMap**. И двата имат за интерфейс **HashMap** (понеже и двата реализират **Map**), но се различават по нещо съществено: ефективността. Ако гледате какво прави **get()**, доста бавно е да се търси през **ArrayList** за ключа. Това е мястото където **HashMap** ускорява нещата. Вместо тъпото линейно търсене се използва *hash code*. Наш кодът е начин да се вземе нещо от ключа и да се направи “относително уникален” **int** за обекта на ключа. Всички обекти имат наш код, а **hashCode()** е метод в кореновия клас **Object**. **HashMap** взима **hashCode()** за обект и го използва за бърз лов на ключа. Това резултира в драматично подобрене на производителността.² Начинът по който **HashMap** работи е извън обхвата на тази книга³ – всичко което трябва да знаете е че **HashMap** е бърз **Dictionary** и че **Dictionary** е полезен инструмент.

Като пример за използването на **HashMap** да вземем програма за проверка на случайността на **Math.random()** метода в Java. В идеалния случай той трябва да дава перфектно разпределени случајни числа, но за да се провери това трябва да се произведат множество числа и да се провери дали попадат в съответния обхват. **HashMap** е перфектния начин, понеже асоциира обекти с обекти (в този случай стойностите произведени от **Math.random()** с броя пъти които числата се появяват):

² If these speedups still don't meet your performance needs, you can further accelerate table lookup by writing your own hash table routine. [to-This](#) avoids [the](#)-delays due to [a](#)-casting to and from **Objects**, and [b](#)-synchronization built into the Java Class Library hash table routine. To reach even higher levels of performance, speed enthusiasts can use Donald Knuth's *The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition* to replace overflow bucket lists with arrays [which](#)-[that](#) have two additional benefits: they can be optimized for disk storage characteristics and they can save most of the time of creating and garbage collecting individual records.

³ The best reference I know of is *Practical Algorithms for Programmers*, by Andrew Binstock and John Rex, Addison-Wesley 1995.

```

//: c08:Statistics.java
// Simple demonstration of HashMap
import java.util.*;

class Counter {
    int i = 1;
    public String toString() {
        return Integer.toString(i);
    }
}

class Statistics {
    public static void main(String[] args) {
        HashMap ht = new HashMap();
        for(int i = 0; i < 10000; i++) {
            // Produce a number between 0 and 20:
            Integer r =
                new Integer((int)(Math.random() * 20));
            if(ht.containsKey(r))
                ((Counter)ht.get(r)).i++;
            else
                ht.put(r, new Counter());
        }
        System.out.println(ht);
    }
} /**

```

В **main()** когато се произведе число се обвива в **Integer** така че манипуляторът да може да се използва с **HashMap**. (Не може да се използва примитив с колекция, само манипулятор на обект.) Методът **containsKey()** методът проверява дали този ключ не е вече в колекцията. (Тоест, числото е излизало вече?) Ако е така методът **get()** методът взема асоциирана стойност за ключа, която в случая е **Counter** обект. Стойността **i** вътре в обекта се инкрементира за да индицира, че още веднъж тази стойност е излизала.

Ако ключът не е намиран още методът **put()** ще сложи нова двойка ключ-стойност в **HashMap**. Понеже **Counter** автоматично инициализира стойността на **i** с единица когато се създава, това дава и първата появя на числото.

Методът на **HashMap toString()** се движи през всички двойки ключ-стойност и изпълнява **toString()** за всяка. **Integer toString()** е по начало дефиниран, така че може да погледнете **toString()** за **Counter**. Изходът от едно пускане (с малко редактиране за прегледност) е:

```
{19=526, 18=533, 17=460, 16=513, 15=521, 14=495,
 13=512, 12=483, 11=488, 10=487, 9=514, 8=523,
 7=497, 6=487, 5=480, 4=489, 3=509, 2=503, 1=475,
 0=505}
```

Може да се зачудите дали е необходим класът **Counter** който сякаш няма и функционалността на обгръщащия клас **Integer**. Защо не използваме **int** или **Integer**? Не може да се използва **int** понеже всичките колекции работят само с манипулятори към обекти **Object**. След срещата с колекциите обгръщащите класове могат да придобият по-голям смисъл за вас, понеже не може да се работи с примитиви в колекциите. Обаче единственото нещо което можете да правите с обхващащите класове на Java е да ги инициализирате с някаква стойност и след това да я прочитате. Тоест няма начин да се промени стойността след като е създаден обектът. Това прави **Integer** направо безполезен за решаване на нашия проблем, така че се налага да направим нов клас за да си удовлетворим нуждите.

Създаване на “ключови” класове

В предишния пример стандартен библиотечен клас (**Integer**) бе използван като ключ за **HashMap**. Той работеше добре като ключ, понеже е снабден с всичко необходимо за това. Има капан обаче когато се използва **HashMap**ове когато създавате собствени класове за използване като ключове. Например да вземем система за предсказване на времето която съпоставя **Groundhog** обекти с **Prediction** обекти. Изглежда много праволинейно: създавате двата класа и използвате **Groundhog** като ключ и **Prediction** като стойност:

```
//: c08:SpringDetector.java
// Изглежда правдоподобно, но не работи добре.
import java.util.*;

class Groundhog {
    int ghNumber;
    Groundhog(int n) { ghNumber = n; }
}

class Prediction {
    boolean shadow = Math.random() > 0.5;
    public String toString() {
        if(shadow)
            return "Six more weeks of Winter!";
        else
            return "Early Spring!";
    }
}

public class SpringDetector {
    public static void main(String[] args) {
        HashMap ht = new HashMap();
        for(int i = 0; i < 10; i++)
            ht.put(new Groundhog(i), new Prediction());
        System.out.println("ht = " + ht + "\n");
        System.out.println(
            "Looking up prediction for groundhog #3:");
        Groundhog gh = new Groundhog(3);
        if(ht.containsKey(gh))
            System.out.println((Prediction)ht.get(gh));
    }
} ///:~
```

Всеки **Groundhog** номер за идентичност, така че може да търсите **Prediction** в **HashMap** като кажете “Дай ми **Prediction** асоцииран с **Groundhog** номер 3.” Класът **Prediction** съдържа **boolean** което се инициализира чрез **Math.random()** и **toString()** което интерпретира резултата. В **main()** **HashMap** се попълва с **Groundhog**ове и асоциираните им **Prediction**и. **HashMap** се извежда така че да видите с какво е бил запълнен. Тогава **Groundhog** с номер за идентичност 3 се използва за намиране на предсказание с **Groundhog** номер 3.

Изглежда доста просто, но не работи. Проблемът е в това, че **Groundhog** е наследено от общия коренов клас **Object** (което се случва като не споменете базов клас, така всички класове непременно наследяват **Object**). Използва се метода на **Object hashCode()** за генерация на хаш код за всеки обект, а по подразбиране за тази цел се взема адреса на обекта. Така първият екземпляр на **Groundhog(3)** не дава хаш код като втория екземпляр на **Groundhog(3)** който се опитахме да използваме за намирането.

Може да изглежда, че всичко което трябва да се направи е да се напише код за подтискането на **hashCode()**. Но и така няма да работи докато не направите още нещо: да подтиснете **equals()** което също е част от **Object**. Този метод се използва от **HashMap** когато се опитвате да определите дали един ключ е равен на друг ключ. По-нататък, **Object.equals()** просто сравнява адресите на обекти, така че единият **Groundhog(3)** не е равен на другия **Groundhog(3)**.

Вижда се че за да се използват собствени класове за ключове в **HashMap** трябва да се подтиснат както **hashCode()** така и **equals()**, както е показано в следното решение за програмата:

```
//: c08:SpringDetector2.java
// If you create a class that's used as a key in
// a HashMap, you must override hashCode()
// and equals().
import java.util.*;

class Groundhog2 {
    int ghNumber;
    Groundhog2(int n) { ghNumber = n; }
    public int hashCode() { return ghNumber; }
    public boolean equals(Object o) {
        return (o instanceof Groundhog2)
            && (ghNumber == ((Groundhog2)o).ghNumber);
    }
}

public class SpringDetector2 {
    public static void main(String[] args) {
        HashMap ht = new HashMap();
        for(int i = 0; i < 10; i++)
            ht.put(new Groundhog2(i),new Prediction());
        System.out.println("ht = " + ht + "\n");
        System.out.println(
            "Looking up prediction for groundhog #3:");
        Groundhog2 gh = new Groundhog2(3);
        if(ht.containsKey(gh))
            System.out.println((Prediction)ht.get(gh));
    }
} ///:~
```

Забележете че се използва класа **Prediction** от предишния пример, та **SpringDetector.java** трябва да се компилира първо иначе ще се получи грешка по време на компилацията на **SpringDetector2.java**.

Groundhog2.hashCode() връща ground hog номера като идентификатор. (В този пример програмистът е отговорен да осигури че няма два с един и същ идентификационен номер.) **hashCode()** не е необходимо за връщане на уникален идентификатор, но метода **equals()** трябва да е в състояние твърдо да установи дали два обекта са равни.

Въпреки че наглед **equals()** метода само проверява дали аргументът е екземпляр на **Groundhog2** (използвайки ключовата дума **instanceof** която е напълно описана в глава 11), **instanceof** фактически прави тихичко втора санитарна проверка дали обектът е **null**, понеже **instanceof** дава **false** ако левият аргумент е **null**. Предполагайки че правилният тип не е **null**, сравняването става по **ghNumbers**. Този път, като пуснете програмата, ще видите че тя дава коректен изход. (Много от класовете в библиотеките на Java подтискат **hashCode()** и **equals()** методите за работа с конкретното съдържание на библиотеките.)

Properties: тип **HashMap**

В първия пример в тази книга беше използван един тип **HashMap**, а именно **Properties**. Във въпросния пример редовете:

```
Properties p = System.getProperties();
p.list(System.out);
```

извикваха **static** метода **getProperties()** да вземе специален **Properties** който описващо характеристиките на системата. Метода **list()** е метод на **Properties** който изпраща съдържанието към който и да е избран изходен поток. Има също **save()** метод за да се запише списък от характеристики във файл така че по-късно да бъде интерпретиран от **load()** метода.

Въпреки че класът **Properties** е наследен от **HashMap**, той също съдържа втори **HashMap** който владее обектите с характеристики "по подразбиране". Така че ако дадена характеристика не е намерена в основния лист, потърсва се такава по подразбиране.

Класът **Properties** може също да се използва във вашите програми (примерът **ClassScanner.java** в глава 17). Може да намерите повече подробности в документацията на Java библиотеката.

Пак енумератори

Сега може да демонстрираме истинската мощ на **Iterator**: способността да се проучи последователност без да се засяга структурата на тази последователност. В следващия пример класът **PrintData** използва **Iterator** за придвижване през последователността и извикване на **toString()** метода за всеки обект. Създадени са два различни типа колекции, **Vector** и **HashMap**, те са попълнени с, респективно, **Mouse** и **Hamster** обекти. (Тези класове са дефинирани по-рано в тази глава; забележете че трябва да сте компилирали **HamsterMaze.java** и **WorksAnyway.java** за да може да се компилира следващата програма.) Понеже **Iterator** скрива структурата на подлежащата колекция, **PrintData** не трябва да знае или да се грижи от каква колекция идва **Iterator**:

```
//: c08:Iterators2.java
// Revisiting Iterators
import java.util.*;

class PrintData {
    static void print(Iterator e) {
        while(e.hasNext())
            System.out.println(
                e.next().toString());
    }
}

class Enumerators2 {
    public static void main(String[] args) {
        ArrayList v = new ArrayList();
        for(int i = 0; i < 5; i++)
            v.add(new Mouse(i));

        HashMap h = new HashMap();
        for(int i = 0; i < 5; i++)
            h.put(new Integer(i), new Hamster(i));

        System.out.println("ArrayList");
    }
}
```

```

PrintData.print(v.iterator());
System.out.println("HashMap");
PrintData.print(h.entrySet().iterator());
}
} //:~

```

Забележете че **PrintData.print()** се възпроизвежда от факта че обектите в колекциите са от типа **Object** така че може да се вика **toString()**. По-вероятно е във вашия решаван проблем да направите предположението че **Iterator** се разхожда из колекция от някакъв определен тип. Например бихте могли да предположите че всичко е **Shape** с метод **draw()**. Тогава трябва да направите даункаст от **Object** така че връщанията от **Iterator.next()** да дават **Shape**.

Сортиране

Едни от нещата липсващи в библиотеките на Java 1.0 и 1.1 са алгоритмичните операции, даже простото сортиране. Така че има смисъл да се създаде **ArrayList** който да може да сортира себе си чрез класическия Quicksort.

Един проблем при писането на родов код е, че сортирането е специфично за типа на сортираните неща. Разбира се, един подход е да се напише отделен метод за сортиране за всеки отделен тип, но би трябвало вече да можете да кажете, че това няма да доведе до лесен за повторно използване с други типове код.

Основна задача на проектирането на програми е да се "разделят нещата които се променят от тези които остават същите," а тук кодът който остава същия е общия сортиращ алгоритъм, променят се нещата, които има да се сортират. Така че вместо да се кодира твърдо алгоритъмът в множество сортиращи програми ще се използва техниката на *callback*. С техниката на обратното извикване кодът който се променя се запечатва (капсулира) в негов собствен клас, а частта от кода която остава същата го вика обратно при нужда. По този начин може да се направят различни обекти да изразяват различни начини на кодиране, а да се захроят от един и същ код.

Следващият **interface** описва как да се сравняват два обекта, капсулирайки с това "нещата които се променят" за този конкретен проблем:

```

//: c08:Compare.java
// Interface for sorting callback:
package c08;

interface Compare {
    boolean lessThan(Object lhs, Object rhs);
    boolean lessThanOrEqual(Object lhs, Object rhs);
}
} //:~

```

За двета метода **lhs** представя "левия" обект и **rhs** представя "десния" обект при сравняването.

Един подклас на **ArrayList** може да създадем който да реализира Quicksort чрез **Compare**. Алгоритъмът, който е известен заради скоростта си, няма да бъде обясняван тук. За подробности виж *Practical Algorithms for Programmers*, от Binstock & Rex, Addison-Wesley 1995.

```

//: c08:SortList.java
// A generic sorting list
package c08;
import java.util.*;

public class SortList extends ArrayList {

```

```

private Compare compare; // To hold the callback
public SortList(Compare comp) {
    compare = comp;
}
public void sort() {
    quickSort(0, size() - 1);
}
private void quickSort(int left, int right) {
    if(right > left) {
        Object o1 = get(right);
        int i = left - 1;
        int j = right;
        while(true) {
            while(compare.lessThan(
                get(++i), o1))
            ;
            while(j > 0)
                if(compare.lessThanOrEqual(
                    get(--j), o1))
                    break; // out of while
            if(i >= j) break;
            swap(i, j);
        }
        swap(i , right);
        quickSort(left, i-1);
        quickSort(i+1, right);
    }
}
private void swap(int loc1, int loc2) {
    Object tmp = get(loc1);
    set(loc1, get(loc2));
    set(loc2, tmp);
}
} //:~

```

Сега може да се види основанието за термина “callback,” щом методът **quickSort()** “извиква обратно” методите в **Compare**. Може също да видите как тази техника дава родов, лесен за повторно използване код.

За да се използва **SortList** трябва да се създаде клас реализиращ **Compare** за вида обекти които ще сортирате. Това е място където вътрешен клас не е основното нещо, но може да подобри организацията на кода. Ето пример за **String** обекти:

```

//: c08:StringSortTest.java
// Testing the generic sorting ArrayList
package c08;
import java.util.*;

public class StringSortTest {
    static class StringCompare implements Compare {
        public boolean lessThan(Object l, Object r) {
            return ((String)l).toLowerCase().compareTo(
                ((String)r).toLowerCase()) < 0;
        }
        public boolean
        lessThanOrEqual(Object l, Object r) {
            return ((String)l).toLowerCase().compareTo(

```

```

        ((String)r).toLowerCase() <= 0;
    }
}

public static void main(String[] args) {
    SortList sv =
        new SortList(new StringCompare());
    sv.add("d");
    sv.add("A");
    sv.add("C");
    sv.add("c");
    sv.add("b");
    sv.add("B");
    sv.add("D");
    sv.add("a");
    sv.sort();
    Iterator e = sv.iterator();
    while(e.hasNext())
        System.out.println(e.next());
}
} //:~

```

Вътрешният клас е **static** понеже не е необходима връзка с външния клас за да работи.

Може да се види как, щом веднъж е създадена работната рамка, лесно може да се използва повторно код като този – просто пишете клас който капсулира “нещата които се променят” и давате обекта на **SortList**.

Сравнението прави стринговете в долн регистър, така че голямо **A** се нарежда до малко **a** и не заема никакво отделно място. Този пример показва, обаче, малка недостатъчност на този подход, а именно нарежда малките и големи разновидности на една буква по реда на появяването им: A a b B c C d D. Това обикновено не е голям проблем, понеже обикновено се работи с по-дълги стрингове и ефектът не проличава. (Колекциите на Java 2 дава функционалност на стринговете която решава този проблем.)

Наследяването (**extends**) е използвано тук за създаване на нов тип от **ArrayList** – тоест, **SortList** е **ArrayList** с добавена функционалност. Използването на наследяване тук е мощен метод, но поражда проблеми. Оказва се, че някои методи са **final** (описано в глава 7), тако че не може да се подтиснат. Ако искате да създадете сортиран **ArrayList** който приема и поражда само **String** обекти удряте на камък, понеже **add()** и **elementAt()** са **final**, а точно те са методите които трябва да се подтиснат, за да приемат само **String** обекти. Няма късмет тук.

От друга страна, да видим с композиция: слагането на обект *вътре* в нов клас. Наместо да пренаписваме горния код за да изпълним това, може просто да използваме **SortList** *вътре* в новия клас. В този случай вътрешният клас за реализация на **Compare** ще бъде създаден анонимно:

```

//: c08:StrSortList.java
// Automatically sorted ArrayList that
// accepts and produces only Strings
package c08;
import java.util.*;

public class StrSortList {
    private SortList v = new SortList(
        // Anonymous inner class:
        new Compare() {
            public boolean
            lessThan(Object l, Object r) {

```

```

        return
            ((String)l).toLowerCase().compareTo(
                ((String)r).toLowerCase()) < 0;
    }
    public boolean
    lessThanOrEqual(Object l, Object r) {
        return
            ((String)l).toLowerCase().compareTo(
                ((String)r).toLowerCase()) <= 0;
    }
}
);
private boolean sorted = false;
public void add(String s) {
    v.add(s);
    sorted = false;
}
public String get(int index) {
    if(!sorted) {
        v.sort();
        sorted = true;
    }
    return (String)v.get(index);
}
public Iterator iterator() {
    if(!sorted) {
        v.sort();
        sorted = true;
    }
    return v.iterator();
}
// Test it:
public static void main(String[] args) {
    StrSortList sv = new StrSortList();
    sv.add("d");
    sv.add("A");
    sv.add("C");
    sv.add("c");
    sv.add("b");
    sv.add("B");
    sv.add("D");
    sv.add("a");
    Iterator e = sv.iterator();
    while(e.hasNext())
        System.out.println(e.next());
}
} //:~

```

Това бързичко използва пак кода от **SortList** за да се създаде исканата функционалност. Обаче не всичките **public** методи от **SortList** и **ArrayList** се появяват в **StrSortList**. Когато използваме код по този начин може да направим дефиниция за всеки в новия клас, а може и да започнем с няколко и периодично да се връщаме и да добавяме нови. В края на краишата новият клас ще улегне.

Предимството на този дизайн е че ще взема само **String** обекти и ще произвежда само **String** обекти и проверката става по време на компилация а не по време на изпълнението. Разбира се това се отнася само за **add()** и **elementAt()**; **elements()** все още дава **Iterator** който няма

определен тип по време на компилация. Проверката на типа за **Iterator** и в **StrSortList** се прави, разбира се, това става по време на изпълнение и се изхвърля изключение ако нещо не е наред. Това е цената: установявате ли нещо със сигурност по време на компилация или вероятно по време на изпълнение? (Тоест, “вероятно не когато тествате кода” и “вероятно когато потребителят прави нещо, с което не сте тествали.”) Като имате възможностите за избор и суматохата, по-лесно е да използвате наследяване и да издържите на кастинга – още веднъж, ако параметризираните типове се добавят към Java някой ден те ще решат този проблем.

Може да се види че в този клас има флаг наречен **sorted**. Бихте могли да сортирате **ArrayList** всеки път когато се вика **add()** и той винаги ще се поддържа сортиран. Но обикновено се добавят множество елементи към **ArrayList** преди започване на четенето му. Така сортирането след всеки **add()** ще бъде по-малко ефективно отколкото дза се изчака някой да иска да чете **ArrayList** и тогава да се сортира, както е направено тук. Техниката на отлагане на даден процес докато той стане абсолютно необходим се нарича **мързеливо изчисление**. (Има аналогична техника наречена **мързелива инициализация** която чака дадено поле да стане необходимо за използване, та тогава го инициализира.)

Колекциите на Java 2

За мен колекциите са един от най-мощните инструменти за програмиране. Може да сте забелязали, че съм малко разочарован от колекциите на Java до версия 1.1. Като резултат огромно удоволствие бе да видя, че в Java 2 на колекциите бе обърнато необходимото внимание, те бяха цялостно преработени (от Joshua Bloch от Sun). Считам че колекциите на Java 2 са едната главна черта на версията (другата е Swing библиотеката, разгледана в глава 13) понеже те значително увеличават програмистките мускули и извеждат Java на една линия с по-утвърдени програмни системи.

Част от преработката прави нещата по-свързани и смислени. Например много имена са по-къси, по-изразителни, по-лесни за разбиране и четене, както и за писане. Някои имена са променени за съгласуване с приетата терминология: мой любим пример е “*iterator*” вместо “*enumeration*.”

Преработена е също функционалността на библиотеката на колекциите. Сега може да се получи поведение на свързан списък, опашки и декове (опашки с два края).

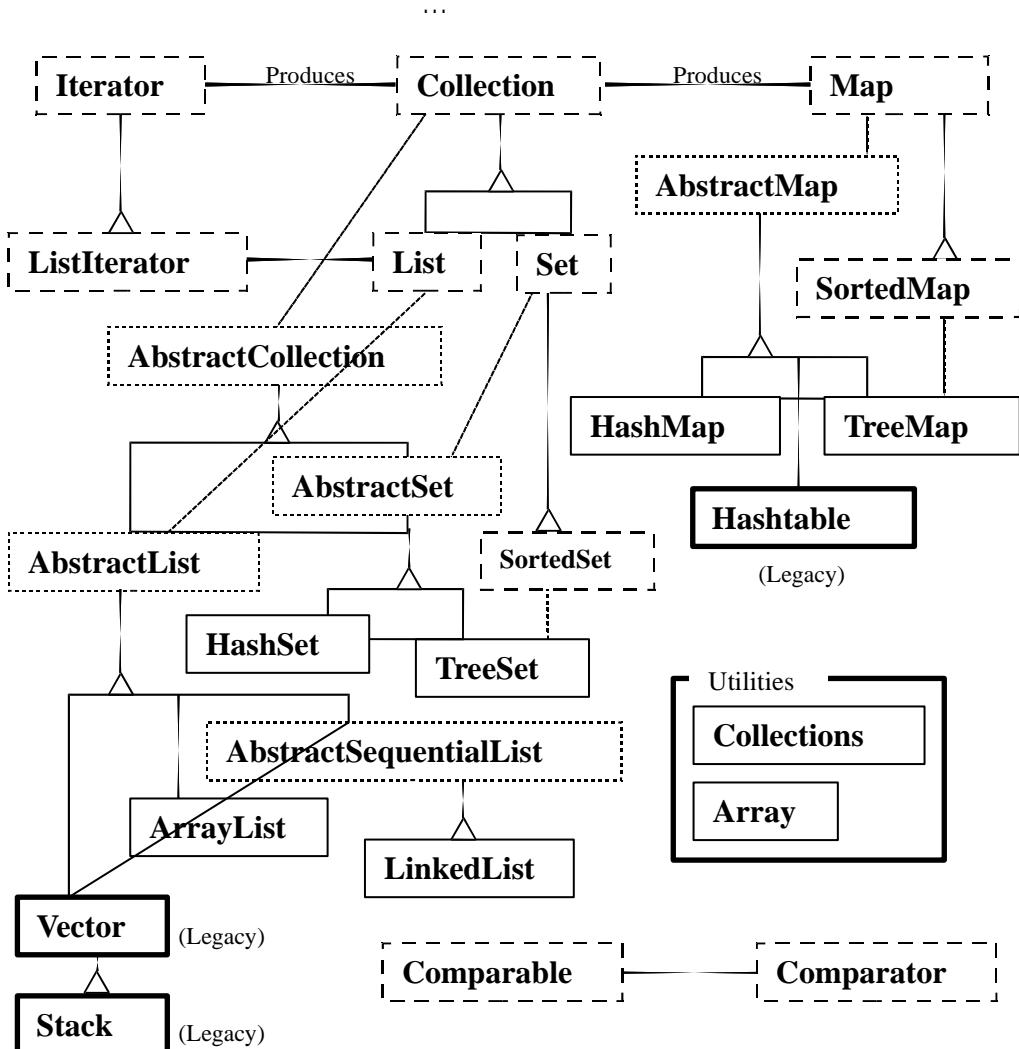
Проектирането на библиотека колекции е трудно (както повечето проблеми свързани с библиотеки). В C++ STL се основава на много различни класове. Това е по-добре от наличното преди STL (нищо), но не се вписваше добре в Java. Резултатът беше малко смущаващо блато от класове. В другата крайност аз съм виждал библиотека, състояща се от един клас, “collection,” който работи като **ArrayList** и **HashMap** едновременно. Проектантите на колекциите в библиотеката на Java 2 искаха да постигнат баланс: пълната функционалност, която се иска от редовна библиотека, но по-голяма леснота за използване отколкото STL и други подобни библиотаки. Резултатът може да изглежда малко недобър на места. За разлика от решенията приети в ранните Java тези неуспехи не са нещастни случаи, а внимателно подбрани компромиси с цената на ефективността. Може да отнеме повече време да свикнете с някои аспекти на библиотеките, но мисля че бързо ще приемете и започнете да използвате тези нови инструменти.

Колекциите на Java 2 библиотеката вземат въпроса за “владеене на вашите обекти” и го разделят между две различни концепции:

1. **Collection**: група от отделни елементи, често с прилагане на някакво правило към тях. **List** трябва да държи елементите на конкретна последователност, а **Set** не може да има дублиращи се елементи. (*Bag*, което не е реализирано в Collections библиотеката на Java 2 поради това, че **List**овете дават тази функционалност, няма такива правила.)

2. **Map**: група от двойки ключ-стойност (каквите сте виждали досега като **Hashtable**). На пръв поглед това трябва да бъдат **Collection**и от двойки, но когато се опитате да го реализирате по този начин става тромаво, така че е по-добре да бъде отделна концепция. От друга страна, удобно е да се преглеждат части от **Map** чрез създаване на **Collection** да представя съответната порция. Така **Map** може да връща **Set** от своите ключове, **List** от своите стойности или **List** от своите двойки. **Маровете**, както масивите, могат лесно да бъдат разширени до нови стойности без необходимост от нови концепции: просто правите **Map** чиито стойности са **Марове** (и стойностите на тези **Марове** могат да бъдат **Марове** и т.н.).

Collectionите и **Маровете** могат да бъда прилагани по много различни начини, съответно на програмистките нужди. Полезно е да се погледне диаграмата на колекциите на Java 2:



Тази диаграма може да изглежда замайващо сложна на пръв поглед, но до края ня главата ще видите, че има само три класа колекции: **Map**, **List** и **Set** и само две или три реализации на всяка една (с, типично, предпочитана версия). Когато това стане колекциите на Java 2 не би трявало да изглеждат толкова заплашително.

Кутийките с прекъсната линия представляват **interface**ите, с линия от точки — **abstract** класовете, а с непрекъсната линия са нормалните (конкретни) класове. Стрелките с прекъсната линия показват че даден клас прилага даден **interface** (или в случая на **abstract** клас, частично прилага този **interface**). Стрелките с двойна линия показват, че може да се правят обекти от същия клас. Например всяка **Collection** може да прави **Iterator**, докато **List** може да прави **ListIterator** (както и обикновен **Iterator**, тъй като **List** е наследен от **Collection**).

Интерфейсите свързани с владеенето на обекти са **Collection**, **List**, **Set** и **Map**. Типично ще е да пишете по-голямата част от кода си да разговаря с тези интерфейси, а единственото място където точния тип ще е необходим да бъде точката на създаване на обект. Така че може да създавате **List** по този начин:

```
| List x = new LinkedList();
```

Разбира се, бихте могли да решите да направите **x LinkedList** (вместо родов **List**) и да дадете точна информация за типа на **x**. Красотата (и целта) на използването на **interface** е че ако решите да промените реализацията, всичко което ще е необходимо е да смените типа в точката на създаването, както тук:

```
| List x = new ArrayList();
```

Останалата част от кода може да остане незасегната.

В йерархията на класовете може да се видят множество класове, чиято декларация започва с **"Abstract,"** а това може да е смущаващо отначало. Те са просто инструменти, които частично прилагат даден интерфейс. Ако правехте ваш собствен **Set**, например, не бихте започнали с интерфейса на **Set** и реализация на всички методи, вместо това бихте наследили от **AbstractSet** и сторили минимално необходимото за създаване на новия клас. Обаче Колекциите в библиотеката на Java 2 съдържат достатъчно функционалност за удовлетворяване на нуждите ви практически за всичко. Така че за вашите цели бихте могли да игнорирате всичко, което започва с **"Abstract."**

Поради това, както гледате диаграмата, ви засягат само онези **interface**и от горната част на диаграмата и конкретните класове (онези с непрекъсната линия). Типично ще правите обекти от конкретни класове, ъпкаст към съответния **interface**, а после ще използвате **interface**-а в останалата част от вашия код. Ето прост пример, който попълва **Collection** със **String** обекти и после извежда всеки обект от тази **Collection**:

```
//: c08:newcollections:SimpleCollection.java
// A simple example using Java 2 Collections
package c08.newcollections;
import java.util.*;

public class SimpleCollection {
    public static void main(String[] args) {
        Collection c = new ArrayList();
        // Upcast because we just want to
        // work with Collection features
        for(int i = 0; i < 10; i++)
            c.add(Integer.toString(i));
        Iterator it = c.iterator();
        while(it.hasNext())
            System.out.println(it.next());
    }
} ///:~
```

Всичките примери за Java 2 Collections библиотеките ще трябва да се сложат в поддиректория **newcollections**, така че да се напомни че ще работят само с Java 2. Като резултат може да стартирате програмата така:

```
| java c08.newcollections.SimpleCollection
```

с подобен синтаксис за другите програми от главата.

Може да се види че Java 2 Collections са част от **java.util** библиотеката, така че няма нужда от никакви допълнителни **import** оператори за използването им.

Първата линия в **main()** създава **ArrayList** и после го превръща към **Collection**. Тъй като този пример използва само методите на **Collection** всеки обект наследен от **Collection** би работил, но **ArrayList** е типичният работен кон на **Collection** и заема мястото на **Vector**.

Методът **add()**, както показва името му, добавя нов елемент към **Collection**. Обаче документацията грижливо твърди че **add()** "осигурява че тази Collection съдържа посочения елемент." Това е за да се позволи в смисъла на **Set**, който добавя елемент само ако още няма такъв добавен. С **ArrayList**, или който и да е сорт **List**, **add()** винаги значи "пъхни го вътре."

Всички **Collection** могат да дадат **Iterator** чрез техния **iterator()** метод. **Iterator** е точно като **Enumeration**, който замества, освен че:

1. Използва име (**iterator**) което е исторически признато и прието в ООП общността.
2. Използва по-къси имена на методи от **Enumeration**: **hasNext()** вместо **hasMoreElements()**, **next()** вместо **nextElement()**.
3. Въвежда нов метод, **remove()**, който маха последния елемент даден от **Iterator**. Така че може да извикате **remove()** само веднъж сле всяко извикване на **next()**.

В **SimpleCollection.java** може да видите че **Iterator** е създаден и използван за работа с **Collection**, извеждайки всеки елемент.

Използване на **Collection**и

Следващата таблица показва какво може да се направи с **Collection**, а по този начин и какво може да се направи със **Set** или **List**. (**List** има и друга функционалност.) **Map**овете не са наследени от **Collection** и ще се разгледат отделно.

Boolean add(Object)	*Осигурява че Collection съдържа аргумента. Връща <code>false</code> ако не е добавило аргумента.
Boolean addAll(Collection)	*Добавя елементите в аргумента. Връща <code>true</code> ако има добавени елементи.
void clear()	*Маха всички елементи в Collection.
Boolean contains(Object)	<code>True</code> ако Collection съдържа аргумента.
Boolean containsAll(Collection)	<code>True</code> ако Collection съдържа всичките елементи в аргумента.
Boolean isEmpty()	<code>True</code> ако Collection няма елементи.
Iterator iterator()	Връща Iterator който може да се използва за обикаляне елементите на Collection.
Boolean remove(Object)	*Ако аргументът е в Collection, маха се един негов екземпляр. Връща <code>true</code> ако е имало премахване.
Boolean removeAll(Collection)	*Маха всички елементи които се съдържат в аргумента. Връща <code>true</code> ако е имало махане.
Boolean retainAll(Collection)	*Връща само елементите които се съдържат в аргумента ("intersection" от теорията на множествата). Връща <code>true</code> ако е имало промяна.
int size()	Връща броя на елементите в Collection.
Object[] toArray()	Връща масив съдържащ всичките

	елементи на Collection.
Object() toArray(Object[] a)	<p>Връща масив съдържащ всички елементи на Collection, чийто тип съвпада с този на масива a а не е просто Object (необходимо е да се превърне масива в правилния тип).</p> <p>*Това е метод „по избор“ което значи че може и да не е реализиран от конкретна Collection. Ако не е, методът изхвърля UnsupportedOperationException. Изключениета са описани в глава 9.</p>

Следващият пример демонстрира всички тези методи. Отново, те работят с всяко нещо наследено от **Collection**; **ArrayList** е използван като един вид „най-малък общ знаменател“:

```
//: c08:newcollections:Collection1.java
// Things you can do with all Collections
package c08.newcollections;
import java.util.*;

public class Collection1 {
    // Fill with 'size' elements, start
    // counting at 'start':
    public static Collection
    fill(Collection c, int start, int size) {
        for(int i = start; i < start + size; i++)
            c.add(Integer.toString(i));
        return c;
    }
    // Default to a "start" of 0:
    public static Collection
    fill(Collection c, int size) {
        return fill(c, 0, size);
    }
    // Default to 10 elements:
    public static Collection fill(Collection c) {
        return fill(c, 0, 10);
    }
    // Create & upcast to Collection:
    public static Collection newCollection() {
        return fill(new ArrayList());
        // ArrayList is used for simplicity, but it's
        // only seen as a generic Collection
        // everywhere else in the program.
    }
    // Fill a Collection with a range of values:
    public static Collection
    newCollection(int start, int size) {
        return fill(new ArrayList(), start, size);
    }
    // Moving through a List with an iterator:
    public static void print(Collection c) {
        for(Iterator x = c.iterator(); x.hasNext();)
            System.out.print(x.next() + " ");
        System.out.println();
    }
}
```

```

public static void main(String[] args) {
    Collection c = newCollection();
    c.add("ten");
    c.add("eleven");
    print(c);
    // Make an array from the List:
    Object[] array = c.toArray();
    // Make a String array from the List:
    String[] str =
        (String[])c.toArray(new String[1]);
    // Find max and min elements; this means
    // different things depending on the way
    // the Comparable interface is implemented:
    System.out.println("Collections.max(c) = " +
        Collections.max(c));
    System.out.println("Collections.min(c) = " +
        Collections.min(c));
    // Add a Collection to another Collection
    c.addAll(newCollection());
    print(c);
    c.remove("3"); // Removes the first one
    print(c);
    c.remove("3"); // Removes the second one
    print(c);
    // Remove all components that are in the
    // argument collection:
    c.removeAll(newCollection());
    print(c);
    c.addAll(newCollection());
    print(c);
    // Is an element in this Collection?
    System.out.println(
        "c.contains(\"4\") = " + c.contains("4"));
    // Is a Collection in this Collection?
    System.out.println(
        "c.containsAll(newCollection()) = " +
        c.containsAll(newCollection()));
    Collection c2 = newCollection(5, 3);
    // Keep all the elements that are in both
    // c and c2 (an intersection of sets):
    c.retainAll(c2);
    print(c);
    // Throw away all the elements in c that
    // also appear in c2:
    c.removeAll(c2);
    System.out.println("c.isEmpty() = " +
        c.isEmpty());
    c = newCollection();
    print(c);
    c.clear(); // Remove all elements
    System.out.println("after c.clear():");
    print(c);
}
} //:~

```

Първият метод дава начин да се запълни всяка **Collection** с тестови данни, в този случай **int** превърнати в **String**. Вторият метод често ще бъде използван по протежение на тази глава.

Двете версии на **newCollection()** създават **ArrayList**ове съдържащи различни множества от данни и ги връщат като **Collection** обекти, така че е ясно че не се използва нищо освен интерфейса на **Collection**.

Методът **print()** също ще бъде използван в тази глава. Тъй като се придвижва през **Collection** чрез **Iterator**, който всяка **Collection** може да направи, той ще работи с **List**ове и **Set**ове и всяка **Collection** колекция създадена от **Map**.

main() използва прости задачи за показване на всичките методи на **Collection**.

Следващите секции дават сравнение на различните реализации на **List**, **Set** и **Map** и подсказват във всеки един случай (със звездичка) кое да бъде вашият избор по подразбиране. Ще видите че наследените класове **Vector**, **Stack** и **Hashtable** не са включени понеже във всички случаи те са предпочитаните класове в Java 2 Collections.

Използване на **List**ове

List (interface))	Най-важната черта на един List е редът; той обещава да обработва елементите в конкретен ред. List добавя методи към Collection които позволяват вмъкване и махане на елементи и в средата на List . (Това се препоръчва само за LinkedList .) List ще направи ListIterator , а използвайки го може да извървите List а в двете посоки, както и да вмъквате и махате елементи в средата на листа (отново, препоръчва се само за LinkedList).
ArrayList *	List подкрепен от масив. Използва се вместо Vector за обща употреба при владеенето на обекти. Позволява бърз достъп до произволен елемент, но е бавен когато се вмъкват и изтриват елементи от средата. ListIterator трябва да се използва само за назад-напред разходки по ArrayList , не и за вмъкване и махане на елементи, което е скъпо сравнено с LinkedList .
LinkedList	Дава оптимален последователен достъп, с несъкъпи вмъквания и изтривания от средата на списъка. Относително бавен за произволен достъп. (Използвайте ArrayList вместо него.) Има също addFirst() , addLast() , getFirst() , getLast() , removeFirst() и removeLast() (които не са дефинирани в никой интерфейс или базов клас) за да позволи да се използва като стек, опашка и дек.

Всеки метод в следващия пример покрива различна група активности: такива каквито всеки списък може (**basicTest()**), разходки наоколо с **Iterator** (**iterMotion()**) versus промени чрез **Iterator** (**iterManipulation()**), разглеждане на резултата от манипулирането на **List** (**testVisual()**) и операции налични само за **LinkedList**ове.

```
//: c08:newcollections:List1.java
// Things you can do with Lists
package c08.newcollections;
import java.util.*;

public class List1 {
```

```

// Wrap Collection1.fill() for convenience:
public static List fill(List a) {
    return (List)Collection1.fill(a);
}

// You can use an Iterator, just as with a
// Collection, but you can also use random
// access with get():
public static void print(List a) {
    for(int i = 0; i < a.size(); i++)
        System.out.print(a.get(i) + " ");
    System.out.println();
}

static boolean b;
static Object o;
static int i;
static Iterator it;
static ListIterator lit;
public static void basicTest(List a) {
    a.add(1, "x"); // Add at location 1
    a.add("x"); // Add at end
    // Add a collection:
    a.addAll(fill(new ArrayList()));
    // Add a collection starting at location 3:
    a.addAll(3, fill(new ArrayList()));
    b = a.contains("1"); // Is it in there?
    // Is the entire collection in there?
    b = a.containsAll(fill(new ArrayList()));
    // Lists allow random access, which is cheap
    // for ArrayList, expensive for LinkedList:
    o = a.get(1); // Get object at location 1
    i = a.indexOf("1"); // Tell index of object
    b = a.isEmpty(); // Any elements inside?
    it = a.iterator(); // Ordinary Iterator
    lit = a.listIterator(); // ListIterator
    lit = a.listIterator(3); // Start at loc 3
    i = a.lastIndexOf("1"); // Last match
    a.remove(1); // Remove location 1
    a.remove("3"); // Remove this object
    a.set(1, "y"); // Set location 1 to "y"
    // Keep everything that's in the argument
    // (the intersection of the two sets):
    a.retainAll(fill(new ArrayList()));
    // Remove everything that's in the argument:
    a.removeAll(fill(new ArrayList()));
    i = a.size(); // How big is it?
    a.clear(); // Remove all elements
}

public static void iterMotion(List a) {
    ListIterator it = a.listIterator();
    b = it.hasNext();
    b = it.hasPrevious();
    o = it.next();
    i = it.nextIndex();
    o = it.previous();
    i = it.previousIndex();
}

public static void iterManipulation(List a) {

```

```

ListIterator it = a.listIterator();
it.add("47");
// Must move to an element after add():
it.next();
// Remove the element that was just produced:
it.remove();
// Must move to an element after remove():
it.next();
// Change the element that was just produced:
it.set("47");
}
public static void testVisual(List a) {
print(a);
List b = new ArrayList();
fill(b);
System.out.print("b = ");
print(b);
a.addAll(b);
a.addAll(fill(new ArrayList()));
print(a);
// Insert, remove, and replace elements
// using a ListIterator:
ListIterator x = a.listIterator(a.size()/2);
x.add("one");
print(a);
System.out.println(x.next());
x.remove();
System.out.println(x.next());
x.set("47");
print(a);
// Traverse the list backwards:
x = a.listIterator(a.size());
while(x.hasPrevious())
    System.out.print(x.previous() + " ");
System.out.println();
System.out.println("testVisual finished");
}
// There are some things that only
// LinkedLists can do:
public static void testLinkedList() {
LinkedList ll = new LinkedList();
Collection1.fill(ll, 5);
print(ll);
// Treat it like a stack, pushing:
ll.addFirst("one");
ll.addFirst("two");
print(ll);
// Like "peeking" at the top of a stack:
System.out.println(ll.getFirst());
// Like popping a stack:
System.out.println(ll.removeFirst());
System.out.println(ll.removeFirst());
// Treat it like a queue, pulling elements
// off the tail end:
System.out.println(ll.removeLast());
// With the above operations, it's a dequeuel
print(ll);

```

```

}
public static void main(String args()) {
    // Make and fill a new list each time:
    basicTest(fill(new LinkedList()));
    basicTest(fill(new ArrayList()));
    iterMotion(fill(new LinkedList()));
    iterMotion(fill(new ArrayList()));
    iterManipulation(fill(new LinkedList()));
    iterManipulation(fill(new ArrayList()));
    testVisual(fill(new LinkedList()));
    testLinkedList();
}
} //:~

```

В **basicTest()** и **iterMotion()** виканията са сложени просто да се покаже правилният синтаксис, а ако и да е хваната връщаната стойност, тя не се използва. В някои случаи връщаната стойност не се хваща, понеже типично не се използва. Ще разгледате пълната документация за тези методи преди да ги използвате.

Използване на Setове

Set има точно същия интерфейс като **Collection**, така че няма допълнителна функционалност както с двата различни **List**а. **Set** е точно **Collection**, той има само различно поведение. (Това е идеалната употреба на наследяването и полиморфизма: да се изрази различно поведение.) **Set** позволява само един екземпляр от всяка стойност на обект да съществува (което конституира "стойноста" по-сложно, както ще видите).

Set (interface)	Всеки елемент който се добавя към Set трябва да бъде уникален; иначе Set не добавя дублицирана елемент. Обектите добавени към Set трябва да дефинират equals() за да има уникалност на обекти. Set има точно същия интерфейс като Collection . Интерфейсът на Set не гарантира че обектите ще бъдат в някакъв определен ред.
HashSet*	За Setове където бързото търсене е важно. Обектите трябва да дефинират също и hashCode() .
TreeSet	Подреден Set подкрепен от червено-черно дърво. По този начин може да извлечете подредена последователност от Set .

Следния пример не показва всичко което може да се направи със **Set**, тъй като интерфейсът е като на **Collection** и беше изпитан в предишния пример. Вместо това се демонстрира поведението, което прави **Set** уникален:

```

//: c08:newcollections:Set1.java
// Things you can do with Sets
package c08.newcollections;
import java.util.*;

public class Set1 {
    public static void testVisual(Set a) {
        Collection1.fill(a);
        Collection1.fill(a);
        Collection1.fill(a);
        Collection1.print(a); // No duplicates!
        // Add another set to this one:
    }
}

```

```

a.addAll(a);
a.add("one");
a.add("one");
a.add("one");
Collection1.print(a);
// Look something up:
System.out.println("a.contains(\"one\"): " +
    a.contains("one"));
}
public static void main(String[] args) {
    testVisual(new HashSet());
    testVisual(new TreeSet());
}
} //:~

```

Дублиращите стойности са добавени към **Set**, но когато се извежда ще видите че **Set** е приело само един екземпляр от всяка стойност.

Като пускате тази програма ще видите че редът поддържан от **HashSet** е различен от **TreeSet**, понеже всеки има различен начин на запомняне на елементи които по-късно да бъдат търсени. (**TreeSet** ги държи сортирани, докато **HashSet** използва хеш функция, което е нарочно за бързи търсения.) Като създавате собствени типове ще видите предупредени че **Set** иска начин да поддържа подредба в паметта, точно както с "groundhog" примерите показани по-рано в главата. За да се реализира сравняемост в Java 2 Collections, обаче, трябва да приложите **Comparable** интерфейс и да реализирате **compareTo()** метод (това ще се опише пълно по-късно). Ето пример:

```

//: c08:newcollections:Set2.java
// Putting your own type in a Set
package c08.newcollections;
import java.util.*;

class MyType implements Comparable {
    private int i;
    public MyType(int n) { i = n; }
    public boolean equals(Object o) {
        return
            (o instanceof MyType)
            && (i == ((MyType)o).i);
    }
    public int hashCode() { return i; }
    public String toString() { return i + " "; }
    public int compareTo(Object o) {
        int i2 = ((MyType) o).i;
        return (i2 < i ? -1 : (i2 == i ? 0 : 1));
    }
}

public class Set2 {
    public static Set fill(Set a, int size) {
        for(int i = 0; i < size; i++)
            a.add(new MyType(i));
        return a;
    }
    public static Set fill(Set a) {
        return fill(a, 10);
    }
}

```

```

public static void test(Set a) {
    fill(a);
    fill(a); // Try to add duplicates
    fill(a);
    a.addAll(fill(new TreeSet()));
    System.out.println(a);
}

public static void main(String[] args) {
    test(new HashSet());
    test(new TreeSet());
}
} ///:~

```

Дефинициите за **equals()** и **hashCode()** следват формата дадена в “groundhog” примерите. Трябва да дефинирате **equals()** в двата случая, но **hashCode()** е абсолютно необходим само ако класът ще се слага в **HashSet** (което е вероятно, понеже това ще бъде първото ви предпочтение за реализация на **Set**). Обаче за програмистки стил винаги ще подтискате **hashCode()** когато подтискате **equals()**.

В **compareTo()** забележете че не съм използвал “простата и очевидна форма” **return i-i2**. Макар и това да е честа програмна грешка, тя ще работи само ако **i** и **i2** са **unsigned int** (ако Java имаше ключовата дума “**unsigned**” каквато той няма). Начинът не върви за целите със знак в Java които са недостатъчно големи да представят разликата между две **int**. Ако **i** е голямо положително цяло а **j** е голямо отрицателно число, **i-j** ще препълни и ще върне отрицателна стойност, което е грешка.

Използване на Марове

Map (interface)	Поддържа ключ-стойност асоциации (pairs), така че може да намерите стойност по ключ.
HashMap*	Реализация основана на хеш таблица. (Използвайте го вместо Hashtable .) Дава константно време на изпълнение за вмъкване и изтриване на двойки. Скоростта може да бъде на-гласявана чрез конструктори които позволяват да се задават капацитет и фактор на натоварването за хеш таблицата.
TreeMap	Реализация, основана на червено-бяло дърво. Когато гледате ключове или двойки, те ще бъдат сортирани (определеното от Comparable или Comparator , обсъждани по-късно). Намерението на TreeMap е че вземате резултатите сортирани. TreeMap е единствен Map с метода subMap() кое-то позволява да се върне поддърво.

Следващият пример съдържа две множества от данни и **fill()** метод който позволява да попълните всяка карта с двуразмерен масив от **Objects**. Тези инструменти ще се използват и в други **Map** примери също така.

```

//: c08:newcollections:Map1.java
// Things you can do with Maps
package c08.newcollections;
import java.util.*;

public class Map1 {
    public final static String[] testData1 = {

```

```

    { "Happy", "Cheerful disposition" },
    { "Sleepy", "Prefers dark, quiet places" },
    { "Grumpy", "Needs to work on attitude" },
    { "Doc", "Fantasizes about advanced degree" },
    { "Dopey", "A' for effort" },
    { "Sneezy", "Struggles with allergies" },
    { "Bashful", "Needs self-esteem workshop" },
};

public final static String[] testData2 = {
    { "Belligerent", "Disruptive influence" },
    { "Lazy", "Motivational problems" },
    { "Comatose", "Excellent behavior" }
};

public static Map fill(Map m, Object[] o) {
    for(int i = 0; i < o.length; i++)
        m.put(o[i](0), o[i](1));
    return m;
}

// Producing a Set of the keys:
public static void printKeys(Map m) {
    System.out.print("Size = " + m.size() + ", ");
    System.out.print("Keys: ");
    Collection1.print(m.keySet());
}

// Producing a Collection of the values:
public static void printValues(Map m) {
    System.out.print("Values: ");
    Collection1.print(m.values());
}

// Iterating through Map.Entry objects (pairs):
public static void print(Map m) {
    Collection entries = m.entrySet();
    Iterator it = entries.iterator();
    while(it.hasNext()) {
        Map.Entry e = (Map.Entry)it.next();
        System.out.println("Key = " + e.getKey() +
            ", Value = " + e.getValue());
    }
}

public static void test(Map m) {
    fill(m, testData1);
    // Map has 'Set' behavior for keys:
    fill(m, testData1);
    printKeys(m);
    printValues(m);
    print(m);
    String key = testData1(4)(0);
    String value = testData1(4)(1);
    System.out.println("m.containsKey(\"" + key +
        "\"): " + m.containsKey(key));
    System.out.println("m.get(\"" + key + "\"): " +
        + m.get(key));
    System.out.println("m.containsValue(\"" +
        + value + "\"): " +
        m.containsValue(value));
    Map m2 = fill(new TreeMap(), testData2);
    m.putAll(m2);
}

```

```

printKeys(m);
m.remove(testData2(0)(0));
printKeys(m);
m.clear();
System.out.println("m.isEmpty(): "
+ m.isEmpty());
fill(m, testData1);
// Operations on the Set change the Map:
m.keySet().removeAll(m.keySet());
System.out.println("m.isEmpty(): "
+ m.isEmpty());
}
public static void main(String args) {
System.out.println("Testing HashMap");
test(new HashMap());
System.out.println("Testing TreeMap");
test(new TreeMap());
}
} //:~

```

Методите **printKeys()**, **printValues()** и **print()** са не само полезни ютилита, те също демонстрират произвеждането на **Collection** изгледи на **Map**. Методът **keySet()** дава **Set** подкрепена от ключовете в **Map**; тук той се третира само като **Collection**. Подобно е третирането на **values()**, който дава **List** съдържащ всичките стойности в **Map**. (Забележете че ключовете трябва да са уникални, докато стойностите могат да съдържат дубликати.) Тъй като тези **Collection**и са поддържани от **Map**, всяка промени в **Collection** ще рефлектират в асоциирания **Map**.

Методът **print()** грабва **Iterator**a даден от **entries** и го използва за извеждане на ключа и стойността за всяка двойка. Останалата част от програмата дава прости приложения на всяка операция на **Map** и тества всеки тип **Map**.

Когато създавате собствени класове за използване като ключ в **Map** трябва да се справите със същите въпроси дискутирани по-рано във връзка със **Set**овете.

Избиране на реализация

Има фактически само три компонента колекции: **Map**, **List** и **Set** и само две или три реализации за всеки интерфейс. Ако се нуждаете от функционалността предлагана от конкретен **interface**, как да решите коя да изберете?

За да разберете отговора трябва да знаете, че всяка реализация има своите собствени черти, силни страни и слабости. Например може да видите на диаграмата (не е показана-б.пр.) че "чертата" на **Hashtable**, **Vector** и **Stack** е че те са наследени класове, така че съществуващия код няма да се скапва. От друга страна най-добре е да не ги използвате за нов (Java 2) код.

Разликата между колекциите често изчезва от това, което изразяваме с "подкрепяно от;" тоест структурите данни които физически реализират вашия **interface**. Това значи че, например, **ArrayList**, **LinkedList** и **Vector** (което е груба еквивалентност на **ArrayList**) всички прилагат интерфейса на **List** така че вашата програма ще дава едни и същи резултати независимо какво използвате. Обаче **ArrayList** (и **Vector**) се подкрепят от масив, докато **LinkedList** е реализирано по обикновения начин за двойно свързани листове, като отделни обекти съдържащи данните заедно с полета с адресите на предишния и следващия елемент. Поради това ако имате да правите множество вмъквания и изтривания в средата на списъка **LinkedList** е подходящия избор. (**LinkedList** има също допълнителни възможности свързани с **AbstractSequentialList**.) Ако не, **ArrayList** е вероятно по-бърз.

Като друг пример **Set** може да бъде реализиран чрез **TreeSet** или **HashSet**. **TreeSet** се подкрепя от **TreeMap** и е проектиран да поддържа постоянно сортирано множество. Обаче ако ще имате големи количества във вашия **Set**, скоростта на добавянията на **TreeSet** ще стане малка. Когато пишете програма която ще използва **Set** ще използвате **HashSet** по подразбиране, а ще преминете на **TreeSet** когато стане по-важно да имате постоянно сортирано множество.

Избор между **List**ове

Най-убедителният начин да се видят разликите между различните колекции е тествът на скоростта. Следващия код задава вътрешен базов клас който да служи като тестова рамка, после създава анонимен вътрешен клас за всеки отделен тест. Всеки от тези вътрешни класове се вика от метода **test()**. Този подход позволява лесно да се добавят нови видове тест.

```
//: c08:newcollections>ListPerformance.java
// Demonstrates performance differences in Lists
package c08.newcollections;
import java.util.*;

public class ListPerformance {
    private static final int REPS = 100;
    private abstract static class Tester {
        String name;
        int size; // Test quantity
        Tester(String name, int size) {
            this.name = name;
            this.size = size;
        }
        abstract void test(List a);
    }
    private static Tester() tests = {
        new Tester("get", 300) {
            void test(List a) {
                for(int i = 0; i < REPS; i++) {
                    for(int j = 0; j < a.size(); j++)
                        a.get(j);
                }
            }
        },
        new Tester("iteration", 300) {
            void test(List a) {
                for(int i = 0; i < REPS; i++) {
                    Iterator it = a.iterator();
                    while(it.hasNext())
                        it.next();
                }
            }
        },
        new Tester("insert", 1000) {
            void test(List a) {
                int half = a.size()/2;
                String s = "test";
                ListIterator it = a.listIterator(half);
                for(int i = 0; i < size * 10; i++)
                    it.add(s);
            }
        },
    };
}
```

```

new Tester("remove", 5000) {
    void test(List a) {
        ListIterator it = a.listIterator(3);
        while(it.hasNext()) {
            it.next();
            it.remove();
        }
    }
},
};

public static void test(List a) {
    // A trick to print out the class name:
    System.out.println("Testing " +
        a.getClass().getName());
    for(int i = 0; i < tests.length; i++) {
        Collection1.fill(a, tests(i).size);
        System.out.print(tests(i).name);
        long t1 = System.currentTimeMillis();
        tests(i).test(a);
        long t2 = System.currentTimeMillis();
        System.out.println(": " + (t2 - t1));
    }
}
}

public static void main(String[] args) {
    test(new ArrayList());
    test(new LinkedList());
}
} //:~

```

Вътрешният клас **Tester** е **abstract**, за да се получи базов клас за тестовете. Той съдържа **String** за извеждане когато стартира тестът, **size** параметър за да бъде използван за количество елементи или брой повторения на тестове, конструктор за инициализация на полетата и **abstract** метод **test()** който върши работата. Всичките различни типове тестове са събрани на едно място, масива **tests**, който се инициализира с анонимни вътрешни класове наследени от **Tester**. За да се добавят или мащнат тестове просто добавете или мащнете дефиниция на вътрешен клас в масива, а всичко останало ще стане автоматично.

Listът предаден на **test()** първо се попълва с елементи, после всеки тест в **tests** се замерва. Резултатите ще се менят от машина на машина; те дават само сравнение между скоростта на различните измервани неща. Ето резюме от едно пускане:

Type	Get	Iteration	Insert	Remove
ArrayList	110	490	3790	8730
LinkedList	1980	220	110	110

Може да се види, че произволния достъп (**get()**) е евтин за **ArrayList** и скъп за **LinkedList**. (Наопаки, итерацията е по-бърза за **LinkedList** отколкото за **ArrayList**, което противоречи на интуицията.) От друга страна, вмъкванията и изтриванията в средата на списъка са драматично по-евтини за **LinkedList** отколкото за **ArrayList**. Вероятно най-добрият подход е да се използва **ArrayList** първоначално и да се мие на **LinkedList** ако се натъкнете на проблеми със скоростта дължащи се на множество вмъквания и изтривания.

Избор между **Set**овете

Имате избор между **TreeSet** и **HashSet**, в зависимост от дължината на **Set**(ако ви трябва наредена последователност от **Set**, използвайте **TreeSet**). Следната тестова програма разглежда тези неща:

```
//: c08:newcollections:SetPerformance.java
package c08.newcollections;
import java.util.*;

public class SetPerformance {
    private static final int REPS = 200;
    private abstract static class Tester {
        String name;
        Tester(String name) { this.name = name; }
        abstract void test(Set s, int size);
    }
    private static Tester[] tests = {
        new Tester("add") {
            void test(Set s, int size) {
                for(int i = 0; i < REPS; i++) {
                    s.clear();
                    Collection1.fill(s, size);
                }
            }
        },
        new Tester("contains") {
            void test(Set s, int size) {
                for(int i = 0; i < REPS; i++)
                    for(int j = 0; j < size; j++)
                        s.contains(Integer.toString(j));
            }
        },
        new Tester("iteration") {
            void test(Set s, int size) {
                for(int i = 0; i < REPS * 10; i++) {
                    Iterator it = s.iterator();
                    while(it.hasNext())
                        it.next();
                }
            }
        },
    };
    public static void test(Set s, int size) {
        // A trick to print out the class name:
        System.out.println("Testing " +
            s.getClass().getName() + " size " + size);
        Collection1.fill(s, size);
        for(int i = 0; i < tests.length; i++) {
            System.out.print(tests(i).name);
            long t1 = System.currentTimeMillis();
            tests(i).test(s, size);
            long t2 = System.currentTimeMillis();
            System.out.println(": " +
                ((double)(t2 - t1)/(double)size));
        }
    }
}
```

```

public static void main(String[] args) {
    // Small:
    test(new TreeSet(), 10);
    test(new HashSet(), 10);
    // Medium:
    test(new TreeSet(), 100);
    test(new HashSet(), 100);
    // Large:
    test(new HashSet(), 1000);
    test(new TreeSet(), 1000);
}
} ///:~

```

Следната таблица показва резултатите от едно пускане (с Beta3 софтуер на една конкретна платформа; ще го пуснете и при себе си също):

Type	Test size	Add	Contains	Iteration
TreeSet	10	22.0	11.0	16.0
	100	22.5	13.2	12.1
	1000	31.1	18.7	11.8
HashSet	10	5.0	6.0	27.0
	100	6.6	6.6	10.9
	1000	7.4	6.6	9.5

HashSet изобщо превъзхожда **TreeSet** за всички операции, а скоростта е ефективно независима от дължината.

Избор между Марове

Когато избираме измежду реализации на **Map** дължината на **Map** е нещото, което най-много засяга скоростта, а следната програма разработва това:

```

//: c08:newcollections:MapPerformance.java
// Demonstrates performance differences in Maps
package c08.newcollections;
import java.util.*;

public class MapPerformance {
    private static final int REPS = 200;
    public static Map fill(Map m, int size) {
        for(int i = 0; i < size; i++) {
            String x = Integer.toString(i);
            m.put(x, x);
        }
        return m;
    }
    private abstract static class Tester {
        String name;
        Tester(String name) { this.name = name; }
        abstract void test(Map m, int size);
    }
    private static Tester() tests = {
        new Tester("put") {
            void test(Map m, int size) {
                for(int i = 0; i < REPS; i++) {
                    m.clear();
                    fill(m, size);
                }
            }
        }
    }
}

```

```

        }
    },
},
new Tester("get") {
    void test(Map m, int size) {
        for(int i = 0; i < REPS; i++)
            for(int j = 0; j < size; j++)
                m.get(Integer.toString(j));
    }
},
new Tester("iteration") {
    void test(Map m, int size) {
        for(int i = 0; i < REPS * 10; i++) {
            Iterator it = m.entrySet().iterator();
            while(it.hasNext())
                it.next();
        }
    }
},
};

public static void test(Map m, int size) {
    // A trick to print out the class name:
    System.out.println("Testing " +
        m.getClass().getName() + " size " + size);
    fill(m, size);
    for(int i = 0; i < tests.length; i++) {
        System.out.print(tests(i).name);
        long t1 = System.currentTimeMillis();
        tests(i).test(m, size);
        long t2 = System.currentTimeMillis();
        System.out.println(": " +
            ((double)(t2 - t1)/(double)size));
    }
}
}

public static void main(String[] args) {
    // Small:
    test(new Hashtable(), 10);
    test(new HashMap(), 10);
    test(new TreeMap(), 10);
    // Medium:
    test(new Hashtable(), 100);
    test(new HashMap(), 100);
    test(new TreeMap(), 100);
    // Large:
    test(new HashMap(), 1000);
    test(new Hashtable(), 1000);
    test(new TreeMap(), 1000);
}
}
} ///:~

```

Понеже дължината е гвоздеят, ще видите че тестовете делят времето на изпълнение на дължината за нормализация на резултата. Ето едни резултати. (Вашите вероятно ще са различни.)

Type	Test size	Put	Get	Iteration
Hashtable	10	11.0	5.0	44.0
	100	7.7	7.7	16.5

Type	Test size	Put	Get	Iteration
TreeMap	1000	8.0	8.0	14.4
	10	16.0	11.0	22.0
	100	25.8	15.4	13.2
	1000	33.8	20.9	13.6
HashMap	10	11.0	6.0	33.0
	100	8.2	7.7	13.7
	1000	8.0	7.8	11.9

Както може да се очаква, скоростта на **Hashtable** е приблизително равна на тази на **HashMap** (може също да забележите че **HashMap** е изобщо малко по-бърза. Помните че **HashMap** има да замества **Hashtable**). **TreeMap** е изобщо по-бавен от **HashMap**, та щащо въобще би се използвал? Не бихте могли да я използвате като **Map**, а като начин за създаване на подреден списък. Поведението на дървото е такова, че то винаги е подредено и няма нужда специално да се сортира. (*Начинът* на сортиране ще се обсъди по-късно.) Веднъж като запълните **TreeMap**, може да викате **keySet()** за да вземете **Set** изглед на ключовете, **toArray()** за да създадете масив от ключовете. Може да използвате **static** метода **Arrays.binarySearch()** (обсъждан по-късно) за скоростно намиране на обекти в сортирания ви масив. Разбира се това ще се прави само ако по някаква причина поведението на **HashMap** е неприемливо, понеже **HashMap** е проектиран бързо да намира нещата. Накрая, като използвате **Map** вашият първи избор ще бъде **HashMap**, а само ако искате непрекъснато сортиран **Map** ще ви трябва **TreeMap**.

Има друг въпрос свързан със скоростта, който горната таблица не адресира, а именно скоростта на създаване. Следващата програма изпроверва скоростта на създаване на **Map**:

```
//: c08:newcollections:MapCreation.java
// Demonstrates time differences in Map creation
package c08.newcollections;
import java.util.*;

public class MapCreation {
    public static void main(String[] args) {
        final long REPS = 100000;
        long t1 = System.currentTimeMillis();
        System.out.print("Hashtable");
        for(long i = 0; i < REPS; i++)
            new Hashtable();
        long t2 = System.currentTimeMillis();
        System.out.println(": " + (t2 - t1));
        t1 = System.currentTimeMillis();
        System.out.print("TreeMap");
        for(long i = 0; i < REPS; i++)
            new TreeMap();
        t2 = System.currentTimeMillis();
        System.out.println(": " + (t2 - t1));
        t1 = System.currentTimeMillis();
        System.out.print("HashMap");
        for(long i = 0; i < REPS; i++)
            new HashMap();
        t2 = System.currentTimeMillis();
        System.out.println(": " + (t2 - t1));
    }
} ///:~
```

По времето когато тази програма беше писана **TreeMap** беше драматично по-бърз от другите два типа. Това, заедно с приемливата и смислена скорост на **put()** от **TreeMap**, дава

възможна стратегия в случай че създавате множество **Марове**, а чак по-късно в програмата си правите много търсения: Да се създадат и запълнят **TreeMaps**, а после когато започнете търсенията, да се превърнат важните **TreeMapове** в **HashMapове** чрез **HashMap(Map)** конструктора. Так да подчертая, това ще се прави само ако има доказано тясно място и значително намаление на скоростта. (“Първо го направи да върви, после го прави да бъде бързо – ако трябва.”)

Неподдържани операции

Възможно е да се превърне масив в **List** със статичния метод **static Arrays.asList()**:

```
//: c08:newcollections:Unsupported.java
// Sometimes methods defined in the Collection
// interfaces don't work!
package c08.newcollections;
import java.util.*;

public class Unsupported {
    private static String[] s = {
        "one", "two", "three", "four", "five",
        "six", "seven", "eight", "nine", "ten",
    };
    static List a = Arrays.asList(s);
    static List a2 = Arrays.asList(
        new String[] { s(3), s(4), s(5) });
    public static void main(String[] args) {
        Collection1.print(a); // Iteration
        System.out.println(
            "a.contains(" + s(0) + ") = " +
            a.contains(s(0)));
        System.out.println(
            "a.containsAll(a2) = " +
            a.containsAll(a2));
        System.out.println("a.isEmpty() = " +
            a.isEmpty());
        System.out.println(
            "a.indexOf(" + s(5) + ") = " +
            a.indexOf(s(5)));
        // Traverse backwards:
        ListIterator lit = a.listIterator(a.size());
        while(lit.hasPrevious())
            System.out.print(lit.previous());
        System.out.println();
        // Set the elements to different values:
        for(int i = 0; i < a.size(); i++)
            a.set(i, "47");
        Collection1.print(a);
        // Compiles, but won't run:
        lit.add("X"); // Unsupported operation
        a.clear(); // Unsupported
        a.add("eleven"); // Unsupported
        a.addAll(a2); // Unsupported
        a.removeAll(a2); // Unsupported
        a.remove(s(0)); // Unsupported
        a.removeAll(a2); // Unsupported
    }
}
```

| } //:/~

Ще откриете че само част от интерфейсите на **Collection** и **List** са в действителност реализирани. Останалите методи довеждат до неприятната поява на неща, наречено **UnsupportedOperationException**. Ще научите всичко за изключениета в следващата глава, но накъсо историята е, че **Collection interface**, както и някои други **interface**и в библиотеката Колекции на Java 2, съдържа методи "по желание", които могат да бъдат или да не бъдат "поддържани" в конкретен клас който **implements** този **interface**. Извикването на неподдържан метод предизвиква **UnsupportedOperationException** за да се индицира програмна грешка.

"Какво?!?" казвате вие скептично. "Цялата работа на **interface**ите и базовите класове е че те обещават техните методи да направят нещо полезно! Това наруши обещанието – значи не само че ще извикаме методи които ги няма, но ще спре и програмата! Безопасността на типовете просто е изхвърлена през прозореца!" Това не е чак така лошо. С **Collection**, **List**, **Set** и **Map** компилаторът продължава да ви ограничава да викате методи само в рамките на този **interface**, така че не е като в Smalltalk (където може да се извика всеки метод на всеки обект, но да се разбере какво прави може чак по време на изпълнение). В добавка, повечето методи кайто взимат **Collection** като аргумент само четат тази **Collection** – всичките "четящи" методи в **Collection** не са по желание.

Този подход предотвратява експлозията на интерфейсите в дизайна. Други разработки на библиотеки от колекции винаги свършват в блато от интерфейси които да описват всичките вариации по темата и са трудни за изучаване. Даже е невъзможно да се проследят различните специални класове и **interface**и, понеже някой винаги би могър да наследи някой **interface**. С одхода на "неподдържаната операция" се достига важна цел на Java 2 библиотеката колекции: тя е лесна за учене и използване. За да работи подходът, обаче:

1. **UnsupportedOperationException** трябва да е рядък случай. Тоез за повечето случаи класовете трябва да работят, а само в специални случаи да има неподдържана операция. Това е вярно за библиотеката колекции на Java 2 тъй като класовете които използвате 99% от времето – **ArrayList**, **LinkedList**, **HashSet** и **HashMap**, както и другите конкретни реализации – поддържат всичките операции. Дизайнът дава "задна вратичка" ако искате да създадете нова **Collection** без да се предефинира всичко за **Collection interface**, и все пак да пасва на съществуващата библиотека.
2. Когато операцията е неподдържана, има достатъчна вероятност че **UnsupportedOperationException** ще се появи по време на разработката, а не когато доставите програмата на потребителя. Най-после, това индицира програмна грешка: използвали сте клас некоректно. Това е по-малко сигурно и излиза когато експерименталната натура на този проект излиза на преден план. Само с течение на времето ще видим дали работи добре.

В горния пример **Arrays.asList()** дава **List** който е подкрепен от масив с фиксирана дължина. Това прави смислено само поддържаните операции да са тези, които не променят дължината на масива. Ако, от друга страна, нов **interface** беше необходим за да се изрази тази нова черта на поведението (наречен, да кажем, "**FixedSizeList**"), щеше да се отвори вратата за усложняване и скоро нямаше да знаете от къде да започнете, ако искате да използвате библиотеката.

Документацията за метод който взима **Collection**, **List**, **Set** или **Map** трябва да показва какви методи по желания трябва да се реализират. Например сортирането изиска **set()** и **Iterator.set()** методите но не и **add()** и **remove()**.

Сортиране и търсене

Java 2 дава помощ при търсене и сортиране в **List**ове. Това са **static** методи от два нови класа: **Arrays** за сортиране и търсене с масиви и **Collections** за сортиране и търсене в **List**ове.

Arrays

Класът **Arrays** има претоварен **sort()** и **binarySearch()** за масиви от всичките примитивни типове, както и за **String** и **Object**. Ето пример който показва търсене и сортиране на масив от **byte** (всички останали примитиви изглеждат по същия начин) и масив от **String**:

```
//: c08:newcollections:Array1.java
// Testing the sorting & searching in Arrays
package c08.newcollections;
import java.util.*;

public class Array1 {
    static Random r = new Random();
    static String ssource =
        "ABCDEFGHIJKLMNPQRSTUVWXYZ" +
        "abcdefghijklmnopqrstuvwxyz";
    static char[] src = ssource.toCharArray();
    // Create a random String
    public static String randString(int length) {
        char[] buf = new char[length];
        int rnd;
        for(int i = 0; i < length; i++) {
            rnd = Math.abs(r.nextInt()) % src.length;
            buf(i) = src(rnd);
        }
        return new String(buf);
    }
    // Create a random array of Strings:
    public static
    String[] randStrings(int length, int size) {
        String[] s = new String(size);
        for(int i = 0; i < size; i++)
            s(i) = randString(length);
        return s;
    }
    public static void print(byte[] b) {
        for(int i = 0; i < b.length; i++)
            System.out.print(b(i) + " ");
        System.out.println();
    }
    public static void print(String[] s) {
        for(int i = 0; i < s.length; i++)
            System.out.print(s(i) + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        byte[] b = new byte(15);
        r.nextBytes(b); // Fill with random bytes
        print(b);
        Arrays.sort(b);
        print(b);
        int loc = Arrays.binarySearch(b, b(10));
        System.out.println("Location of " + b(10) +
            " = " + loc);
        // Test String sort & search:
        String[] s = randStrings(4, 10);
        print(s);
```

```

        Arrays.sort(s);
        print(s);
        loc = Arrays.binarySearch(s, s(4));
        System.out.println("Location of " + s(4) +
            " = " + loc);
    }
} //:~

```

Първата част от класа съдържа ютилита за създаване на случаини **String** обекти използвайки масив от букви, от които случаино може да се избира. **randString()** връща стринг с произволна дължина, а **randStrings()** създава масив от случаини **String**ове, ако му се даде дължината на **String**a и желаната дължина на масива. Двета **print()** метода опростяват извеждането на примерните масиви. В **main()** **Random.nextBytes()** пълни масива-аргумент със случаино избрани **byte**ове. (Няма съответни **Random** методи за създаване на масиви от другите примитивни типове.) Като получите масива, ще видите че има единствено извикване на метода **sort()** или **binarySearch()**. Има важно предупреждение засягащо **binarySearch()**: Ако не викате **sort()** преди изпълнение на **binarySearch()**, може да се получи непредсказуемо поведение, включително безкрайни цикли.

Сортиране и търсене при **String**овете става по същия начин, но когато пуснете програмата ще забележите нещо интересно: сортирането е лексикографично, така че големите букви предхождат малките в знаковия набор. Така всички главни букви са в началото на списъка, следвани от малките, така че 'Z' предхожда 'a'. Види се даже телефонните указатели са сортирани по този начин.

Comparable и Comparator

А ако това не е желаното? Например индексът в тази книга не би бил твърде полезен ако трябваше да гледате на две различни места за всичко което започва с 'A' и 'a' респективно.

Когато искате да сортирате масив от **Object** има проблем. Какво определя наредбата на два **Object**a? За нещастие първите проектанти на Java не смятаха това за важен проблем, или той се е появил в кореновия клас **Object**. Като резултат наредбата трябва да се внесе отвън за **Object**и, а Java 2 библиотеката Колекции дава стандартен начин да се прави това (което е почти толкова добро като да се прави в кореновия **Object**).

Има **sort()** за масиви от **Object** (и **String**, разбира се, е **Object**) който приема втори аргумент: обект който прилага **Comparator** интерфейс (част от библиотеката Колекции на Java 2) и изпълнява сравнения с неговия единствен **compare()** метод. Този метод взема като аргументи двата обекта които ще се сравняват и връща отрицателно цяло ако първият е по-малък, нула ако са равни и положително цяло ако първият е по-голям. С това знание **String** частта от горния пример може да бъде преписана за да дава азбучно сортиране:

```

//: c08:newcollections:AlphaComp.java
// Using Comparator to perform an alphabetic sort
package c08.newcollections;
import java.util.*;

public class AlphaComp implements Comparator {
    public int compare(Object o1, Object o2) {
        // Assume it's used only for Strings...
        String s1 = ((String)o1).toLowerCase();
        String s2 = ((String)o2).toLowerCase();
        return s1.compareTo(s2);
    }
    public static void main(String[] args) {
        String[] s = Array1.randStrings(4, 10);
    }
}

```

```

Array1.print(s);
AlphaComp ac = new AlphaComp();
Arrays.sort(s, ac);
Array1.print(s);
// Must use the Comparator to search, also:
int loc = Arrays.binarySearch(s, s(3), ac);
System.out.println("Location of " + s(3) +
" = " + loc);
}
} //:~

```

Чрез превръщане към **String** методът **compare()** неявно проверява да е използван само със **String** обекти – системата ще хване всякакви несъответствия по времето на изпълнение. След превръщане на двета **Stringa** към малки букви, методът **String.compareTo()** дава желаните резултати.

Когато използвате ваш собствен **Comparator** за да изпълните **sort()**, трябва да използвате същия **Comparator** когато използвате **binarySearch()**.

Класът **Arrays** има друг **sort()** който взема единствен аргумент: масив от **Object**, но без **Comparator**. Този **sort()** също трябва да има начин да сравни два **Objecta**. Той използва естествения начин на сравнение който е вграден в класа чрез **Comparable interface**. Този **interface** има единствен метод, **compareTo()**, който сравнява обекта с аргумента си и връща отрицателно цяло, нула или положително цяло ако обектът е по-малък, равен или по-голям от аргумента. Прост пример демонстрира това:

```

//: c08:newcollections:CompClass.java
// A class that implements Comparable
package c08.newcollections;
import java.util.*;

public class CompClass implements Comparable {
    private int i;
    public CompClass(int ii) { i = ii; }
    public int compareTo(Object o) {
        // Implicitly tests for correct type:
        int argi = ((CompClass)o).i;
        if(i == argi) return 0;
        if(i < argi) return -1;
        return 1;
    }
    public static void print(Object[] a) {
        for(int i = 0; i < a.length; i++)
            System.out.print(a[i] + " ");
        System.out.println();
    }
    public String toString() { return i + ""; }
    public static void main(String[] args) {
        CompClass[] a = new CompClass[20];
        for(int i = 0; i < a.length; i++)
            a[i] = new CompClass(
                (int)(Math.random() * 100));
        print(a);
        Arrays.sort(a);
        print(a);
        int loc = Arrays.binarySearch(a, a(3));
        System.out.println("Location of " + a(3) +

```

```
    " = " + loc);
}
} //://~
```

Разбира се, вашият **compareTo()** може да бъде толкова сложен, колкото е необходимо.

Listове

List може да бъде сортиран и претърсван по същия начин както масив. **static** методите за сортировка и претърсване на **List** се съдържат в класа **Collections**, но имат подобни сигнатури на тези в **Arrays**: **sort(List)** за сортиране на **List** от обекти който прилага **Comparable**, **binarySearch(List, Object)** за намиране на обект в списъка, **sort(List, Comparator)** за сортиране на **List** чрез **Comparator**, **binarySearch(List, Object, Comparator)** за намиране на обект в него лист.⁴ Този пример използва дефинирания преди **CompClass** и **AlphaComp** за демонстриране на инструменти за сортиране в **Collections**:

```
//: c08:newcollections>ListSort.java
// Sorting and searching Lists with 'Collections'
package c08.newcollections;
import java.util.*;  
  
public class ListSort {
    public static void main(String[] args) {
        final int SZ = 20;
        // Using "natural comparison method":
        List a = new ArrayList();
        for(int i = 0; i < SZ; i++)
            a.add(new CompClass(
                (int)(Math.random() * 100)));
        Collection1.print(a);
        Collections.sort(a);
        Collection1.print(a);
        Object find = a.get(SZ/2);
        int loc = Collections.binarySearch(a, find);
        System.out.println("Location of " + find +
            " = " + loc);
        // Using a Comparator:
        List b = new ArrayList();
        for(int i = 0; i < SZ; i++)
            b.add(Array1.randString(4));
        Collection1.print(b);
        AlphaComp ac = new AlphaComp();
        Collections.sort(b, ac);
        Collection1.print(b);
        find = b.get(SZ/2);
        // Must use the Comparator to search, also:
        loc = Collections.binarySearch(b, find, ac);
        System.out.println("Location of " + find +
            " = " + loc);
    }
} //://~
```

Използването на тези методи е идентично на това от **Arrays**, използва се **List** вместо масив.

TreeMap трябва също да подреди обектите си според **Comparable** или **Comparator**.

⁴ At the time of this writing, **Collections.sort()** has been modified to use a *stable sort algorithm* (one that does not reorder equal elements).

ЮТИЛИТА

Има определен брой полезни ютилита в класа **Collections**:

enumeration(Collection)	Дава Enumeration по стария стил за аргумента.
max(Collection) min(Collection)	Дава максималния или минималния елемент на аргумента използвайки натураното за Collection сравняване.
max(Collection, Comparator) min(Collection, Comparator)	Дава максималния или минималния елемент за Collection чрез Comparator .
nCopies(int n, Object o)	Връща неизменяем лист List с дължина n чиито манипулатори сочат o .
subList(List, int min, int max)	Връща нов List подкрепян от посочения за аргумент List който е прозорец в който този аргумент ще индексира започвайки от min и завършвайки точно преди max .

Забележете че **min()** и **max()** работят с обекти **Collection**, не с **List**ове, така че няма нужда да се тревожите дали **Collection** е сортиран или не. (Както се спомена по-рано, трябва да сортирате **sort()** **List** или масив преди изпълнение на **binarySearch()**.)

Правене на **Collection** или **Map** неизменяеми

Често е удобно да се създава версия само за четене на **Collection** или **Map**. Класът **Collections** позволява да се направи това чрез подаване на оригиналната колекция на метод, който връща версия само за четене. Има четири вариации на този метод, по езва за **Collection** (ако не искате да третирате **Collection** като по-специфичен тип), **List**, **Set** и **Map**. Този пример показва правилният начин за построяване на версия само за четене от всеки:

```
//: c08:newcollections:ReadOnly.java
// Using the Collections.unmodifiable methods
package c08.newcollections;
import java.util.*;

public class ReadOnly {
    public static void main(String[] args) {
        Collection c = new ArrayList();
        Collection1.fill(c); // Insert useful data
        c = Collections.unmodifiableCollection(c);
        Collection1.print(c); // Reading is OK
        //! c.add("one"); // Can't change it

        List a = new ArrayList();
        Collection1.fill(a);
        a = Collections.unmodifiableList(a);
        ListIterator lit = a.listIterator();
        System.out.println(lit.next()); // Reading OK
        //! lit.add("one"); // Can't change it
    }
}
```

```

Set s = new HashSet();
Collection1.fill(s);
s = Collections.unmodifiableSet(s);
Collection1.print(s); // Reading OK
//! s.add("one"); // Can't change it

Map m = new HashMap();
Map1.fill(m, Map1.testData1);
m = Collections.unmodifiableMap(m);
Map1.print(m); // Reading OK
//! m.put("Ralph", "Howdy!");
}
} //://~
```

Във всеки отделен случай трябва да попълните контейнера със значещи данни преди да го направите само за четене. Като веднъж е натоварен, най-добрият подход е да заместите манипулатора с този, получен от "непроменимото" извикване. По този начин се избягва рисъкът да се промени нежелано съдържанието след като е направено непроменимо. От друга страна, този инструмент също позволява да задържите променимия контейнер като **private** в клас и да върщате манипулатор "само за четене" чрез извикване на метод. Така че може да правите промени от извънре на класа но всеки друг може само да чете.

Извикване на "непроменимия" метод за конкретен тип не предизвиква проверка по време на компилация, но веднъж като е станала трансформацията, каквото и извикване да е на метод който променя съдържанието на конкретен контейнер предизвиква **UnsupportedOperationException**.

Синхронизиране на **Collection** или **Map**

Ключовата дума **synchronized** е малък момент от предмета на *multithreading* (многонишковост-б.пр.), по-сложна тема която няма да бъде въведена до глава 14. Тук ще отбележа само че класът **Collections** съдържа начин автоматично да се синхронизира целия контейнер. Синтаксисът е подобен на "непроменимите" методи:

```

//: c08:newcollections:Synchronization.java
// Using the Collections.synchronized methods
package c08.newcollections;
import java.util.*;

public class Synchronization {
    public static void main(String[] args) {
        Collection c =
            Collections.synchronizedCollection(
                new ArrayList());
        List list = Collections.synchronizedList(
            new ArrayList());
        Set s = Collections.synchronizedSet(
            new HashSet());
        Map m = Collections.synchronizedMap(
            new HashMap());
    }
} //://~
```

В този случай непосредствено се подава нов контейнер чрез подходящ "синхронизиран" метод; така няма начин непреднамерено да се покаже на света несинхронизирана версия.

Колекциите на Java 2 имат също механизъм да предотвратят повече от един процес едновременно да променя съдържанието на контейнер. Проблемът се появава когато

правите итерации в даден контейнер и друг процес вмъква, маха или променя обект в същия контейнер. Може би вече се преминали този обект, може той да не е достигнат още, може дължината на контейнера да се увеличи след извикването от вас на `size()` – има много сценарии за катастрофа. Колекциите на Java 2 библиотеките включват *fail fast* механизъм който внимава за промени, направени не от вашия процес който отговаря за тях. Ако се установи промяна направена от нещо друго еднага се изхвърля **ConcurrentModificationException**. Това е "fail-fast" аспектът – той не се опитва да установи какъв е проблемът после чрез някакъв усложнен алгоритъм.

Резюме

Да прегледаме колекциите доставяни със стандартната Java (1.0 и 1.1) библиотека (**BitSet** не е включена тук понеже е клас за по-специална употреба):

1. Масивът асоциира числени индекси с обектите. Той съдържа обекти от известен тип, така че не е необходимо да се прави кастиране. Може да бъде многоразмерен и може да съдържа примитиви. Обаче дължината му не може да бъде променяна след като е създаден.
2. **Vector** също асоциира числени индекси с обекти – мае да мислите за масивите и **Vectors** като колекции с произволен достъп. **Vector** автоматически си променя дължината с въвеждането на нови елементи. Но **ArrayList** може да съдържа само манипулатори на **Object**, така че не може да съдържа примитиви и трябва да се прави кастиране когато се извлича **Object** манипулатор от колекцията.
3. **Hashtable** е тип **Dictionary**, което е начин да се асоциират не числа, но обекти с други обекти. **Hashtable** също поддържа произволен достъп до обектите, фактически цялата работа в тях е фокусирана върху бързия достъп.
4. **Stack** е "проследен влязъл-първи излиза" (LIFO) опашка.

Ако сте запознати със структурите данни, може да се чудите защо няма по-богато множество колекции. От функционална гледна точка нуждаете ли се от повече колекции? С **Hashtable** може да вмъквате неща и да ги намирате бързо, а с **Enumeration** може да се движите през колекцията и да извършвате операция над всеки елемент от последователността. Това е мощен инструмент и може би той би трябвало да е достатъчен.

Но **Hashtable** няма концепция за подреждане. **Vectors** дават линейно наредждане, но е скъпо да се вмъкне елемент в средата на всеки от тях. В добавка опашките, дековете, опашките с приоритет и дърветата са за подреждане на елементите, не само за слагане и последващо намиране на елементи линейно. Тези структури данни са също полезни, затова са включени в Standard C++. Поради тази причина ще считате колекциите в стандартната Java библиотека само като начална точка и, ако трябва да използвате Java 1.0 или 1.1, използвайте JGL когато се нуждаете от нещо извън това.

Ако можете да използвате Java 2 ще използвате само Колекциите на Java 2, които вероятно ще удовлетворят всичките ви нужди. Забележете че повечето от тази книга беше създадена чрез Java 1.1, така че ще видите, че колекциите които са използвани до края на книгата са тези от Java 1.1: **Vector** и **Hashtable**. Това е малко болезнено ограничение на моменти, но дава по-добра обратна съвместимост със стар Java код. Ако пишете нов код с Java 2, Колекциите на Java 2 ще ви служат много по-добре.

Упражнения

1. Създайте нов клас наречен **Gerbil** с **int gerbilNumber** който се инициализира в конструктора (подобно на **Mouse** примера в тази глава). Дайте му метод наречен **hop()** който извежда кое gerbil число е и че то подскача. Създайте **ArrayList** и добавете много **Gerbil** обекти във **Vector**. Сега използвайте метода **elementAt()** за придвижване през **Vector** и извикайте **hop()** за всеки **Gerbil**.
2. Променете упражнение 1 така че да използва **Iterator** за придвижване през **Vector** докато се вика **hop()**.
3. В **AssocArray.java** сменете примера така че да използва **Hashtable** вместо **AssocArray**.
4. Вземете класа **Gerbil** от упражнение 1 и го сложете в **Hashtable**, асоциирайки името на **Gerbil** като **String** (ключ) за всеки **Gerbil** (стойността) които слагате в таблицата. Вземете **Iterator** за **keys()** и го използвайте за придвижване през **Hashtable**, оглеждайки **Gerbil** за всеки ключ и печатайки ключа, казвайки на **gerbil** да направи **hop()**.
5. Променете упражнение 1 в глава 7 да използва **ArrayList** за съдържане на **Rodent** и **Iterator** за придвижване през последователността от **Rodent**. Помнете че **ArrayList** съдържа само **Objects** така че трябва да се прави каст (т.е.: RTTI) когато се прави достъп до отделни **Rodent**.
6. (Преходно) В глава 7 намелете примера **GreenhouseControls.java**, който се състои от три файла. В **Controller.java** класът **EventSet** е точно колекция. Променете кода да използва **Stack** вместо **EventSet**. Това ще изисква повече от само заместването на **EventSet** със **Stack**; ще трябва да използвате също и **Iterator** за циклене в множеството елементи. Вероятно ще намерите че е по-лесно да третирате на моменти колекцията като **Stack** и в други моменти като **Vector**.
7. (Предизвикателство). Намерете сорса на **Vector** в библиотеката сорс на Java която идва с всички Java дистрибуции. Копирайте този код и направете специална версия наречена **intVector** която съдържа само **int**. Проучете какво ще трябва за да се направи специална версия на **Vector** за всички примитивни типове. Сега проучете какво ще стане ако създадете клас-свързан списък който работи с всички примитивни типове. Ако някога се реализират параметризираните типове в Java, те ще дадат начин да се свърши тази работа автоматично за вас (както и много други ползи).

9: Обработка на грешки чрез изключения

Основната философия на Java е че “лошо формиран код няма да бъде задействан.”

Както и със C++, идеалния момент да се хване грешка е времето на компилиация, преди даже да се опитате да пуснете програмата. Обаче не всички грешки могат да бъдат открити по време на компилиация. Определена част проблеми могат да се обработват само по време на изпълнение, по някакъв формализъм, където източникът на грешката трябва да има начин да подаде информация на кода, който ще я обработва, за да може обработката да бъде подходяща.

В С и други ранни езици можеше да съществуват няколко такива формализма, а освен това те бяха задавани като конвенция, а не като част от програмния език. Типично се връщаща стойност и се слагаше флаг, като се предполагаше, че получателят ще провери флага и ще изследва стойността за да разбере проблема. Обаче с течение на годините стана ясно, че програмистите които правят библиотеките имат тенденция да се считат за непобедими, както в: “Да, грешки могат да се появят при другите но не и в моя код.” Така, не много изненадващо, те не правеха нужните проверки (а и понякога ситуацията на грешка бяха твърде тъпи за да се проверява за тях¹). Ако бяхте достатъчно последователни да проверявате за грешки всеки път като извикате метод, вашият код рискуваше да се превърне в нечитаем кошмар. Понеже програмистите трябваше да се се оправят с тези програмни системи те упорито поддържаха че: Този подход към обработката на грешки е главното ограничение пред правенето на добри, големи, управляеми програми.

Решението е да се разкара решаването на въпросите според случая от обработката на грешки и да се въведе формализъм. Това фактически има дълга история, понеже реализации на поддръжка на изключения се отнасят и за операционни системи от 1960-те години и даже за **on error goto** на BASIC. Но поддръжката на изключения в C++ беше базирана на тази в Ada, а в Java е основана предимно на C++ (макар и повече да изглежда на Object Pascal).

Думата “exception” се има пред вид в значението “Вземам изключение за нещо.” В точката на възникване на проблема би могло да не се знае какво да се прави с него, но се знае, че не може просто весело да се продължи; трябва да се спре и някой, някъде, трябва да изясни какво ще се прави. Но вие нямаете информация в текущия момент за решаване на проблема. Така че представяте проблема в по-висок контекст така че някой с нужната информация ще вземе мерките (доста подобно на веригата от команди (при командването на нещо, например кораб-бел.пр.)).

Другата голяма полза от поддръжкането на изключенията е че те очистват кода. Наместо да се проверява за конкретна грешка и това да се прави в няколко места в програмата, повече няма да се проверява в точката на извикване на метода (понеже изключението гарантира, че

¹ The C programmer can look up the return value of **printf()** for an example of this.

някой ще го хване). Необходимо е да се обработи проблема само в едно място, тъй наречения *exception handler*. Това спестява писане и разделя кода за нещата които трябва да се правят от този който е за в случай че се развали нещо. Изобщо да се пише, чете и тества код става много по-ясно като се използват изключения, отколкото по стария начин.

Понеже поддръжката на изключения се налага от компилатора на Java, има толкова много примери в тази книга които могат да се напишат без знание на подробности. Тази глава ви въвежда в нещата, които позволяват да напишете код за правилна обработка на изключенията, а също и как да генерирате свои собствени изключения, ако ваш метод има неприятности.

Основни изключения

Изключително състояние е проблем, който не дава да продължи изпълнението на текущия метод или обхват. Важно е да се различи изключителното състояние от обикновения проблем, в който има достатъчно информация в текущия контекст, даваща възможност нещо да се направи по въпроса. В изключителното състояние не може нищо да се направи поради липсата на информация в текущия контекст. Всичко което може да се направи е да се изскочи от текущия контекст и да се разгледа проблема в по-висок контекст. Това е, което се случва като се изхвърли изключение.

Прост пример е делението. Ако може да се опитате да делите на нула, може и да си струва да проверявате и да не продължите с делението. Но какво следва ако делителят е нула? Може да знаете, в контекста на конкретния метод, какво да се прави в такъв случай. Но може и да не знаете, ако такава стойност не се очаква и тогава трябва да изхвърлите изключение, а не да правите проверки.

Като изхвърлите изключение се случват няколко неща. Първо се създава обект на изключението по начин по който се създава всеки обект в Java: на хийпа, с **new**. После текущия път на изпълнение (този, който не може да продължи, запомнете) се спира и манипулятор към обекта на изключението се изтласква от текущия контекст. В този момент механизма за обслужване на изключенията влиза в действие и започва да търси подходящо място където програмата да продължи. Това подходящо място е *exception handler*-а, чиято задача е да се справи с проблема така, че програмата да може да вземе друг курс или просто да продължи.

Като прост пример на изхвърляне на изключение да вземем обектов манипулятор наречен **t**. Възможно е да се даде манипулятор, който не е бил инициализиран, така че може да пожелаете да правите проверка преди подаването на въпросния манипулятор като аргумент. Може да пратите информация за грешката в по-широкия контекст чрез създаване на обект представящ нужната информация и "изхвърлянето му" извън текущия контекст. Това се казва *изхвърляне на изключение*. Ето го как изглежда:

```
if(t == null)  
    throw new NullPointerException();
```

Това изхвърля изключение, като ви позволява – в текущия контекст – да абдикирате от отговорността да мислите повече за проблема. Той се оправя никакси магически някъде другаде. Скоро ще се покаже точно къде.

Аргументи на изключението

Всеки обект в Java изключенията винаги се създават в хийпа с **new** и се вика конструктор. Има два конструктора за всички стандартни изключения; първият е по подразбиране, а вторият приема стринг за аргумент така че може да включите уместна информация в изключението:

```
if(t == null)
    throw new NullPointerException("t = null");
```

Този стринг после може да бъде извлечен чрез различни методи както ще се покаже по-късно.

Ключовата дума **throw** причинява случването на относително магически неща. Първо се изпълнява **new**-израз за да се създаде обект, който не е там при нормалното изпълнение на програмата и, разбира се, конструктор се вика за обекта. После обекта, фактически, се "връща" от метода, макар и типът му да не е като този, закойто е проектиран метода. Опростен начин да се мисли за механизма на изключениета е да се смятат за друг начин на връщане от метод, макар и да има трудности ако се прокара тази аналогия твърде далеч. Може също да се излезе от най-вътрешния обхват чрез изхвърляне на изключение. Връща се стойност, а методът или обхватът завършват изпълнението си.

Всяка прилика с обикновеното завършване на метод свършва тук, понеже мястото където се връща изпълнението на програмата е напълно различен от това при нормално завършване на метода. (Завършвате с подходящия обработчик на изключения който може да бъде на километри далеч – много нива позниско в стека на извикванията – от мястото където изключението е изхвърлено.)

В добавка може да изхвърляте който тип от **Throwable** обект си искате. Типично ще изхвърляте различен тип изключение за различни типове грешки. Идеята е да се съхранят информация в обекта на изключението и в типа на избрания обект, така че някой в по-широкия контекст да може да разбере какво да прави с вашето изключение. (Често единствената информация е типа на обекта на изключението, а нищо значително не се съхранява в обекта.)

Хващане на изключение

Ако метод изхвърли изключение той трябва да предполага, че то ще бъде хванато и обработено. Едно от предимствата на обработката на изключения в Java е, че позволява да се съсредоточите върху решаването на даден проблем на едно място, а после да се оправяте с грешките от този код на друго място.

За да видим как се хваща изключение трябва първо да усвоим концепцията за *guarded region*, което е секция от код, която може да изхвърля изключения и е следвана от код, който обработва тези изключения.

БЛОКЪТ **try**

Ако сте вътре в метод и изхвърлите изключение (или друг метод извикан извътре на този изхвърли изключение), методът ще завърши като част от процеса на изхвърлянето. Ако не искате **throw** да напусне метод, може да напишете специален блок в метода който да хване изключението. Това го казват *try block* понеже "опитвате" различните извиквания на методи вътре. Трай блокът е обикновен обхват предшестван от ключовата дума **try**:

```
try {
    // Code that might generate exceptions
}
```

Ако трябваше грижливо да пишете код за обработка на грешки на програмен език, който не поддържа изключения, щеше да е необходимо да окръжавате всяко извикване на метод с код за откриване и обработка на грешки, даже ако извиквате един и същ метод няколко пъти. С изключениета слагате всичко в един блок и обработвате всички грешки в него. Това значи че основната програма е много по-лесна за написване и четене, понеже кодът не се преплита с проверките за грешки.

Обработчици на изключения

Разбира се, изхвърленото изключение трябва да завърши някъде. Това “място” е *exception handler*-а, а има по един за всеки тип изключение което искате да хванете. Обработчиците на изключения непосредствено следват трай блока и са означени с ключовата дума **catch**:

```
try {
    // Code that might generate exceptions
} catch(Type1 id1) {
    // Handle exceptions of Type1
} catch(Type2 id2) {
    // Handle exceptions of Type2
} catch(Type3 id3) {
    // Handle exceptions of Type3
}

// etc...
```

Всяка клауза за хващане (*exception handler*) прилича на малък метод, който взема един и само един аргумент от определен тип. Идентификаторът (**id1**, **id2**, и т.н.) може да бъде използван вътре в хендлъра, точно като аргумент на метод. Понякога не употребявате идентификатора, понеже типът дава дотатъчно информация, но идентификаторът трябва да си бъде там.

Обработчиците трябва да се появят непосредствено до трай блока. Ако е изхвърлено изключениемеханизмът за изключенията тръгва на лов за първия хендлър който е за дадения тип. После влиза в клаузата за хващане, а изключението се счита обработено. (Търсено спира щом се влезе в **catch** клаузата.) Изпълнява се само съвпадащата клауза; не е както при **switch** оператора където трябва **break** след всеки **case** за да не се изпълнят и следващите.

Забележете че, с **try** блока, различни методи могат да генерират същото изключение, но е необходим само един обработчик.

Прекратяване vs. продължаване

Има два основни модела в теорията на обработката на изключения. В *прекратяването* (което се поддържа в Java и C++) се предполага, че грешката е толкова критична, че не може нищо да се направи на мястото на възникване. Който е изхвърлил изключението е преценил, че не може да се спаси положението и *не иска да се връща*.

Алтернативата се нарича *продължаване*. Това значи че обработчикът на изключения се оставя да свърши нещо за поправяне на ситуацията, а после методът в който е възникнало изключението се повтаря, с предполагаем успех този път. Ако предпочитате продължаване, значи се надявате да може да продължите след обработката на изключението. В този случай вашето изключение е по-подобно на извикване на метод – което и ще направите, за да имитирате тази идеология в Java в случаите когато искате такова поведение. (Тоест, не изхвърляйте изключение; извикайте метод, който да оправи нещата.) Алтернативно, сложете вашия **try** вътре в **while** цикъл който продължава да влиза пак в **try** блока докато резултатът стане удовлетворителен.

Исторически програмистите са използвали нещо подобно на продължаване от операционната система, което в края на краишата е завършвало с прекратяване чрез пропускане на продължаващия код. Така че и да изглежда по-привлекателно на пръв поглед, продължаването изглежда че не е чак толкова полезно на практика. Главната причина е вероятно *свързването* което се получава: вашият хендлър често трябва да знае каде е възникнало изключението и да съдържа не-родов код за конкретното място. Това прави кода

труден за писане и поддържане, особено за големи системи, където изключението може да възникне в различни точки.

Специфициране на изключението

В Java се изиска да информирате клиент-програмиста, който вика вашия метод, за изключението които биха могли да бъдат изхвърлени от този метод. Това е цивилизирано, понеже потребителят може да узнае какъв точно код да напише за прихващането на всичките възможни изключения. Разбира се, ако резполага със сурса, въпросният програмист може да го разгледа и да намери къде има **throw** оператори, но често библиотеките не изват със сурс. За да се избегне проблемът, Java дава синтаксис (и изиска да го използвате) за да може да кажете учило на клиента какви изключения изхвърля вашия метод, така че да може да бъдат обработени. Това е спецификация на изключението и е част от декларацията на метода, появяваща се след списъка аргументи.

Спецификацията на изключението използва допълнителна ключова дума, **throws**, следвана от типовете на потенциалните изключения. Тоест дефиницията на метода би могла да изглежда така:

```
| void f() throws tooBig, tooSmall, divZero { //...}
```

Ако напишем

```
| void f() { // ...}
```

това значи че не се изхвърлят изключения от метода. (Освен от тип **RuntimeException**, който може да бъде изхвърлен навсякъде – това ще се опише по-нататък.)

Не може да се лъже със спецификацията на изключението – ако вашият метод предизвиква изключения и не се справя с тях, компилаторът ще открие това и ще ви застави или да поддържате изключението, или да го опишете (споменете) в спецификацията. Чрез налагане на спецификация на изключението отгоре до долу Java че ще има коректност в изключениета *по време на изпълнение*.²

Има едно място където може да лъжете: може да твърдите че изхвърляте изключение, а да не го правите. Компилаторът си взема наум вашето и заставя потребителите на метода да поддържат въпросното изключение. Това има полезния ефек да означава изключението само, така че да може да започнете да изхвърляте изключението по-късно, без промяна на съществуващия код.

Хващане на кое да е изключение

Възможно е да се създаде обработчик, който хваща всякачъв тип изключение. Това се прави чрез хващане на базовия тип **Exception** (има и други видове базови изключения, но **Exception** е базата която е уместна практически за всички програмни активности):

```
catch(Exception e) {
    System.out.println("caught an exception");
}
```

Това ще хване всякакво изключение, така че ако ще го използвате ще го сложите *накрая* на вашия списък от обработчици, за да се избегне отнемането на изключението на обработчиците, които иначе биха се намирали след него.

² This is a significant improvement over C++ exception handling, which doesn't catch violations of exception specifications until run time, when it's not very useful.

Тъй като класът **Exception** е базов за всичките изключения които са важни за програмиста, не получавате много информация за изключението, но може да викате методите които идват от неговия базов тип **Throwable**:

String getMessage()

Взема детайлно съобщение.

String toString()

Връща кратко описание на Throwable, включително детайлно съобщение ако има такова.

void printStackTrace() **void printStackTrace(PrintStream)**

Извежда Throwable и трасирания стек на извикванията на Throwable. Стекът на извикванията показва веригата от викания на методи, която е довела до мястото, където е изхвърлено изключението.

Първата версия извежда на стандартния изход за грешки, втората на поток по ваш избор. Ако работите под Windows, не можете да пренасочите стандартната грешка и затова може да искате да използвате друг поток и накрая **System.out**; по този начин изходът може да бъде пренасочен по какъвто вие искате начин.

В добавка получавате някои други методи от базовия тип на **Throwable — Object** (базовия тип на всичко). Метод който може да е полезен при изключениета е **getClass()**, който връща обект, представящ класа на този обект. Може тогава да питате този **Class** обект за името му чрез **getName()** или **toString()**. Може да правите също по-сложни неща с **Class** обектите които неща не са необходими при обработката на изключения. **Class** обектите ще се изучават по-късно в тази книга.

Ето пример който показва използването на методите на **Exception**: (Виж страница 63 при проблеми с пускането на програмата.)

```
//: c09:ExceptionMethods.java
// Demonstrating the Exception Methods
package c09;

public class ExceptionMethods {
    public static void main(String[] args) {
        try {
            throw new Exception("Here's my Exception");
        } catch(Exception e) {
            System.out.println("Caught Exception");
            System.out.println(
                "e.getMessage(): " + e.getMessage());
            System.out.println(
                "e.toString(): " + e.toString());
            System.out.println("e.printStackTrace():");
            e.printStackTrace();
        }
    }
} ///:~
```

Изведеното е:

```
Caught Exception
e.getMessage(): Here's my Exception
e.toString(): java.lang.Exception: Here's my Exception
e.printStackTrace():
java.lang.Exception: Here's my Exception
at ExceptionMethods.main
```

Вижда се, че методите измеждат все повече информация с наследяването – всеки ефективно е надмножество на предишния.

Преизхвърляне на изключение

Понякога ще искате да изхвърлите повторно изключението, което току-що сте хванали, в частност когато сте използвали **Exception** за хващане на което и да е изключение. Понеже вече имате манипулатор към текущото изключение, може просто да изхвърлите този манипулатор:

```
catch(Exception e) {
    System.out.println("An exception was thrown");
    throw e;
}
```

Преизхвърлянето на изключението довежда изключението до обработчиците от следващия по-голям контекст. Всякакъв по-нататъшън **catch** блок за същия **try** блок се игнорира. Освен това всичко за обекта на изключението се съхранява, така че обработчикът от по-високия контекст разполага с всичката информация за изключението.

Ако просто преизхвърлите текущото изключение, информацията която бихте извели за него в **printStackTrace()** ще отразява оригиналното възникване на изключение, а не мястото, където го преизхвърляте. Ако искате да инсталирате нова трасираща информация за стека, може да го направите чрез извикване на **fillInStackTrace()**, което връща обект-изключение чрез наслагването на текущата информация за стека към стария обект. Ето как изглежда:

```
//: c09:Rethrowing.java
// Demonstrating fillInStackTrace()

public class Rethrowing {
    public static void f() throws Exception {
        System.out.println(
            "originating the exception in f()");
        throw new Exception("thrown from f()");
    }
    public static void g() throws Throwable {
        try {
            f();
        } catch(Exception e) {
            System.out.println(
                "Inside g(), e.printStackTrace()");
            e.printStackTrace();
            throw e; // 17
            // throw e.fillInStackTrace(); // 18
        }
    }
    public static void
    main(String[] args) throws Throwable {
        try {
            g();
        } catch(Exception e) {
            System.out.println(
                "Caught in main, e.printStackTrace()");
            e.printStackTrace();
        }
    }
} ///:~
```

Важните номера на редове са дадени в коментар. С ред 17 некоментиран (както е показано), изходът е:

```
originating the exception in f()
Inside g(), e.printStackTrace()
java.lang.Exception: thrown from f()
    at Rethrowing.f(Rethrowing.java:8)
    at Rethrowing.g(Rethrowing.java:12)
    at Rethrowing.main(Rethrowing.java:24)
Caught in main, e.printStackTrace()
java.lang.Exception: thrown from f()
    at Rethrowing.f(Rethrowing.java:8)
    at Rethrowing.g(Rethrowing.java:12)
    at Rethrowing.main(Rethrowing.java:24)
```

Така че трасирането на стека на изключението винаги помни своето място на появяване, без значение колко пъти е преизхвърляно.

С ред 17 коментиран и ред 18 некоментиран се използва този път **fillInStackTrace()** и резултатът е:

```
originating the exception in f()
Inside g(), e.printStackTrace()
java.lang.Exception: thrown from f()
    at Rethrowing.f(Rethrowing.java:8)
    at Rethrowing.g(Rethrowing.java:12)
    at Rethrowing.main(Rethrowing.java:24)
Caught in main, e.printStackTrace()
java.lang.Exception: thrown from f()
    at Rethrowing.g(Rethrowing.java:18)
    at Rethrowing.main(Rethrowing.java:24)
```

Поради **fillInStackTrace()** ред 18 става нова начална точка на изключението.

Класът **Throwable** трябва да се появи в спецификацията на изключениета на **g()** и **main()** понеже **fillInStackTrace()** произвежда манипулатор към **Throwable** обект. Понеже **Throwable** е базов клас на **Exception**, възможно е да се получи обект който е **Throwable** но не **Exception**, така че манипулаторът **Exception** в **main()** би могъл да го пропусне. За да осигури че всичко е наред компилаторът принуждава към спецификация на изключение **Throwable**. Например изключението в следната програма не се хваща в **main()**:

```
//: c09:ThrowOut.java
public class ThrowOut {
    public static void
    main(String[] args) throws Throwable {
        try {
            throw new Throwable();
        } catch(Exception e) {
            System.out.println("Caught in main()");
        }
    }
} ///:~
```

Възможно е също да се преизхвърли различно от хванатото съобщение. Ако направите това получавате подобен ефект като с използването на **fillInStackTrace()**: информацията за оригиналното положение на изхвърлянето е загубена, а имате информацията свързана с новия **throw**:

```
//: c09:RethrowNew.java
```

```

// Rethrow a different object from the one that
// was caught

public class RethrowNew {
    public static void f() throws Exception {
        System.out.println(
            "originating the exception in f()");
        throw new Exception("thrown from f()");
    }
    public static void main(String[] args) {
        try {
            f();
        } catch(Exception e) {
            System.out.println(
                "Caught in main, e.printStackTrace()");
            e.printStackTrace();
            throw new NullPointerException("from main");
        }
    }
} ///:~

```

Изходът е:

```

originating the exception in f()
Caught in main, e.printStackTrace()
java.lang.Exception: thrown from f()
    at RethrowNew.f(RethrowNew.java:8)
    at RethrowNew.main(RethrowNew.java:13)
java.lang.NullPointerException: from main
    at RethrowNew.main(RethrowNew.java:18)

```

Последното изключение знае само че е възникнало в **main()**, а не от **f()**. Забележете че **Throwable** не е непременно в някоя спецификация на изключения.

Никога не се грижим за почистване на предишното изключение, или за каквото и да е изключение. Те са обекти в хийпа създадени с **new**, така че боклучарят автоматично ги почиства всичките.

Стандартни изключения в Java

Java съдържа клас наречен **Throwable** който описва всяко нещо, което може да бъде изхвърлено като изключение. Има два общи типа **Throwable** обекти ("типа" = "наследени от"). **Error** представя системни и грешки по време на компилация, за които не се беспокоите за хващането им (освен в специални случаи). **Exception** е основен тип който може да бъде изхвърлен от всеки от методите в стандартните библиотеки класове на Java и от ваши методи и несполуки по време на изпълнение.

Най-добрия начин да се получи поглед върху изключенията е да се разгледа онлайн Java документацията от <http://java.sun.com>. (Разбира се, по-лесно е първо да се разтовари.) Заслужава си да се направи това веднъж само за да се получи чувство за различните видове изключения, но скоро ще забележите че няма нещо кой знае колко различно между две изключения освен името. Също, броят на изключенията в Java продължава да расте; общо взето е безпредметно да се печатат в книга. Всяка библиотека от трети доставчик също би имала свои собствени изключения вероятно. Важното нещо тук е да се разбере концепцията и вие така ще направите с изключенията.

| `java.lang.Exception`

Това е основният клас на изключение който може да хване вашата програма. Други изключения са извлечени от това. Основната идея е имената на изключенията да отразяват същността на възникналия проблем и да са самоговорящи. Не всички изключения са определени в **java.lang**; някои са за поддръжка на други библиотеки като **util**, **net** и **io**, което може да видите от техните пълни класови имена откъдето са наследени. Например всички IO изключения са наследени от **java.io.IOException**.

Специалният случай **RuntimeException**

Първият пример в тази глава беше

```
if(t == null)  
    throw new NullPointerException();
```

Може да е малко стряскащо да трябва да се проверява всеки път за **null** всеки манипулятор който се подава на метод (понеже не може да се знае дали е подаден валиден манипулятор). За щастие това не е необходимо – то е част от стандартните проверки по време на изпълнение, които Java прави за вас, та ако някое извикване се прави с нулев манипулятор, Java автоматически изхвърля **NullPointerException**. Така че горната проверка е винаги излишна.

Има цяла група типове изключения които попадат в тази категория. Те винаги се изхвърлят автоматично от Java и не е необходимо да ги включвате във вашите спецификации на изключения. Достатъчно удобно те са свързани помежду си чрез единствен базов клас **RuntimeException**, което е перфектен пример на наследяване: основава се фамилия от типове които имат нещо общо в поведението си. Също никога не е необходимо да правите спецификация в която да споменавате че може да се изхвърли **RuntimeException**, понеже това си се предполага. Понеже те индицират бъгове, практически никога не прихващате **RuntimeException** – това е направено автоматично. Ако непременно трябва да проверявате за **RuntimeException** кодът ви би станал объркан. Въпреки че типично не прихващате **RuntimeException**, във ваши собствени пакети може да поискате да изхвърлите **RuntimeException**.

Какво се случва ако не прихванете такива изключения? Понеже компилаторът не ви налага да правите спецификации за тях, твърде правдоподобно е че **RuntimeException** би могъл да се проциди по някакъв начин през вашия **main()** метод без да бъде хванат. За да видите какво става в такъв случай, пробвайте следния пример:

```
//: c09:NeverCaught.java  
// Ignoring RuntimeExceptions  
  
public class NeverCaught {  
    static void f() {  
        throw new RuntimeException("From f()");  
    }  
    static void g0 {  
        f0;  
    }  
    public static void main(String[] args) {  
        g0;  
    }  
} ///:~
```

Може вече да сте видели че **RuntimeException** (или нещо наследено от него) е специален случай, понеже компилаторът не изисква спецификация за тези типове.

Изходът е:

```
java.lang.RuntimeException: From f()
    at NeverCaught.f(NeverCaught.java:9)
    at NeverCaught.g(NeverCaught.java:12)
    at NeverCaught.main(NeverCaught.java:15)
```

Така че отговорът е следният: Ако `RuntimeException` се просмуче чрез `main()` без да бъде хванато, вика се `printStackTrace()` за същото изключение и изпълнението на програмата се прекратява.

Помните че е възможно да се игнорират само `RuntimeException`ите във вашият код, нали за всички други компилаторът грижливо ви заставя да осигурите поддръжка. Основанието е че `RuntimeException` представя програмни грешки:

1. Грешка която не можете да хванете (получаване на нулев манипулятор подаден на ваш метод от клиента-програмист, например)
2. Грешка, която вие, като програмист, трябва да сте проверили в своя код (като `ArrayIndexOutOfBoundsException` където е трявало да внимавате за обхвата на индекса да не се надвиши).

Може да се види какви огромни ползи има от този начин на работа с изключенията, те могат да се използват и за тестване.

Интересно е да се отбележи, че не може да се класифицира за поддръжка на изключения в Java като инструмент с едно приложение. Да, то е проектирано да се справя с онези случаи, когато възникват грешки по време на изпълнение поради причини, които не са подвластни на вашия код, но също е важно и за някои типове програмистки грешки, които компилаторът не може да открие.

Създаване на ваши собствени изключения

Не сте огепеничени да използвате само изключенията на Java. Това е важно, понеже често трябва да се подработи въпросът с някои грешки, които вашата библиотека може да сътвори, които обаче не са могли да бъдат предсказани когато се е създавала йерархията на Java.

За да създадете ваш собствен клас за изключение, принудени сте да наследите от съществуващ тип изключение, за предпочитане от това, което е най-близко по значението си до вашето. Наследяването на изключение е доста просто:

```
//: c09:Inheriting.java
// Inheriting your own exceptions

class MyException extends Exception {
    public MyException() {}
    public MyException(String msg) {
        super(msg);
    }
}

public class Inheriting {
    public static void f() throws MyException {
        System.out.println(
            "Throwing MyException from f()");
        throw new MyException();
    }
}
```

```

}
public static void g() throws MyException {
    System.out.println(
        "Throwing MyException from g()");
    throw new MyException("Originated in g()");
}
public static void main(String[] args) {
    try {
        f();
    } catch(MyException e) {
        e.printStackTrace();
    }
    try {
        g();
    } catch(MyException e) {
        e.printStackTrace();
    }
}
} ///:~

```

Наследяването се случва при създаването на нов клас:

```

class MyException extends Exception {
    public MyException() {}
    public MyException(String msg) {
        super(msg);
    }
}

```

Ключовата фраза тук е **extends Exception**, тоест “това е всичко което е в **Exception** и още.” Добавеният код е малък – добавени са два конструктора които показват как да се създава **MyException**. Помните че компилаторът автоматично извиква конструктора на базовия клас ако не го извикате явно, както в конструктора по подразбиране **MyException()**. Във втория конструктор конструкторът на базовия кас със **String** аргумент явно се извиква с ключовата дума **super**.

Изходът от програмата е:

```

Throwing MyException from f()
MyException
    at Inheriting.f(Inheriting.java:16)
    at Inheriting.main(Inheriting.java:24)
Throwing MyException from g()
MyException: Originated in g()
    at Inheriting.g(Inheriting.java:20)
    at Inheriting.main(Inheriting.java:29)

```

Може да видите отсъствието на детайлно съобщение в изхвърленото от **f()** **MyException**.

Процесът на създаване на ваши собствени изключения може да бъде продължен. Може да добавите конструктори и членове:

```

//: c09:Inheriting2.java
// Inheriting your own exceptions

class MyException2 extends Exception {
    public MyException2() {}
    public MyException2(String msg) {
        super(msg);
    }
}

```

```

}

public MyException2(String msg, int x) {
    super(msg);
    i = x;
}

public int val() { return i; }

private int i;

}

public class Inheriting2 {
    public static void f() throws MyException2 {
        System.out.println(
            "Throwing MyException2 from f()");
        throw new MyException2();
    }

    public static void g() throws MyException2 {
        System.out.println(
            "Throwing MyException2 from g()");
        throw new MyException2("Originated in g()");
    }

    public static void h() throws MyException2 {
        System.out.println(
            "Throwing MyException2 from h()");
        throw new MyException2(
            "Originated in h()", 47);
    }

    public static void main(String[] args) {
        try {
            f();
        } catch(MyException2 e) {
            e.printStackTrace();
        }
        try {
            g();
        } catch(MyException2 e) {
            e.printStackTrace();
        }
        try {
            h();
        } catch(MyException2 e) {
            e.printStackTrace();
            System.out.println("e.val() = " + e.val());
        }
    }
}

} //:~

```

Даннов член **i** е добавен, заедно с метод **къто** чете стойността му и допълнителен конструктор който я задава. Изходът е:

```

Throwing MyException2 from f()
MyException2
    at Inheriting2.f(Inheriting2.java:22)
    at Inheriting2.main(Inheriting2.java:34)

Throwing MyException2 from g()
MyException2: Originated in g()
    at Inheriting2.g(Inheriting2.java:26)
    at Inheriting2.main(Inheriting2.java:39)

```

```
Throwing MyException2 from h()
MyException2: Originated in h()
    at Inheriting2.h(Inheriting2.java:30)
    at Inheriting2.main(Inheriting2.java:44)
e.val() = 47
```

Понеже изключението е само друг вид обект, може да продължите процеса на украсяване на мощта на вашите класове-изключения. Помните, обаче, че цялото това дрешено може да бъде загубено при клиент-програмистите, понеже те биха могли просто да видят дали се е изхвърлило изключение и нищо повече. (Това е начинът, по който се използват повечето от изключениета в библиотеките на Java.) Ако случаят е такъв, възможно е да се създаде нов тип изключение с почти никакъв код:

```
//: c09:SimpleException.java
class SimpleException extends Exception {
} ///:~
```

Това се обляга на задължението на компилатора да създаде конструктор по подразбиране (който автоматично вика конструктора на базовия клас). Разбира се, в този случай нямаете **SimpleException(String)** конструктор, но на практика той не се и използва много.

Ограничения при изключенията

Когато подтискате метод може да изхвърляте само изключенията, зададени за метода от базовия клас. Това е полезно ограничение, понеже води до това, че код, който работи с базовия клас ще работи също така и с който и да е извлечен клас (основна концепция в ООП, разбира се), включително изключенията.

Този пример демонстрира видовете наложени ограничения (по време на компилация) за изключенията:

```
//: c09:StormyInning.java
// Overridden methods may throw only the
// exceptions specified in their base-class
// versions, or exceptions derived from the
// base-class exceptions.

class BaseballException extends Exception {}
class Foul extends BaseballException {}
class Strike extends BaseballException {}

abstract class Inning {
    Inning() throws BaseballException {}
    void event() throws BaseballException {
        // Doesn't actually have to throw anything
    }
    abstract void atBat() throws Strike, Foul;
    void walk() {} // Throws nothing
}

class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}

interface Storm {
    void event() throws RainedOut;
    void rainHard() throws RainedOut;
```

```

}

public class StormyInning extends Inning
    implements Storm {
    // OK to add new exceptions for constructors,
    // but you must deal with the base constructor
    // exceptions:
    StormyInning() throws RainedOut,
        BaseballException {}
    StormyInning(String s) throws Foul,
        BaseballException {}
    // Regular methods must conform to base class:
    //! void walk() throws PopFoul {} //Compile error
    // Interface CANNOT add exceptions to existing
    // methods from the base class:
    //! public void event() throws RainedOut {}
    // If the method doesn't already exist in the
    // base class, the exception is OK:
    public void rainHard() throws RainedOut {}
    // You can choose to not throw any exceptions,
    // even if base version does:
    public void event() {}
    // Overridden methods can throw
    // inherited exceptions:
    void atBat() throws PopFoul {}
    public static void main(String[] args) {
        try {
            StormyInning si = new StormyInning();
            si.atBat();
        } catch(PopFoul e) {
        } catch(RainedOut e) {
        } catch(BaseballException e) {}
        // Strike not thrown in derived version.
        try {
            // What happens if you upcast?
            Inning i = new StormyInning();
            i.atBat();
            // You must catch the exceptions from the
            // base-class version of the method:
        } catch(Strike e) {
        } catch(Foul e) {
        } catch(RainedOut e) {
        } catch(BaseballException e) {}
    }
} ///:~

```

В **Inning** може да видите че както конструкторът така и методът **event()** казват че ще изхвърлят изключение но не го правят. Това е законно понеже кара потребителя да прихване всяко изключение което би могло да се появи в подтиснатите версии на **event()**. Същата идея важи и за **abstract** методи, както се вижда в **atBat()**.

interface Storm е интересен с това, че съдържа един метод (**event()**) дефиниран в **Inning** и един метод който не е. И двата метода изхвърлят нов тип изключение, **RainedOut**. Когато **StormyInning extends Inning** и **implements Storm**, ще видите че методът **event()** в **Storm** не може да смени интерфейса на изключението **event()** в **Inning**. Отново това има смисъл понеже иначе никога няма да е сигурно дали имате провилното нещо когато хващате изключение от

базовия клас. Разбира се ако метод описан в **interface** не е в базовия клас, както **rainHard()**, няма проблеми ако той изхвърля изключения.

Ограничението на изключенията не се прилага за конструкторите. В **StormyInning** може да видите, че конструкторът може да изхвърли всичко каквото иска, без значение какво изхвърля конструкторът на базовия клас. Обаче тъй като базовият конструктор се вика винаги по един или друг начин (тука конструктор по подразбиране се вика автоматично), конструкторът на извлечения клас трябва да декларира всички изключения на конструктора на базовия клас в своята спецификация на изключения.

Причината да не се компилира **StormyInning.walk()** е че изхвърля изключение, докато **Inning.walk()** не е. Ако това беше допустимо, можеше да се напише код който вика **Inning.walk()** и той нямаше да е нужно да обработи изключения, но когато замените обект от клас низвлечен от **Inning**, биха били изхвърляни изключения и кодът щеше да се скапе. Чрез заставянето на методите на извлечения клас да отговарят на спецификацията на изключенията на базовия клас се обслужва взаимозаменяемостта на методите.

Подписанятия **event()** метод показва, че версията на метод в извлечен клас може да избере въобще да не изхвърля изключения, макар и версията на базовия клас да го прави. Отново това е необходимото за да работи с всякакъв наследен код. Подобна логика важи за **atBat()**, който изхвърля **PopFoul**, изключение извлечено от **Foul** изхвърлено от базовата версия на **atBat()**. По този начин ако някой напише код кайто работи с **Inning** и вика **atBat()**, те трябва да хванат **Foul** изключението. Понеже **PopFoul** е извлечено от **Foul**, обработчикът ще хване и **PopFoul**.

Последната интересна точка е в **main()**. Тук се вижда че ако се разправяте точно с **StormyInning** обект, компилаторът ви заставя да хващате изключенията специфични точно за него, но ако направите ъпкаст към базовия клас компилаторът (коректно) ви заставя да хващате изключенията на базовия тип. Всички тези ограничения довеждат до много по-добър код за обработка на изключения.³

Полезно е да се разбере, че макар и ограниченията при изключенията да се налагат от компилатора чрез наследяване, спецификациите на ограничения не са част от даден тип, който е съставен само от името и типовете на аргументите. Затова не може да се пренатоварват методи чрез спецификацията на изключения. Освен това ако има спецификация на изключенията в базовия метод това не значи че непременно тя съществува и в извлечения, а това е доста различно от наследяване на методи (тоест, метод в базовия клас трябва също да съществува и в извлечения клас). С други думи казано, “интерфейсът на спецификация на изключенията” за конкретен метод може да се стеснява в процеса на наследяване и подискане, но не може да се разширява – това е точно наопаки на интерфейса на базовия и извлечения клас при наследяване.

Финализиране с **finally**

Често има парче код, което бихте искали да се изпълни в **try** блока, независимо дали има или не изключение. То обикновено е свързано с операции различни от освобождаването на рачет (понеже за последното се грижи боклучарят). За да се постигне това се използва клаузата **finally**⁴ в края на всички обработчици на изключения. Така пълната картина на кода за обработка на изключения е следната:

³ ANSI/ISO C++ added similar constraints that require derived-method exceptions to be the same as, or derived from, the exceptions thrown by the base-class method. This is one case [where-in which](#) C++ is actually able to check exception specifications at compile time.

⁴ C++ exception handling does not have the **finally** clause because it relies on destructors to accomplish this sort of cleanup.

```

try {
    // Охраняваният регион:
    // Опасни неща които могат да изхвърлят A, B или C
} catch (A a1) {
    // Манипулятор A
} catch (B b1) {
    // Манипулятор B
} catch (C c1) {
    // Манипулятор C
} finally {
    // Парченцето код, което трябва да се изпълни винаги
}

```

За да се демонстрира че **finally** винаги се пуска, опитваме следната програма:

```

//: c09:FinallyWorks.java
// The finally clause is always executed

public class FinallyWorks {
    static int count = 0;
    public static void main(String[] args) {
        while(true) {
            try {
                // post-increment is zero first time:
                if(count++ == 0)
                    throw new Exception();
                System.out.println("No exception");
            } catch(Exception e) {
                System.out.println("Exception thrown");
            } finally {
                System.out.println("in finally clause");
                if(count == 2) break; // out of "while"
            }
        }
    }
} ///:~

```

Тази програма показва и начин какво да правите с факта, че изключенията в Java (подобно на C++) не позволяват да се продължи от мястото, където изключението е било изхвърлено, както беше дискутирано преди. Ако сложите вашия **try** блок в цикъл, може да носочите условие, което трябва да бъде изпълнено, преди да продължи програмата. Може също да добавите **static** бояч или нещо друго подходящо, което да позволи на цикъла да пробва няколко подхода, преди да се откаже. По този начин може да се постигне по-високо ниво на "робастност" с вашата програма.

Изходът е:

```

Exception thrown
in finally clause
No exception
in finally clause

```

Било или не изхвърлено изключение, клаузата на **finally** винаги се изпълнява.

За какво е **finally**?

В език без боклукосъбиране и без автоматично извикване на деструктори,⁵ **finally** е важна, понеже дава възможност на програмиста да гарантира освобождаването на паметта без значение какво се е случило в **try** блока. Но Java има събиране на боклука, така че освобождаването на паметта практически никога не е проблем. Също, няма деструктори за извикване. Кога има нужда от **finally** в Java?

finally е необходима когато трябва нещо различно от памет да се сложи обратно в предишното му състояние. Това е обикновено нещо от рода на отворен файл или мрежова връзка, нещо начертано на еcran или даже превключвател в отвъдния, реалния свят, както е моделирано в следния пример:

```
//: c09:OnOffSwitch.java
// Why use finally?

class Switch {
    boolean state = false;
    boolean read() { return state; }
    void on() { state = true; }
    void off() { state = false; }
}

public class OnOffSwitch {
    static Switch sw = new Switch();
    public static void main(String[] args) {
        try {
            sw.on();
            // Code that can throw exceptions...
            sw.off();
        } catch(NullPointerException e) {
            System.out.println("NullPointerException");
            sw.off();
        } catch(IllegalArgumentException e) {
            System.out.println("IOException");
            sw.off();
        }
    }
} ///:~
```

Целта тук е ключът да бъде изключен когато **main()** завърши, така че **sw.off()** се слага на края на трай блока на края на обработчика. Възможно е обаче да е изхвърлено изключение което да не е прихванато точно тук, така че **sw.off()** ще се пропусне. С **finally** може да сложите завършващия код на трай блока в само едно място:

```
//: c09:WithFinally.java
// Finally Guarantees cleanup

class Switch2 {
    boolean state = false;
    boolean read() { return state; }
    void on() { state = true; }
    void off() { state = false; }
}
```

⁵ A destructor is a function that's always called when an object becomes unused. You always know exactly where and when the destructor gets called. C++ has automatic destructor calls, but Delphi's Object Pascal versions 1 &and 2 do not (which changes the meaning and use of the concept of a destructor for that language).

```

public class WithFinally {
    static Switch2 sw = new Switch2();
    public static void main(String[] args) {
        try {
            sw.on();
            // Code that can throw exceptions...
        } catch(NullPointerException e) {
            System.out.println("NullPointerException");
        } catch(IllegalArgumentException e) {
            System.out.println("IOException");
        } finally {
            sw.off();
        }
    }
} ///:~

```

Тук **sw.off()** е преместено на въпросното място, където гарантирано ще се изпълни без значение какво ще се случи.

Даже и в случаите когато изключението не е хванато в текущото множество на **catch** клаузи, **finally** ще се изпълни преди механизъмът за поддръжка на изключениета да продължи търсенето си на обработчик в следващия по-широк контекст:

```

//: c09:AlwaysFinally.java
// Finally is always executed

class Ex extends Exception {}

public class AlwaysFinally {
    public static void main(String[] args) {
        System.out.println(
            "Entering first try block");
        try {
            System.out.println(
                "Entering second try block");
            try {
                System.out.println(
                    "finally in 2nd try block");
            }
        } catch(Ex e) {
            System.out.println(
                "Caught Ex in first try block");
        } finally {
            System.out.println(
                "finally in 1st try block");
        }
    }
} ///:~

```

Изходът от тази програма показва какво се случва:

```

Entering first try block
Entering second try block
finally in 2nd try block
Caught Ex in first try block

```

```
| finally in 1st try block
```

Операторът **finally** ще се изпълни в ситуации където има **break** и **continue** оператори. Забележете че заедно с етикетирания **break** и **continue**, **finally** елиминира нуждата от **goto** оператор в Java.

Капан: загубеното изключение

Изобщо, реализацията на изключениета в Java е забележителна, но за нещастие има пукнатина. Въпреки че изключениета са признак за криза с вашата програма и не бива да се игнорират, възможно е просто да се загуби изключение. Това се случва в конкретна ситуация с използване на **finally** клаузата:

```
//: c09:LostMessage.java
// How an exception can be lost

class VeryImportantException extends Exception {
    public String toString() {
        return "A very important exception!";
    }
}

class HoHumException extends Exception {
    public String toString() {
        return "A trivial exception";
    }
}

public class LostMessage {
    void f() throws VeryImportantException {
        throw new VeryImportantException();
    }
    void dispose() throws HoHumException {
        throw new HoHumException();
    }
    public static void main(String[] args)
        throws Exception {
        LostMessage lm = new LostMessage();
        try {
            lm.f();
        } finally {
            lm.dispose();
        }
    }
} ///:~
```

Изходът е:

```
A trivial exception
at LostMessage.dispose(LostMessage.java:21)
at LostMessage.main(LostMessage.java:29)
```

Може да се види, че няма следа от **VeryImportantException**, което просто е заместено от **HoHumException** във **finally** клаузата. Това е достатъчно сериозен капан, понеже изключение може да бъде напълно изгубено, също и в много по-засукана и трудна за проследяване ситуация отколкото горната. За контраст, C++ третира ситуацията в която се изхвърля едно изключение преди да е приключила обработката на друго като ужасяваща програмна

грешка. Може би някоя бъзеща версия на Java ще оправи проблема. (Горните резултати се произведоха с Java 1.1.)

Конструктори

Когато се пише код за изключения особено важно е да се питаме "ако се случи изключение, това ще бъде ли правилно почиствено?" Повечето време е съвсем безопасно, но с конструкторите има проблем. Конструкторът извежда обекта до безопасно стартово състояние, но би могъл да изпълнява и никаква операция – като отваряне на файл – която да не бъде почиствена (откъм въздействията се - б.пр.) докато потребителят не изпълни специално написан за това код. Ако изхвърлите изключение извънре на конструктор този код може и да не се изпълни, почистването да не стане правилно. Това значи че трябва да бъдете особено прилежни когато пишете своя конструктор.

Понеже току-що научихте за **finally**, може да помислите, че това е коректното решение. Не е така просто, обаче, понеже **finally** изпълнява кода **всякога**, даже и в ситуации, когато не искате да се изпълнява почистване докато не се пусне почистващия метод. Така, ако почиствате във **finally**, трябва да сложите някакъв флаг (семафор) ако конструкторът завърши нормално и само тогава да почиствате (ако е сложен). Тъй като това не е много елегантно (свързвате едно място на кода си с друго), най-добре е да избягвате да правите този вид почистване във **finally** докато не ви се наложи.

В следващия пример се създава клас наречен **Inputfile**, той отваря файл и дава възможност да се прочете един ред от него (превърнат в **String**) наведнъж. Използвани са класовете **FileReader** и **BufferedReader** от стандартната входно-изходна библиотека на Java която ще се разгледа в глава 10, но които са достатъчно прости така че няма да имате проблеми с разбирането на основната им употреба:

```
//: c09:Cleanup.java
// Paying attention to exceptions
// in constructors
import java.io.*;

class Inputfile {
    private BufferedReader in;
    Inputfile(String fname) throws Exception {
        try {
            in =
                new BufferedReader(
                    new FileReader(fname));
            // Other code that might throw exceptions
        } catch(FileNotFoundException e) {
            System.out.println(
                "Could not open " + fname);
            // Wasn't open, so don't close it
            throw e;
        } catch(Exception e) {
            // All other exceptions must close it
            try {
                in.close();
            } catch(IOException e2) {
                System.out.println(
                    "in.close() unsuccessful");
            }
            throw e;
        } finally {
```

```

        // Don't close it here!!!
    }
}

String getLine() {
    String s;
    try {
        s = in.readLine();
    } catch(IOException e) {
        System.out.println(
            "readLine() unsuccessful");
        s = "failed";
    }
    return s;
}

void cleanup() {
    try {
        in.close();
    } catch(IOException e2) {
        System.out.println(
            "in.close() unsuccessful");
    }
}
}

public class Cleanup {
    public static void main(String[] args) {
        try {
            InputFile in =
                new InputFile("Cleanup.java");
            String s;
            int i = 1;
            while((s = in.getLine()) != null)
                System.out.println(""+ i++ + ":" + s);
            in.cleanup();
        } catch(Exception e) {
            System.out.println(
                "Caught in main, e.printStackTrace()");
            e.printStackTrace();
        }
    }
}
} ///:~

```

Този пример използва класовете на Java 1.1 вход-изхода.

Конструкторът на **InputFile** взема **String** аргумент, който е името на файла, който ще се отваря. Вътре в **try** блока той създава **FileReader** използвайки името. **FileReader** не е особено полезен докато не го използвате за създаване на **BufferedReader** на който фактически може да говорите – забележете че една от ползите от **InputFile** е че комбинира тези две действия.

Ако конструктора на **FileReader** е неуспешен, той изхвърля **FileNotFoundException**, което трябва да се хване отделно, понеже иначе ще искате да затваряте файл който не е бил отворен. Всякакви други **catch** клаузи трябва да затворят файла понеже той е бил отворен към времето, когато се е влизало в тях. (Разбира се, още по-измамно е ако повече от един метод би могъл да изхвърли **FileNotFoundException**. В този случаи може да поискате да разделяте нещата в няколко **try** блока.) Методът **close()** изхвърля изключение което се опитва и хваща даже и да е вътре в друга **catch** клауза – това е само друга двойка фигурни скоби за компилатора на Java. След изпълнение на локалните операции изключението се

презхвърля, което е подходящо понеже този конструктор се е провалил, а вие не искате извикваният метод да смята, че всичко е наред.

В този пример, който не използва споменатата преди техника с флаговете, **finally** клаузата определено не е мястото за **close()** файла, повеже ще го затваря всеки път когато конструкторът завършва работата си. Тъй като искаме файлът да бъде отворен за полезна работа времето на живот на **InputFile** обекта не би било подходящо в такъв случай.

Методът **getLine()** връща **String** съдържащ следващия ред от файла. Той вика **readLine()**, който може да изхвърли изключение, но то се хваща, така че **getLine()** не изхвърля никакви изключения. Един от въпросите къто възникват с изключенията е дали да се обработват изцяло на това ниво, или да се обработят частично и да се подаде същото изключение (или някое друго) нататък, или просто да се предаде нататък. Предаването му, когато е уместно, определено може да оправи програмирането. **getLine()** методът става:

```
String getLine() throws IOException {  
    return in.readLine();  
}
```

Но разбира се, извикващият метод сега е отговорен за бработката на всякакво **IOException** което би могло да възникне.

cleanup() методът трябва да бъде извикано от потребителя когато свърши използването на **InputFile** обекта за да се освободят заеманите системни ресурси (като файлови манипулатори) заети чрез **BufferedReader** и/или **FileReader** обекти.⁶ Не искаме да правим това докато не свършим с **InputFile** обекта, точката, в която го искаме. Може да се мисли за слагане на такава функционалност във **finalize()** метод, но както се спомена в глава 4 не може да бъдем сигурни, че ще се извика някога **finalize()** (даже и да можете да бъдете сигурни че ще се извика, не знаете кога). Това е един от минусите на Java – всички финални действия освен освобождаването на паметта не стават автоматично, така че трябва да информирате клиентите-програмисти че са отговорни, та може би да гарантират нещата чрез използване на **finalize()**.

В **Cleanup.java** се създава **InputFile** за да отвори сорса на тази програма, файлът се чете ред по ред, добавят се номера на редовете. Всички изключения се хващат родово в **main()**, макар и да бихте избрали, може би, по-голяма нееднородност.

Една от ползите на този пример е да се покаже защо изключенията бяха въведени точно на това място в книгата. Изключенията са толкова интегрирани в програмирането на Java, особено понеже компилаторът ги прави задължителни, че може да свършите само толкова (колкото знаете до тази глава-б.пр.) без да ги познавате.

Търсене на обработчик

Когато е изхвърлено съобщение системата, обслужваща изключенията преглежда "наблизките" обработчици в реда, в който са написани. Когато намери съвпадение, изключението се счита уредено, по-нататъшно търдене не се прави.

Съвпадането не изисква точно съответствие между изключението и обработчика. Обект от извлечен клас ще стане за обработчик предназначен за базовия клас, както е показано в примера:

```
//: c09:Human.java  
// Catching Exception Hierarchies  
  
class Annoyance extends Exception {}
```

⁶ In C++, a *destructor* would handle this for you.

```

class Sneeze extends Annoyance {}

public class Human {
    public static void main(String[] args) {
        try {
            throw new Sneeze();
        } catch(Sneeze s) {
            System.out.println("Caught Sneeze");
        } catch(Annoyance a) {
            System.out.println("Caught Annoyance");
        }
    }
} //:~

```

Изключението **Sneeze** ще се хване в първата **catch** клауза за която става, като това е първата такава, разбира се. Ако обаче махнете първата клауза:

```

try {
    throw new Sneeze();
} catch(Annoyance a) {
    System.out.println("Caught Annoyance");
}

```

Оставащата ще действа, понеже ще хване базовия клас на **Sneeze**. С други думи, **catch(Annoyance e)** ще хване **Annoyance** и всеки клас извлечен от него. Това е полезно понеже ако решите да добавите изключения към някой метод, ако те са наследени от същия базов клас, клиентският код няма да изисква промяна, като се предполага, че той хваща базовия клас, най-малкото.

Ако се опитате да "маскирате" изключенията на извлечения клас чрез слагане на клаузата на базовия клас първа, както тук:

```

try {
    throw new Sneeze();
} catch(Annoyance a) {
    System.out.println("Caught Annoyance");
} catch(Sneeze s) {
    System.out.println("Caught Sneeze");
}

```

компилаторът ще издаде съобщение за грешка, понеже вижда че **catch**-клаузата на **Sneeze** не може никога да бъде достигната.

Правила за изключенията

Използвайте изключенията за да:

1. Оправите проблема и извикате четода (който е предизвикал изключението) пак.
2. Закърпите нещата и продължите без да викате метода пак.
3. Получите резултат, различен от този който се получава по обичайния начин.
4. Направите каквото може в този контекст и преизхвърлите същото изключение към по-висок контекст.

5. Направите каквото може в този контекст и изхвърлите друго изключение към по-високия контекст.
6. Прекратите програмата.
7. Опростите. Ако вашата схема на изключения усложнява нещата, тя е болезнена и вбесяваща при използване.
8. Направете библиотеката и програмата си по-сигурни. Това е краткосрочна инвестиция (за тестването) и дългосрочна (за готовата програма).

Резюме

Развитото възстановяване след грешки е един от най-мощните начини да се повиши качеството на програмите. Възстановяването от грешки е основно за всяка програма която пишете, а е особено важно в Java, където основна дейност е да се създават части от програми за използване от други хора. За да се създаде качествена система всеки компонент трябва да бъде качествен.

Целите на поддръжката на изключения в Java са да се опрости създаването на големи, надеждни програми чрез писане на по-малко код отколкото в момента е възможно, с по-голяма увереност, че вашата програма няма да допусне грешка, от която няма да се възстанови.

Изключенията не са ужасяващо трудни за научаване, те са една от онези черти, които дават непосредствени и значителни ползи за вашия проект. За щастие Java налага всички аспекти така че те ще бъдат смислено използвани както от проектанта на библиотеката, така и от клиента-програмист.

Упражнения

1. Създайте клас в `main()` който изхвърля обект от клас `Exception` вътре в `try` блок. Дайте на конструктора на `Exception` стрингов аргумент. Хванете изключението вътре в `catch` клауза и изведете стринговия аргумент. Добавете клауза `finally` и изведете съобщение за да докажете че сте били там.
2. Създайте ваш собствен клас-изключение чрез ключовата дума `extends`. Напишете конструктор за този клас който взема `String` аргумент и го запомня вътре в обекта чрез `String` манипулатор. Напишете метод който извежда запомнения `String`. създайте try-catch клауза за изпитание на новото ви изключение.
3. Напишете клас с метод, който изхвърля изключението от упражнение 2. Опитайте да го компилирате без спецификация на изключенията за да видите какво ще каже компилаторът. Добавете подходяща спецификация на изключението. Изprobвайте вашият клас и изключението в try-catch клауза.
4. Намерете в глава 5 двете програми наречени `Assert.java` и ги променете да изхварят собствени изключения вместо да пишат чрез `System.err`. Изключението да бъде вътрешен клас който разширява `RuntimeException`.

10: Входно-изходна система на Java

Създаването на добра входно/изходна (IO) система е една от по-трудните задачи на проектанта на езика.

Това се вижда от бряг на различните подходи. Прадизвикателството каточели е да се покрият всички възможности. Не само че има различни видове вход/изход (IO) с които искате да комуникирате (файлове, конзолата, мрежови връзки), но искате да им говорите по най-разнообразни начини (последователно, с произволен достъп, двоично, знаково, на редове, на думи и т.н.).

Проектантите на библиотеките на Java атакуваха проблема чрез създаване на множество класове. Фактически има толкова много класове в IO системата на Java че може да се уплашите в началото (иронично, дизайнът на Java IO фактически предотвратява експлозията на класовете). Има също значителна промяна в библиотаките IO между Java 1.0 и Java 1.1. Вместо просто да заместят старата библиотека с нова, проектантите в Sun разшириха старата и добавиха успоредно с нея и нова. Като резултат мож понякога да смесвате старата и новата библиотека и да създавате още по-плашещ код.

Тази глава ще ви помогне да разберете разнообразните IO в стандартната библиотека на Java и как да ги използвате. Първата част на главата ще ви запознае със "старата" потокова IO библиотека на Java 1.0, понеже има значително количество съществуващ код, който я използва. Останалата част на главата разглежда новите черти на IO библиотеката на Java 1.1. Забележете че когато компилирате нещо от първата част на главата с компилатор за Java 1.1 може да получите съобщение-предупреждение "deprecated feature" по време на компилация. Кодът си работи; компилаторът просто ви съветва да използвате някои нови черти, описани във втората част на главата. Ценно е, обаче, да се види разликата между двете библиотеки чрез правене на неща по двата начина и това е причината първата част на главата да остане – за да повиши разбирането (и за да ви позволи да четете код написан за Java 1.0).

Вход и изход

Библиотечните класове IO на Java са разделени за вход и изход, както може да видите с вашия броузър от йерархията онлайн на класовете на Java. По наследство всичките класове извлечени от **InputStream** имат основни методи наречени **read()** за четене на единствен байт или масив от байтове. Подобно, всички класове наследени от **OutputStream** имат основни методи наречени **write()** за четене на единствен байт или масив от байтове. Обаче общо взето няма да използвате тези методи; те съществуват за да може по-усложнените класове да ги използват, като се осигурява по-удобен интерфейс. Така рядко ще създавате потоков обект чрез използване на един само клас, а ще слагате няколко обекта заедно за да се постигне желаната функционалност. Фактът че използвате няколко класа за създаване на обект за един поток е основната причина потоковата библиотека на Java да смущава.

Полезно е да се категоризират класовете по тяхната функционалност. Проектантите са започнали с мисълта че всичко което ще има работа с входа ще се наследява от **InputStream** и всичко за изход — от **OutputStream**.

Типове на `InputStream`

Задачата на `InputStream` е да представи класовете които въвеждат от различни източници. Източниците могат да бъдат (и всеки има асоцииран подклас на `InputStream`):

1. Масив от битове
2. `String` обект
3. Файл
4. "тръба," която работи като физически съществуваща тръба: слагате неща от едната страна и те излизат от другата
5. Последователност от други потоци, така че да може да ги съберете в един поток
6. Други източници, такива като Internet връзка. (Този ще бъде разискван в по-късна глава.)

Освен това, `FilterInputStream` е също тип `InputStream`, за да даде базов клас за "декориращи" класове които закачат атрибути или полезни интерфейси на входен поток. Това е дискутирано по-нататък.

Table 10-1. Типове `InputStream`

Клас	Функция	Аргументи на конструктора
		Как да се използва
<code>ByteArray-InputStream</code>	Позволява буфер в паметта да се използва като <code>InputStream</code> .	Буфер от който да се извлекат байтовете.
		Като източник на данни. Свържете го с <code>FilterInputStream</code> обект за да имате полезен интерфейс.
<code>StringBuffer-InputStream</code>	Превръща <code>String</code> в <code>InputStream</code> .	<code>String</code> . Подлежащата реализация фактически използва <code>StringBuffer</code> .
		Като източник на данни. Свържете го към <code>FilterInputStream</code> обект за да даде полезен интерфейс.
<code>File-InputStream</code>	За четене на информация от файл.	<code>String</code> представящ файловото име или <code>File</code> или <code>FileDescriptor</code> обект.
		Като източник на данни. Свържете го към <code>FilterInputStream</code> обект за да даде полезен интерфейс.

Piped-InputStream	Дава данните които са били написани в PipedOutputStream . Реализира концепцията за "piping".	PipedOutputStream
		Като източник на данни при многонишковост. Свържете го с FilterInputStream обект за да даде полезен интерфейс.
Sequence-InputStream	Превръща два или повече InputStream обекта в единственный InputStream .	Два InputStream обекта или Enumeration за контейнер от InputStream обекти. Като източник на данни. Свържете го с FilterInputStream обект за да даде полезен интерфейс.
Filter-InputStream	Абстрактният клас който служи като интерфейс към декоратори, които дават полезни черти на InputStream класовете. Вж. Табл. 10-3.	Вж. Табл. 10-3. Вж. Табл. 10-3.

Типове OutputStream

Тази категория съдържа класове които решават къде ще ходят вашите данни: масив от байтове (не **String**, обаче; предполагамо може да създадете такъв от масив от байтове), файл или "тръба."

Освен това обектът **FilterOutputStream** дава базов клас за "декориращи" класове които задават атрибути или полезна функционалност на изходните потоци. Това се дискутира по-късно.

Таблица 10-2. Типове OutputStream

Клас	Функция	Аргументи на конструктора
		Как да се използва
ByteArray-OutputStream	Създава буфер в паметта. Всички данни които изпращате към потока се слагат в този буфер.	Опционна дължина на буфера
		Определя направление за данните ви. Свържете го с FilterOutputStream обект за полезен интерфейс.
File-OutputStream	За изпращане на информация във файл.	String представящ името на файла или File или FileDescriptor обект.
		Определя направление за данните ви. Свържете го с FilterOutputStream обект за полезен интерфейс.
Piped-OutputStream	Всяка информация записана тук се оказва вход за асоциирания PipedInputStream . Реализира "тръбната" концепция.	PipedInputStream
		Определя направление за данните ви при многонишковост. Свържете го с FilterOutputStream обект за полезен интерфейс.
Filter-OutputStream	Абстрактен клас който е интерфейс към декоратори, които дават полезни черти на OutputStream класовете. Виж Таблица 10-4.	Виж Таблица 10-4.
		Виж Таблица 10-4.

Добавяне на атрибути и полезни интерфейси

Използването на слоеве от обекти за динамично и прозрачно добавяне на отговорности към дадени обекти се нарича декораторски шаблон. (Шаблоните¹ са тема на глава 16.) Декораторският шаблон определя, че всички обекти които обивват вашия ще имат един и същ интерфейс, за да се направи използването на декораторите прозрачно – едно и също съобщение се изпраща към обекта, независимо дали е декориран или не е. Това е основанието за съществуване на "filtъrnите" класове в IO библиотеката на Java: абстрактният "filtърен" е базов клас за всички декоратори. (Един декоратор трябва да има

¹ In *Design Patterns*, Erich Gamma et al., Addison-Wesley 1995. Described later in this book.

същия интерфейс като обекта който декорира, но декораторът също може и да разшири интерфейса, което и става в някои от "фильтрните" класове).

Декораторите често се употребяват когато е необходимо да се поддържат всички необходими комбинации от подкласове – толкова много на брой, че субкласингът е непрактичен. IO библиотеката на Java изиска много комбинации на различни черти, което прави декораторският подход практичен. Има обаче и недостатък с декораторския шаблон. Декораторите дават много гъвкавост когато пишете програмата (понеже може лесно да смесвате и нагласявате атрибути), но се добавя сложност на кода ви. Причината IO библиотеката на Java да е тромава при използване е че трябва да създадете много класове – "чистия" IO тип плюс всичките декоратори – за да може да се сдобиете с единствения IO обект който ви трябва.

Класовете които дават декораторския интерфейс към конкретен **InputStream** или **OutputStream** са **FilterInputStream** и **FilterOutputStream** – които нямат много интуитивни имена. Те са извлечени, респективно, от **InputStream** и **OutputStream**, абстрактни класове са, на теория да дават общ интерфейс на всичките начини, по които искате да говорите на потока. На практика **FilterInputStream** и **FilterOutputStream** просто подражават на базовите си класове, което е ключово изискване за един декоратор.

Четене от **InputStream** с **FilterInputStream**

Класовете **FilterInputStream** правят две много различни неща. **DataInputStream** подволява да се четат различни примитивни типове данни както и **String** обекти. (Всички методи започват с "read," както **readByte()**, **readFloat()** и т.н.) Така, заедно със съществуващи ги **DataOutputStream**, позволява да се преместват данни от едно място на друго посредством поток. Тези "места" се определят от класовете в Таблица 10-1. Ако четете данните на блокове и ги разделяте сами, не се нуждаете от **DataInputStream**, но в повечето други случаи ще го използвате за форматиране на данните при четене.

Останалите класове променят начина на вътрешното поведение на **InputStream**: дали да е буфериран или не, да следи ли прочетените линии (позволявайки ви да го питате за номер на ред и да задавате номер на ред) и дали може да се слага един знак обратно в потока. Последните два класа много приличат на такива за поддържане на компилатор (тоест, те са били добавине за постройката на Java компилатор), така че вероятно няма да ги използвате в общото програмиране.

Вероятно ще искате буфер почти винаги, независимо с какво IO устройство сте се скачали, така че по-смислено щеше да бъде специалният случай в IO библиотеката да бъде небуферирания вход, а не буферирания.

Таблица 10-3. Типове FilterInputStream

Клас	Функция	Аргументи на конструктора
		Как да се използва
DataInputStream	Използван заедно с DataOutputStream , така че може да четете примитиви (int, char, long, и т.н.) от поток по преносим начин.	InputStream Съдържа пълен интерфейс за четене на примитиви.

Buffered-InputStream	Използвайте го за предотвратяване на физическото четене всеки път когато трябват още данни. Казвате "Използвай буфер."	InputStream , с дължина на буфера по желание.
LineNumber-InputStream	Следи числата във входния поток; може да извикате getLineNumber() и setLineNumber(int) .	InputStream Просто добавя номера на линии, така че вероятно ще присъедините интерфейсен обект.
Pushback-InputStream	Има еднобайтов буфер така че може да бутнете обратно прочетен знак.	InputStream Изобщо се използва като скенер за компилатор и вероятно е включен заради Java компилатора. Вероятно няма да го използвате.

Писане в **OutputStream** с **FilterOutputStream**

Допълнителен към **DataInputStream** е **DataOutputStream**, който форматира всеки от римитивните типове и **String** обекти в поток по такъв начин, че **DataInputStream**, на каквато и да е машина, може да ги чете. Всичките методи започват с "write," като **writeByte()**, **writeFloat()** и т.н.

Ако искате да направите истински форматирано извеждане, например към конзолата, използвайте **PrintStream**. Това е крайната точка която позволява да се извеждат всички примитивни типове и **String** обекти във видим формат като противоположност на **DataOutputStream**, чиято цел е да се сложат в поток така, че **DataInputStream** преносимо да може да ги реконструира. Статичният обект **System.out** е **PrintStream**.

Двета важни метода в **PrintStream** са **print()** и **println()**, които са претоварени да извеждат всички примитивни типове. Разликата между **print()** и **println()** е че последният добавя край на ред когато се изпълни.

BufferedOutputStream е модifikатор и въвежда използването на буфер, така че да не е нужно физическо писане при всяко писане в поток. Вероятно винаги ще искате да го използвате с файлове, а възможно е и с конзолен IO.

Таблица 10-4. Типове **FilterOutputStream**

Клас	Функция	Аргументи на конструктора
		Как да се използва
DataOutputStream	Заедно с DataInputStream така че можете да пишете примитиви (int, char, long, и т.н.) в поток по преносим начин.	OutputStream Съдържа пълен интерфейс да позволи писането на всички примитивни типове.
PrintStream	За форматилар изход. Докато DataOutputStream е за запомняне на данни, PrintStream е за изобразяване.	OutputStream , с буleva ктойност по желание, която индицира дали буферът ще се изпразва след всеки нов ред. Трябва да бъде "final" обгръщащ вашия OutputStream обект. Вероятно ще го използвате много.
BufferedOutputStream	Използвайте го за избягване на физическото писане при добавяне на всяко късче данни. Казвате "Използвай буфер." Може да използвате flush() за запис и изпразване на буфера.	OutputStream , с дължина на буфер по желание. Не дава интерфейс reset , само определя използването на буфер. Присъединете интерфейсен обект.

Отделен: RandomAccessFile

RandomAccessFile се използва за файлове състоящи се от записи с известна дължина, така че може да се придвижвате от някакъв запис на друг чрез **seek()**, да го четете и презаписване. Записите не трябва непременно да са с еднаква дължина; само трябва да се знае колко са големи и къде във файла се намират.

Отначало е малко трудно да се повярва че **RandomAccessFile** не е част от **InputStream** или **OutputStream** йерархията. Той няма отношения с тези йерархии освен че се случва да използва **DataInput** и **DataOutput** интерфейсите (които също са реализирани от **DataInputStream** и **DataOutputStream**). Даже не използва нищо от функционалността на **InputStream** или **OutputStream** класовете – това е напълно отделен клас, написан от нулата, с изцяло собствени (най-вече native) методи. Причината за това може да е че **RandomAccessFile** има изоснови различно от другите IO типове поведение, доколкото може да се движите напред-назад във файла. Във всеки случай, той стои самостоятелно, като директен наследник на **Object**.

Основно, **RandomAccessFile** работи подобно на **DataInputStream** заедно с **DataOutputStream** и методите **getFilePointer()** за да намери къде сте във файла, **seek()** за придвижване към нова точка от файла, **length()** за определяне на максималната дължина на файла. Освен това

конструкторите изискват втори аргументи (като `fopen()` в C) показващ дали четете ("r") или четете и пишете ("rw"). Няма поддръжка за файлове само за четене, което означава, че **RandomAccessFile** би могъл да работи добре ако беше наследен от **DataInputStream**.

Още по-разочароващото е, че лесно може да се въобразим произволен достъп в различни потоци, като **ByteArrayInputStream** например, но съответните методи са налични само в **RandomAccessFile**, който работи само с файлове. **BufferedInputStream** не позволява да `mark()` мястото (чиято стойност се пази във вътрешна променлива) и `reset()` към това място, но това е ограничено и не много полезно.

Класът **File**

Класът **File** има измамно име – може да помислите, че е за файлове, но не е. Той може да представи или име на конкретен файл или имена на множество от файлове в директория. Ако това е множество от файлове, може да питате за него с метода `list()`, а той връща масив от **String**. Има смисъл да се връща масив вместо някоя колекция, защото броят на елементите е фиксиран, а ако искате друга директория просто създавате друг **File** обект. Фактически "FilePath" щеше да бъде по-добро име. Тази секция привежда пример с пълно използване на класа, включително асоциирания **FilenameFilter interface**.

A directory lister

Да предположим, че трябва списък на файловете в директория. Обектът **File** може да го направи по два начина. Ако извикате `list()` без аргументи, ще получите пълен списък на имената съдържани във **File** обекта. Обаче ако искате списък с ограничения, например на всички файлове с разширение **.java**, използвате "директорен филтър," което е клас, казващ как да се изберат за изобразяване имената във **File** обектите.

Ето сорса на примера: (Виж стр. 63 при проблеми с пускането на програмата.)

```
//: c10:DirList.java
// Листинг на директорията
package c10;
import java.io.*;

public class DirList {
    public static void main(String[] args) {
        try {
            File path = new File(".");
            String[] list;
            if(args.length == 0)
                list = path.list();
            else
                list = path.list(new DirFilter(args[0]));
            for(int i = 0; i < list.length; i++)
                System.out.println(list[i]);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}

class DirFilter implements FilenameFilter {
    String afn;
    DirFilter(String afn) { this.afn = afn; }
```

```

public boolean accept(File dir, String name) {
    // Strip path information:
    String f = new File(name).getName();
    return f.indexOf(afn) != -1;
}
} //:~

```

Класът **DirFilter** „реализира“ **interface FilenameFilter**. (Интерфейсите бяха разгледани в глава 7.) Полезно е да се види колко прост е **FilenameFilter interface**:

```

public interface FilenameFilter {
    boolean accept(File dir, String name);
}

```

Той казва, че всичко което прави този клас е да даде метода **accept()**. Цялата причина за създаването на този клас е да се даде методът **accept()** на **list()** метода така че **list()** да може да извика обратно **accept()** за да определи кои файлови имена ще се включват в списъка. Тази техника често се нарича *callback* или понякога *functor* (тоест, **DirFilter** е функтор понеже единствената му задача е да съдържа метод). Понеже **list()** взима **FilenameFilter** обект за аргумент, това значи че може да дадете произволен обект от тип **FilenameFilter** за избор (даже по време на изпълнение) поведението на метода **list()**. Предназначението на обратното извикване е да се даде гъвкавост откъм поведението на коде.

DirFilter показва че тъй като **interface** съдържа само множество от методи, не сте ограничени да работите само с тях. (Трябва най-малкото да дадете дефиниции за всичките методи на интерфейса, обаче.) В този случай се създава също и конструктор на **DirFilter**.

Методът **accept()** трябва да приеме **File** обект представляящ къде даден файл е намерен, а **String** съдържа името на този файл. Бихте могли да изберете дали да използвате всеки от параметрите, но най-вероятно ще използвате поне файловото име. Запомнете че метода **list()** вика **accept()** за всяко от файловите имена в списъка за да види дали то ще бъде включено – това се индицира чрез **boolean** резултата връщан от **accept()**.

За да се работи само с имената, без информацията за пътя, трябва само да се вземе **String** обект и да се създаде **File** обект от него, тогава да се извика **getName()** което изрязва цялата информация за пътя по независим от платформата начин). Тогава **accept()** използва **String** класовия метод **indexOf()** да види дали стрингът **afn** се появява някъде във файловото име. Ако **afn** е намерен в стринга, върнатата стойност е началният индекс в **afn**, а ако не се намери върнатата стойност е -1. Помнете че това е просто търсене в стринг и няма търсене по съпадения като в регулярни изрази с “wildcard” знаци като в “fo?.b?r*” което е много по-трудно за реализиране.

Методът **list()** връща масив. Може да проверите дължината му и после да се движите по него. Тази възможност непосредствено да се дават масиви като аргумент и да се връщат е огромно придвижване напред в сравнение с С и С++.

АНОНИМНИ ВЪТРЕШНИ КЛАСОВЕ

Този пример е идеален за пренаписване с анонимни вътрешни класове (описани в глава 7). Като начало се създава методът **filter()** който връща манипулятор към **FilenameFilter**:

```

//: c10:DirList2.java
// Uses Java 1.1 anonymous inner classes
import java.io.*;

public class DirList2 {
    public static FilenameFilter
        filter(final String afn) {

```

```

// Creation of anonymous inner class:
return new FilenameFilter() {
    String fn = afn;
    public boolean accept(File dir, String n) {
        // Strip path information:
        String f = new File(n).getName();
        return f.indexOf(fn) != -1;
    }
}; // End of anonymous inner class
}
public static void main(String[] args) {
    try {
        File path = new File(".");
        String[] list;
        if(args.length == 0)
            list = path.list();
        else
            list = path.list(filter(args[0]));
        for(int i = 0; i < list.length; i++)
            System.out.println(list[i]);
    } catch(Exception e) {
        e.printStackTrace();
    }
}
} ///:~

```

Забележете че аргументът към **filter()** трябва да бъде **final**. Това се изисква от вътрешния клас така че той да може да използва променливите на обекта извън обхвата си.

Този дизайн е усъвършенстван, понеже класът **FilenameFilter** сега е тясно свързан с **DirList2**. Обаче може да се придвижите стъпка напред с този подход и да дефинирате анонимен вътрешен клас за аргумент на **list()**, в който случай той е даже по-малък:

```

//: c10:DirList3.java
// Building the anonymous inner class "in-place"
import java.io.*;

public class DirList3 {
    public static void main(final String[] args) {
        try {
            File path = new File(".");
            String[] list;
            if(args.length == 0)
                list = path.list();
            else
                list = path.list(
                    new FilenameFilter() {
                        public boolean
                        accept(File dir, String n) {
                            String f = new File(n).getName();
                            return f.indexOf(args[0]) != -1;
                        }
                    });
            for(int i = 0; i < list.length; i++)
                System.out.println(list[i]);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}

```

```
    }
}
} ///:~
```

Аргументът на **main()** е сега **final**, понеже анонимният вътрешен клас използва директно **args(0)**.

Това показва как анонимните вътрешни класове позволяват да се създадат бързо и лесно класове. Тъй като всичко в Java се върти около класовете, това би могло да бъде полезна техника. Една полза е че кодът, който решава даден проблем остава тясно свързан и в една точка. От друга страна, не винаги е лесен за четене, така че трябва да се използва разумно.

Сортиран списък на директорията

А-а, искате имената сортирани? Тъй като няма поддръжка за сортиране в Java 1.0 и Java 1.1 (макар и сортирането да е включено в Java 2), ще трябва да го включим директно в програмата използвайки **SortList** създаден в глава 8:

```
//: c10:SortedDirList.java
// Displays sorted directory listing
import java.io.*;
import c08.*;

public class SortedDirList {
    private File path;
    private String[] list;
    public SortedDirList(final String afn) {
        path = new File(".");
        if(afn == null)
            list = path.list();
        else
            list = path.list(
                new FilenameFilter() {
                    public boolean
                    accept(File dir, String n) {
                        String f = new File(n).getName();
                        return f.indexOf(afn) != -1;
                    }
                });
        sort();
    }
    void print() {
        for(int i = 0; i < list.length; i++)
            System.out.println(list(i));
    }
    private void sort() {
        StrSortList sv = new StrSortList();
        for(int i = 0; i < list.length; i++)
            sv.add(list(i));
        // The first time an element is pulled from
        // the StrSortList the list is sorted:
        for(int i = 0; i < list.length; i++)
            list(i) = sv.get(i);
    }
    // Test it:
    public static void main(String[] args) {
        SortedDirList sd;
        if(args.length == 0)
```

```

sd = new SortedDirList(null);
else
    sd = new SortedDirList(args(0));
    sd.print();
}
} //:~

```

Няколко други подобрения бяха направени. Вместо да се създават **path** и **list** като локални променливи на **main()**, те са членове на клас, така че техните стойности са достъпни по времето на живот на класа. Фактически **main()** сега е просто начин да се тества класа. Може да видите, че конструктора на класа автоматично сортира имената щом списъка се създа.

Сортирането е независимо от големи или малки букви, така че не получавате сортиран списък на думите с главни букви следван от тези с малки. Може обаче да забележите, че когато има две имена с еднакви бекви първо се изобразява това с голяма, което все още не е точно желаното повезение за сортирането. Този проблем ще се реши в Java 2.

Проверка за и създаване на директории

Класът **File** е повече от просто представяне на път, файл или група файлове. Също може да използвате **File** обект за създаване на нова директория или цялостен път, който не съществува. Може също да гледате характеристиките на файловете (дължина, дата на последната промяна, четене/писане), дали **File** обекта представя файл или директория и да изтриете файл. Тази програма показва останалите методи налични в класа **File**:

```

//: c10:MakeDirectories.java
// Demonstrates the use of the File class to
// create directories and manipulate files.
import java.io.*;

public class MakeDirectories {
    private final static String usage =
        "Usage:MakeDirectories path1 ...\\n" +
        "Creates each path\\n" +
        "Usage:MakeDirectories -d path1 ...\\n" +
        "Deletes each path\\n" +
        "Usage:MakeDirectories -r path1 path2\\n" +
        "Renames from path1 to path2\\n";
    private static void usage() {
        System.err.println(usage);
        System.exit(1);
    }
    private static void fileData(File f) {
        System.out.println(
            "Absolute path: " + f.getAbsolutePath() +
            "\\n Can read: " + f.canRead() +
            "\\n Can write: " + f.canWrite() +
            "\\n getName: " + f.getName() +
            "\\n getParent: " + f.getParent() +
            "\\n getPath: " + f.getPath() +
            "\\n length: " + f.length() +
            "\\n lastModified: " + f.lastModified());
        if(f.isFile())
            System.out.println("it's a file");
        else if(f.isDirectory())
            System.out.println("it's a directory");
    }
}

```

```

}
public static void main(String[] args) {
    if(args.length < 1) usage();
    if(args[0].equals("-r")) {
        if(args.length != 3) usage();
        File
            old = new File(args[1]),
            rname = new File(args[2]);
        old.renameTo(rname);
        fileData(old);
        fileData(rname);
        return; // Exit main
    }
    int count = 0;
    boolean del = false;
    if(args[0].equals("-d")) {
        count++;
        del = true;
    }
    for( ; count < args.length; count++) {
        File f = new File(args[count]);
        if(f.exists()) {
            System.out.println(f + " exists");
            if(del) {
                System.out.println("deleting..." + f);
                f.delete();
            }
        }
        else { // Doesn't exist
            if(!del) {
                f.mkdirs();
                System.out.println("created " + f);
            }
        }
        fileData(f);
    }
}
} ///:~

```

Във **fileData()** може да видите различни методи за файловете, използвани по подходящ начин.

Първият метод изпитан в **main()** е **renameTo()**, който позволява да се преименува (или премести) файл към напълно нов път, представен от аргумента, който е друг **File** обект. Това също работи за директории с всякааква дължина.

Ако експериментирате с горната програма, може да видите че е възможно да се направи път с всякааква сложност, понеже **mkdirs()** ще направи нужното. В Java 1.0, флагът **-d** показва, че директорията е изтрита, но все още е там (т.е. е белязана за изтриване - б.пр.); в Java 1.1 директорията е изтрита фактически.

ТИПИЧНО ИЗПОЛЗВАНЕ НА IO ПОТОЦИ

Макар и да има множество IO потокови класове в библиотеката, които могат да бъдат комбинирани по много начини, има само няколко, които ще се случи вероятно да използвате.

Те изискват внимание при подбора на коректните комбинации, обаче. Следващия малко дълъг пример показва използването на типични IO конфигурации така че може да го използвате за справки като пишете собствен код. Забележете че всяка комбинация започва с изкоментиран номер и заглавие, което кореспондира на това от обясненията, които следват текста.

```
//: c10:IOStreamDemo.java
// Typical IO Stream Configurations
import java.io.*;
import com.bruceeeckel.tools.*;

public class IOStreamDemo {
    public static void main(String[] args) {
        try {
            // 1. Buffered input file
            DataInputStream in =
                new DataInputStream(
                    new BufferedInputStream(
                        new FileInputStream(args[0])));
            String s, s2 = new String();
            while((s = in.readLine())!= null)
                s2 += s + "\n";
            in.close();

            // 2. Input from memory
            StringBufferInputStream in2 =
                new StringBufferInputStream(s2);
            int c;
            while((c = in2.read()) != -1)
                System.out.print((char)c);

            // 3. Formatted memory input
            try {
                DataInputStream in3 =
                    new DataInputStream(
                        new StringBufferInputStream(s2));
                while(true)
                    System.out.print((char)in3.readByte());
            } catch(EOFException e) {
                System.out.println(
                    "End of stream encountered");
            }

            // 4. Line numbering & file output
            try {
                LineNumberInputStream li =
                    new LineNumberInputStream(
                        new StringBufferInputStream(s2));
                DataInputStream in4 =
                    new DataInputStream(li);
                PrintStream out1 =
                    new PrintStream(
                        new BufferedOutputStream(
                            new FileOutputStream(
                                "IDemo.out")));
                while((s = in4.readLine()) != null )
                    out1.println(

```

```

    "Line " + li.getLineNumber() + s);
out1.close(); // finalize() not reliable!
} catch(EOFException e) {
System.out.println(
    "End of stream encountered");
}

// 5. Storing & recovering data
try {
DataOutputStream out2 =
    new DataOutputStream(
        new BufferedOutputStream(
            new FileOutputStream("Data.txt")));
out2.writeBytes(
    "Here's the value of pi: \n");
out2.writeDouble(3.14159);
out2.close();
DataInputStream in5 =
    new DataInputStream(
        new BufferedInputStream(
            new FileInputStream("Data.txt")));
System.out.println(in5.readLine());
System.out.println(in5.readDouble());
} catch(EOFException e) {
System.out.println(
    "End of stream encountered");
}

// 6. Reading/writing random access files
RandomAccessFile rf =
    new RandomAccessFile("rtest.dat", "rw");
for(int i = 0; i < 10; i++)
    rf.writeDouble(i*1.414);
rf.close();

rf =
    new RandomAccessFile("rtest.dat", "rw");
rf.seek(5*8);
rf.writeDouble(47.0001);
rf.close();

rf =
    new RandomAccessFile("rtest.dat", "r");
for(int i = 0; i < 10; i++)
    System.out.println(
        "Value " + i + ": " +
        rf.readDouble());
rf.close();

// 7. File input shorthand
InFile in6 = new InFile(args[0]);
String s3 = new String();
System.out.println(
    "First line in file: " +
    in6.readLine());
in6.close();

```

```

// 8. Formatted file output shorthand
PrintFile out3 = new PrintFile("Data2.txt");
out3.print("Test of PrintFile");
out3.close();

// 9. Data file output shorthand
OutFile out4 = new OutFile("Data3.txt");
out4.writeBytes("Test of outDataFile\n\r");
out4.writeChars("Test of outDataFile\n\r");
out4.close();

} catch(FileNotFoundException e) {
    System.out.println(
        "File Not Found:" + args(0));
} catch(IOException e) {
    System.out.println("IO Exception");
}
}

} // :~
```

ВХОДНИ ПОТОЦИ

Разбира се, едно нещо което много се използва е извеждането на форматирани данни на конзолата, но това беше вече разгледано в пакета **com.bruceekel.tools** създаден в глава 5.

Части 1 до 4 демонстрират създаването и използването на входни потоци (макар и част 4 също да показва и прост начин на използване на изходен поток като инструмент за тестване).

1. Буфериран вход от файл

За да отворите файл за вход използвате **FileInputStream** със **String** или **File** обект като име на файл. За скорост бихте желали входът да бъде буфериран така че давате получения манипулятор на **BufferedInputStream**. За да се чете входът форматиран, давате получения манипулятор на конструктора на **DataInputStream**, който е вашият краен обект и от чийто интерфейс четете.

В този пример само **readLine()** метода се използва, но разбира се налице са всичките методи на **DataInputStream**. Като достигнете края на файла, **readLine()** връща **null** и то се използва така, че се прекъсва **while** цикъла.

String-ът **s2** се използва за събиране на цялото съдържание на файла (включително знаците за нов ред които трябва да се добавят, понеже **readLine()** ги маха). **s2** после се използва в други части на програмата. Накрая се вика **close()** за да затвори файла. Технически, **close()** ще се извика когато се пусне **finalize()** и това се предполага да се случи (независимо дали ще има събиране на боклука) при завършването на програмата. Обаче Java 1.0 има един важен бъг, така че това няма да се случи. В Java 1.1 трябва явно да се извика **System.runFinalizersOnExit(true)** за да се гарантира, че **finalize()** ще се извика за всеки обект в системата. Най-безопасният подход е явно да се вика **close()** за файлове.

2. Вход от паметта

Това парче взема **String s2** който сега съдържа целия файл и го използва за създаване на **StringBufferInputStream**. (**String**, не **StringBuffer**, се изисква за аргумент на конструктора.) Тогава **read()** се използва за четене на знаците един по един и изпращането им на конзолата. Забележете че **read()** връща следващия байт като **int** и трябва да се превърне в **char** за правилно извеждане.

3. Форматиран вход от паметта

Интерфейсът на **StringBufferInputStream** е ограничен, така че обикновено го разширяваме чрез обгръщане в **DataInputStream**. Обаче ако сте избрали да прочитате байт по байт чрез **readByte()**, всяка стойност е валидна и връщаната стойност не може да се използва за определяне края на въвеждането. Вместо това може да се използва метода **available()** за намиране колко знака остават. Ето пример който показва как да се чете файл байт по байт:

```
//: c10:TestEOF.java
// Testing for the end of file while reading
// a byte at a time.
import java.io.*;

public class TestEOF {
    public static void main(String[] args) {
        try {
            DataInputStream in =
                new DataInputStream(
                    new BufferedInputStream(
                        new FileInputStream("TestEOF.java")));
            while(in.available() != 0)
                System.out.print((char)in.readByte());
        } catch (IOException e) {
            System.err.println("IOException");
        }
    }
} ///:~
```

Забележете че **available()** работи различно според това какъв е видът на медията с която се работи – буквально “броят байтове който може да се прочете не блоково.” С файл това значи целия файл, но с други видове устройства това може да не е така, така че го използвайте внимателно.

Освен това бихте могли да хванете края на файла в подобни случаи чрез изключение. Обаче използването на изключения за управление хода на програмата се счита за неправилно тяхно използване.

4. Номериране на редове и файлов изход

Този пример показва използването на **LineNumberInputStream** за да се следят номерата на редовете. Тук не може просто да се съберат всичките конструктори заедно, понеже трябва да се пази манипулатор към **LineNumberInputStream**. (Забележете че това не е ситуация на наследяване, така че не може да се направи просто кастинг на **in4** към **LineNumberInputStream**.) Така, **li** съдържа манипулатор към **LineNumberInputStream**, който после се използва за създаване на **DataInputStream** за лесно четене.

Този пример също показва писането на форматирани данни във файл. Първо се създава **FileOutputStream** за свързване с файла. За ефективност е направен **BufferedOutputStream**, което почти винаги ще е желания случай, но се налага да го правите явно. После за форматирането той е превърнат в **PrintStream**. Данният файл създаден по този начин е четим като обикновен текстов файл.

Един от методите който показва че **DataInputStream** е изтощен е **readLine()**, който връща **null** когато повече няма стрингове за четене. Всеки ред се записва във файла заедно с номера си, който се следи чрез **li**.

Ще видите явен **close()** за **out1**, което би имало смисъл ако програмата щеше да се върне и да чете файла отново. Тази програма обаче завършва без да се обръща към файла

IODemo.out. Както се спомена преди, ако не извикате **close()** за всички ваши изходни файлове, бихте могли да откриете че буферът не е записан и файлът не е завършен.

Изходни потоци

Двета основни вида изходни потоци се различават по начина на записване на данните: при единия са предназначени за хората, а другият ги пише за да може да бъдат възстановени от **DataInputStream**. **RandomAccessFile** е отделно, макар и форматът на данните при него да е съвместим с **DataInputStream** и **DataOutputStream**.

5. Запазване и възстановяване на данни

PrintStream форматира данните така, че са разбираеми за човек. За извеждане на данни които да могат да бъдат възстановени (прочетени) от друг поток използваме **DataOutputStream** за запис на данните и **DataInputStream** за възстановяването им. Разбира се, тези потоци биха могли да бъдат всякакви, но тук се използва файл.

Забележете че знаковият низ е написан чрез **writeBytes()** а не чрез **writeChars()**. Ако използвате последния бихте писали 16-битовите Unicode знаци. Тъй като няма съответен "readChars" метод в **DataInputStream**, ограничени сте да вземате знаците един по един чрез **readChar()**. Така за ASCII, по-лесно е да пишете знаците като байтове следвани от знак за нов ред; после чрез **readLine()** да четете байтовете обратно като обикновен ASCII ред.

writeDouble() запомня **double** число в потока и съответно **readDouble()** го възстановява. Но за всеки от методите за четене, за да работи коректно, трябва точно да знаете разположението на данната в потока, тъй като е напълно възможно да четете **double** и като пристап последователност от байтове, или **char** и т.н.. Така че трябва или да има фиксиран формат на файловете или в тях да се помни допълнително нужната информация за структурата им.

6. Четене и писане на файлове с произволен достъп

Както бе отбелязано преди, **RandomAccessFile** е почти напълно изолиран от останалата част на IO йерархията, освен че прилага интерфейсите на **DataInput** и **DataOutput**. Така че не може да го комбинирате в никакъв аспект на **InputStream** и **OutputStream** подкласовете. Макар и да има смисъл да се третира **ByteArrayInputStream** като елемент с произволен достъп, може да използвате **RandomAccessFile** само да отвори файл. Трябва да предполагате че **RandomAccessFile** е правилно буфериран, понеже не може да добавите тази черта.

Едничкият избор който имате е аргументът на вторият конструктор: може да отворите **RandomAccessFile** за четене ("r") или четене и писане ("rw").

Използването на **RandomAccessFile** е подобно на използването на комбинирани **DataInputStream** и **DataOutputStream** (понеже прилага еквивалентни интерфейси). Освен това може да видите, че **seek()** се използва за разходки по файла и промяна на една от стойностите.

Кратък начин за операции с файлове

Тъй като има няколко типични форми, които се използват често с файлове, бихте могли да се чудите защо трябва да се пише толкова много код (всеки път-б.пр.) – това е един от недостатъците на декораторския подход. Тази част показва създаването и използването на съкратени процедури за четене и писане от файлове. Тези неща са сложени в пакета **package com.bruceeckel.tools** който беше започнат в глава 5 (Виж стр. 136). За да се добави всеки клас в библиотеката, просто го сложете в съответната директория и добавете оператор **package**.

7. Кратък начин за вход от файл

Създаването на обект чете файл от буфериран **DataInputStream** може да бъде капсулирано в клас наречен **InFile**:

```
//: com:bruceeckel:tools:InFile.java
// Shorthand class for opening an input file
package com.bruceeckel.tools;
import java.io.*;

public class InFile extends DataInputStream {
    public InFile(String filename)
        throws FileNotFoundException {
        super(
            new BufferedInputStream(
                new FileInputStream(filename)));
    }
    public InFile(File file)
        throws FileNotFoundException {
        this(file.getPath());
    }
} ///:~
```

И двете версии на **String** на конструктора и **File** версията са включени, с цел да се прокара паралел със създаването на **FileInputStream**.

Сега може да намалите вероятността от повтарящ се стрес, както се вижда в примера.

8. Кратък път за форматирано писане във файл

Същият подход може да се приеме за създаване на **PrintStream** който пише в буфериран файл. Ето разширението за **com.bruceeckel.tools**:

```
//: com:bruceeckel:tools:PrintFile.java
// Shorthand class for opening an output file
// for human-readable output.
package com.bruceeckel.tools;
import java.io.*;

public class PrintFile extends PrintStream {
    public PrintFile(String filename)
        throws IOException {
        super(
            new BufferedOutputStream(
                new FileOutputStream(filename)));
    }
    public PrintFile(File file)
        throws IOException {
        this(file.getPath());
    }
} ///:~
```

Забележете че не е възможно конструктор да хване изключение, изхвърлено то конструктора на базовия клас.

9. Начин за писане в даннов файл

Накрая, същия вид кратък път може да се използва за запазване на данни (като противоположност на читаемите от човек):

```
//: com:bruceeckel:tools:OutFile.java
// Shorthand class for opening an output file
// for data storage.
package com.bruceeckel.tools;
import java.io.*;

public class OutFile extends DataOutputStream {
    public OutFile(String filename)
        throws IOException {
        super(
            new BufferedOutputStream(
                new FileOutputStream(filename)));
    }
    public OutFile(File file)
        throws IOException {
        this(file.getPath());
    }
} ///:~
```

Любопитно е (и е за нещастие) че проектантите на Java библиотеката не счетоха за необходимо да добавят тези удобства в техния стандарт.

Четене от стандартния вход

Следвайки подхода започнат от Unix за "стандартен вход," "стандартен изход," и "стандартен изход за грешки," Java има **System.in**, **System.out** и **System.err**. По протежение на книгата сте виждали как се извежда на **System.out**, което е вече обгърнато от **PrintStream** обект. **System.err** е подобен на **PrintStream**, но **System.in** е "суров" **InputStream**, без обгръщането. Това значи, че докато можете да използвате **System.out** и **System.err**, **System.in** трябва да бъде обгърнат (с друг обект - б.пр.) преди да го използвате.

Типично ще искате да се чете по ред на един път с **readLine()**, така че ще трябва да обгърнете **System.in** с **DataInputStream**. Този е "старие" начин на Java 1.0 за вход ред по ред. Малко по късно в главата ще видите решението в Java 1.1. Ето пример, който прави "ехо" на всеки въведен ред:

```
//: c10:Echo.java
// How to read from standard input
import java.io.*;

public class Echo {
    public static void main(String[] args) {
        DataInputStream in =
            new DataInputStream(
                new BufferedInputStream(System.in));
        String s;
        try {
            while((s = in.readLine()).length() != 0)
                System.out.println(s);
            // An empty line terminates the program
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
    }
}
} //:/~
```

Причината да има **try** блок е че **readLine()** може да изхвърли **IOException**. Забележете че **System.in** трябва също да бъде буфериран, както с повечето потоци.

Не е много удобно че сте принудени да обгръщате **System.in** в **DataInputStream** във всяка програма, но може би е избран този начин заради максимална гъвкавост.

Скачени потоци

PipedInputStream и **PipedOutputStream** само накъсо се споменаха в тази глава. Това не е защото не са полезни, а защото ползата от тях не е явна докато не започнете да разбирате многонишковостта, понеже свързаните потоци се използват за връзка между нишките. Това е изяснено чрез пример в глава 14.

StreamTokenizer

Макар и **StreamTokenizer** да не е извлечен от **InputStream** или **OutputStream**, той работи само с **InputStream** обекти, така че направо си принадлежи към IO частта на библиотеката.

Класът **StreamTokenizer** се използва за разбиването на **InputStream** последователност от "tokens," които са парчета текст разделени с нещо по ваш избор. Например вашите токен могат да бъдат думи, а те могат да бъдат разделени с интервали и пунктуация.

Да видим програма за броенето на появата на думи в текст:

```
//: c10:SortedWordCount.java
// Counts words in a file, outputs
// results in sorted form.
import java.io.*;
import java.util.*;
import c08.*; // Contains StrSortList

class Counter {
    private int i = 1;
    int read() { return i; }
    void increment() { i++; }
}

public class SortedWordCount {
    private FileReader file;
    private StreamTokenizer st;
    private HashMap counts = new HashMap();
    SortedWordCount(String filename)
        throws FileNotFoundException {
        try {
            file = new FileReader(filename);
            st = new StreamTokenizer(new BufferedReader(file));
            st.ordinaryChar('.');
            st.ordinaryChar('-');
        } catch(FileNotFoundException e) {
            System.out.println(
                "Could not open " + filename);
            throw e;
        }
    }

    void count() {
        while(st.nextToken() != StreamTokenizer.TT_EOF) {
            String token = st.sval;
            if(token.length() > 0) {
                if(counts.containsKey(token))
                    counts.put(token, counts.get(token) + 1);
                else
                    counts.put(token, 1);
            }
        }
    }

    void print() {
        Iterator iter = counts.entrySet().iterator();
        while(iter.hasNext()) {
            Map.Entry entry = (Map.Entry)iter.next();
            System.out.println(entry.getKey() + " " +
                entry.getValue());
        }
    }
}
```

```

    }
}

void cleanup() {
    try {
        file.close();
    } catch(IOException e) {
        System.out.println(
            "file.close() unsuccessful");
    }
}

void countWords() {
    try {
        while(st.nextToken() != StreamTokenizer.TT_EOF) {
            String s;
            switch(st.ttype) {
                case StreamTokenizer.TT_EOL:
                    s = new String("EOL");
                    break;
                case StreamTokenizer.TT_NUMBER:
                    s = Double.toString(st.nval);
                    break;
                case StreamTokenizer.TT_WORD:
                    s = st.sval; // Already a String
                    break;
                default: // single character in ttype
                    s = String.valueOf((char)st.ttype);
            }
            if(counts.containsKey(s))
                ((Counter)counts.get(s)).increment();
            else
                counts.put(s, new Counter());
        }
    } catch(IOException e) {
        System.out.println(
            "st.nextToken() unsuccessful");
    }
}

Collection values() {
    return counts.values();
}

Set keySet() { return counts.keySet(); }

Counter getCounter(String s) {
    return (Counter)counts.get(s);
}

Iterator sortedKeys() {
    Iterator e = counts.keySet().iterator();
    StrSortList sv = new StrSortList();
    while(e.hasNext())
        sv.add((String)e.next());
    // This call forces a sort:
    return sv.iterator();
}

public static void main(String[] args) {
    try {
        SortedWordCount wc =
            new SortedWordCount(args[0]);

```

```

wc.countWords();
Iterator keys = wc.sortedKeys();
while(keys.hasNext()) {
    String key = (String)keys.next();
    System.out.println(key + ": "
        + wc.getCounter(key).read());
}
wc.cleanup();
} catch(Exception e) {
    e.printStackTrace();
}
}
}

} //:~

```

Има смисъл да бъдат сортирани, но доколкото Java 1.0 и Java 1.1 нямат никакви сортиrovки, това ще трябва да се направи. Лесно става с **StrSortList**. (Беше създаден в глава 8 и е част от пакета създаден в нея глава. Помнете че началната директория на поддиректориите на примерите от тази книга трябва да е на вашия "път до класовете" за да може те да се компилират.)

За отваряне на файла се използва **FileInputStream**, а за превръщането му в токени се създава **StreamTokenizer** от **FileInputStream**. В **StreamTokenizer** има списък от разделители по подразбиране, а вие може да добавите още чрез методи. Тука е използван **ordinaryChar()** за да каже "Този знак не е интересен за мен," така че парсерът не го включва изобщо в думите. Например казвайки **st.ordinaryChar('.')** ще получим точките да не се включват в състава на думите които се разделят. Повече информация може да намерите в онлайн документацията която идва с Java.

В **countWords()** токените се изтеглят един по един от потока, а **type** информацията се използва за да се реши какво да се прави с всеки, тъй като токенът може да бъде край на файл, число, стринг или единствен знак.

Щом се намери токенът, пита се **HashMap counts** дали вече го съдържа като ключ. Ако да, съответният **Counter** обект се инкрементира за да покаже, че е намерен още един случай на тази дума. Ако не, нов **Counter** се създава – и доколкото конструкторът на **Counter** се инициализира с единица, това отразява и първата поява.

SortedWordCount не е тип **HashMap**, така че не е наследен. Той изполнява специфична функция, та макар и **keys()** и **values()** методите трябва да ги има, това още не значи че ще се използва наследяване понеже много от **HashMap** методите не стават тук. Освен това други методи като **getCounter()**, които вземат **Counter** за конкретен **String** и **sortedKeys()**, който дава **Iterator**, завършват промяната на интерфейса на **SortedWordCount**.

В **main()** може да се види използването на **SortedWordCount** за отваряне и броене на думи във файл – това са точно два реда код. После се извлича **Iterator** към сортирания списък на ключове (думите) и се използва за измъкване на съответния ключ и броя на срещанията му **Count**. Забележете че викането на **cleanup()** е необходимо за да се осигури затварянето на файла.

Втори пример с използване на **StreamTokenizer** има в глава 17.

StringTokenizer

Макар и да не е част от IO библиотеката, **StringTokenizer** има достатъчно подобна на **StreamTokenizer** функционалност така че ще се опише тук.

StringTokenizer връща токените в стринг един по един. Тези токени са последователни знаци разделиeni с табулации, интервали и знаци за нов ред. Така токените в стринга "Where is my

cat?" са "Where", "is", "my" и "cat?" Като при **StreamTokenizer**, вие може да поръчате на **StringTokenizer** да разделя входния стринг по всякакъв желан начин, но при **StringTokenizer** това се прави чрез даване на втори аргумент на конструктора, който е **String** от желаните разделители. Изобщо, ако ви трябва по-усъвършенстван начин, използвайте **StreamTokenizer**.

Следващия токен се иска от **StringTokenizer** обекта чрез метода **nextToken()**, който връща или токена, или празен стринг за да покаже че няма повече токени.

Като пример следната програма изпълнява ограничен анализ на изречение, оглеждайки за ключови фрази за да разбере дали се предполага радост или тъга.

```
//: c10:AnalyzeSentence.java
// Look for particular sequences
// within sentences.
import java.util.*;

public class AnalyzeSentence {
    public static void main(String[] args) {
        analyze("I am happy about this");
        analyze("I am not happy about this");
        analyze("I am not! I am happy");
        analyze("I am sad about this");
        analyze("I am not sad about this");
        analyze("I am not! I am sad");
        analyze("Are you happy about this?");
        analyze("Are you sad about this?");
        analyze("It's you! I am happy");
        analyze("It's you! I am sad");
    }

    static StringTokenizer st;
    static void analyze(String s) {
        System.out.println("\nnew sentence >> " + s);
        boolean sad = false;
        st = new StringTokenizer(s);
        while (st.hasMoreTokens()) {
            String token = st.nextToken();
            // Look until you find one of the
            // two starting tokens:
            if(!token.equals("I") &&
               !token.equals("Are"))
                continue; // Top of while loop
            if(token.equals("I")) {
                String tk2 = st.nextToken();
                if(!tk2.equals("am")) // Must be after I
                    break; // Out of while loop
                else {
                    String tk3 = st.nextToken();
                    if(tk3.equals("sad")) {
                        sad = true;
                        break; // Out of while loop
                    }
                    if (tk3.equals("not")) {
                        String tk4 = st.nextToken();
                        if(tk4.equals("sad"))
                            break; // Leave sad false
                        if(tk4.equals("happy")) {
                            sad = true;
                        }
                    }
                }
            }
        }
    }
}
```

```

        break;
    }
}
}

if(token.equals("Are")) {
    String tk2 = next();
    if(!tk2.equals("you"))
        break; // Must be after Are
    String tk3 = next();
    if(tk3.equals("sad"))
        sad = true;
    break; // Out of while loop
}
}

if(sad) prt("Sad detected");
}

static String next() {
    if(st.hasMoreTokens()) {
        String s = st.nextToken();
        prt(s);
        return s;
    }
    else
        return "";
}

static void prt(String s) {
    System.out.println(s);
}

} //:~

```

За да бъде анализиран всеки стринг се влиза в **while** цикъл и токените се изваждат от стринга. Забележете първия **if** оператор, който казва да **continue** (да отиде в началото на цикъла и да влезе в него пак) ако токенът на е нито "I" нито "Are." Това значи че ще взимате токени досато биват намирани "I" или "Are". Бихте могли да искате да използвате **==** вместо **equals()** метода, но това не би работило коректно, понеже **==** сравнява стойности на манипулатори, докато **equals()** сравнява съдържанията.

Логиката на останалата част от метода **analyze()** е че се търси за "I am sad," "I am not happy" или "Are you sad?" Без **break** оператора кодът би бил даже по-объркан отколкото е. Така че да сте предупредени че типичен парсър (този е елементарен пример за такъв) нормално има таблица за тези токени и код който се движи по нея, като си променя състоянието в зависимост от появата на нови токени .

Ще считате **StringTokenizer** само за кратък път към един опростен **StreamTokenizer**. Обаче ако имате **String** който искате да разделяте и **StringTokenizer** е твърде ограничен, всичко което трябва да направите е да го превърнате в поток със **StringBufferInputStream** и после да създадете с него много по-мощния **StreamTokenizer**.

IO потоци на Java 1.1

В тази точка може да започнете да си блъскате главата, чудейки се дали има друг възможен дизайн на IO потоци който би изискал повече писане. Би ли могъл някой да направи по-неподходящ дизайн?" Пригответе се: Java 1.1 прави някои значителни промени в IO потоковата библиотека. Като видите **Reader** и **Writer** класовете първо мислите (както и аз) че може би те са предназначени да заменят класовете **InputStream** и **OutputStream**. Но не е така.

Макар и някои аспекти на старата библиотека да се смятат за остарели (ако ги използвате ще получите предупреждение от компилатора), старите потоци са оставени заради обратната съвместимост и понеже:

1. Нови класове са били сложени в старата йерархия, та очевидно Sun не изоставя старите потоци.
2. По някой път се предполага да се използват класовете от старата йерархия в комбинация с класовете от новата йерархия и за да се направи това има "мостови" класове: **InputStreamReader** преобразува **InputStream** в **Reader** и **OutputStreamWriter** преобразува **OutputStream** в **Writer**.

Като резултат има ситуации къде то са налице повече нива на обгръщане с новата IO потокова библиотека отколкото старата. Отново, това е недостатък на подхода с декораторите – цената, която се плаща заради повишената гъвкавост.

Най-важният мотив за добавяне на **Reader** и **Writer** йерархиите в Java 1.1 е интернационализацията. Старата IO потокова йерархия поддържа само 8-битови знакови потоци и не работи с 16-битови Unicode знаци добре. Понеже Unicode се използва за интернационализация (и естественият за Java **char** е 16-битов Unicode), **Reader** и **Writer** йерархиите бяха добавени за поддръжка на Unicode във всичките IO операции. Освен това новите библиотеки са проектирани по-бързи от старите.

Каквато е практиката в тази книга, аз ще се опитам да дам общ поглед върху класовете, като за пълните подробности и изчерпателен списък на методите ще използвате онлайн документацията.

Източници и стоци на данни

Почти всичките IO потокови класове на Java 1.0 имат съответни Java 1.1 класове за да осигурят естествена Unicode манипулация. Би било най-лесно да кажем "Използвайте винаги новите класове и никога старите," но нещата не са толкова прости. Понякога се налага да се използват Java 1.0 IO потоковите класове поради устройството на библиотеката; в частност, **java.util.zip** библиотеките са нови неща към старата библиотека и те разчитат на компоненти от старите потоци. Така че най-смисления подход е да се опитаме да използваме **Reader** и **Writer** където е възможно, а ще се случи и да се върнем към старите библиотеки понеже иначе кодът няма да се компилира.

Ето таблица която показва съответствието между източниците и стоците на информация (тоест, местата откъдето физически идва и където физически отива информацията) в старите и новите библиотеки.

Източници и стоци: Java 1.0 клас	Съответен Java 1.1 клас
InputStream	Reader преобразувател: InputStreamReader
OutputStream	Writer преобразувател: OutputStreamWriter
FileInputStream	FileReader
FileOutputStream	FileWriter
StringBufferInputStream (няма съответен клас)	StringReader StringWriter
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
PipedInputStream	PipedReader

Изобщо, ще откриете че интерфейсите в новите и старите библиотеки много си приличат, ако не са еднакви.

Променяне на поведението на потоците

В Java 1.0 потоците бяха приспособявани към конкретни нужди чрез използване на "декораторните" класове **FilterInputStream** и **FilterOutputStream**. С IO потоците в Java 1.1 тази идея продължава да се прилага, но не се следва извлечането на всички декоратори от единствен базов "фильтърен" клас. Това може да доведе до смущения ако се опитвате да разберете библиотеката гледайки йерархията на класовете.

В следващата таблица съответствието е по-грубо отколкото в предишната. Разликата се дължи на организацията на класовете: докато **BufferedOutputStream** е подклас на **FilterOutputStream**, **BufferedWriter** не е подклас на **FilterWriter** (който, макар че е **abstract**, няма подкласове и затова изглежда да е поставен за запазване на място или просто да не се чудите къде се е дянал). Обаче интерфейсите са достатъчно близки и е очевидно, че се очаква да използвате новите версии навсякъде, където е възможно (тоест освен в класовете, където се налага да се направи **Stream** вместо **Reader** или **Writer**).

Филтри: Java 1.0 клас	Съответен Java 1.1 клас
FilterInputStream	FilterReader
FilterOutputStream	FilterWriter (abstract клас без подкласове)
BufferedInputStream	BufferedReader (също има readLine())
BufferedOutputStream	BufferedWriter
DataInputStream	използвай DataInputStream (Освен когато искате да използвате readLine() , та ще трябва BufferedReader)
PrintStream	PrintWriter
LineNumberInputStream	LineNumberReader
StreamTokenizer	StreamTokenizer (използвайте конструктор който взема Reader)
PushBackInputStream	PushBackReader

Има една насока която е доста ясна: Навсякъде където искате да използвате **readLine()**, няма да го правите с **DataInputStream** повече (това се отбелязва със съобщение от компилатора), а с **BufferedReader**. Извън това **DataInputStream** е все още "предпочитания" член на Java 1.1 IO библиотеката.

За да се направи преходът **PrintWriter** по-лесен, той има конструктори които приемат **OutputStream** обект. Обаче **PrintWriter** повече не поддържа форматирането което прави **PrintStream**; интерфейсите са практически същите.

Непроменени класове

Види се проектантите на Java библиотеката са чувствали, че някои класове са направени както трябва от самото начало и те са оставени без промяна:

Java 1.0 без съответни Java 1.1 класове
DataOutputStream
File
RandomAccessFile
SequenceInputStream

DataOutputStream, в частност, се използва без промяна, токо че за запомняне и четене на данни в транспортируем формат сте принудени да стоите в **InputStream** и **OutputStream** йерархийте.

Пример

За да видим ефекта от новите класове, нека да погледнем подходяща част от **IostreamDemo.java** примера променен да използва класовете **Reader** и **Writer**:

```
//: c10:NewIODemo.java
// Java 1.1 IO typical usage
import java.io.*;

public class NewIODemo {
    public static void main(String[] args) {
        try {
            // 1. Reading input by lines:
            BufferedReader in =
                new BufferedReader(
                    new FileReader(args[0]));
            String s, s2 = new String();
            while((s = in.readLine())!= null)
                s2 += s + "\n";
            in.close();

            // 1b. Reading standard input:
            BufferedReader stdin =
                new BufferedReader(
                    new InputStreamReader(System.in));
            System.out.print("Enter a line:");
            System.out.println(stdin.readLine());

            // 2. Input from memory
            StringReader in2 = new StringReader(s2);
            int c;
            while((c = in2.read()) != -1)
                System.out.print((char)c);

            // 3. Formatted memory input
            try {
                DataInputStream in3 =
                    new DataInputStream(
                        // Oops: must use deprecated class:
                        new StringBufferInputStream(s2));
                while(true)
                    System.out.print((char)in3.readByte());
            } catch(EOFException e) {
                System.out.println("End of stream");
            }
        }
    }
}
```

```

// 4. Line numbering & file output
try {
    LineNumberReader li =
        new LineNumberReader(
            new StringReader(s2));
    BufferedReader in4 =
        new BufferedReader(li);
    PrintWriter out1 =
        new PrintWriter(
            new BufferedWriter(
                new FileWriter("IODemo.out")));
    while((s = in4.readLine()) != null )
        out1.println(
            "Line " + li.getLineNumber() + s);
    out1.close();
} catch(EOFException e) {
    System.out.println("End of stream");
}

// 5. Storing & recovering data
try {
    DataOutputStream out2 =
        new DataOutputStream(
            new BufferedOutputStream(
                new FileOutputStream("Data.txt")));
    out2.writeDouble(3.14159);
    out2.writeBytes("That was pi");
    out2.close();
    DataInputStream in5 =
        new DataInputStream(
            new BufferedInputStream(
                new FileInputStream("Data.txt")));
    BufferedReader in5br =
        new BufferedReader(
            new InputStreamReader(in5));
    // Must use DataInputStream for data:
    System.out.println(in5.readDouble());
    // Can now use the "proper" readLine():
    System.out.println(in5br.readLine());
} catch(EOFException e) {
    System.out.println("End of stream");
}

// 6. Reading and writing random access
// files is the same as before.
// (not repeated here)

} catch(FileNotFoundException e) {
    System.out.println(
        "File Not Found:" + args[1]);
} catch(IOException e) {
    System.out.println("IO Exception");
}
}

} ///:~

```

Изобщо ще видите, че преправянето е доста праволинейно и кодовете доста си приличат. Все пак има важни разлики, обаче. Преди всичко, понеже файловете с произволен достъп не са променени, секция 6 не е повторена.

Секция 1 се свива малко, понеже ако искате само да четете трябва само да обгърнете **FileReader** с **BufferedReader**. Секция 1b показва новия начин да се направи **System.in** за четене от конзолата, а той се разширява защото **System.in** е **DataInputStream** и **BufferedReader** иска **Reader** аргумент, та **InputStreamReader** се въвлича за изпълнение на транслацията.

В секция 2 може да се види че ако имате **String** и искате да четете от него просто използвате **StringReader** вместо **StringBufferInputStream** и останалото е същото.

Секция 3 показва бъг в новата IO потокова библиотека. Ако имате **String** и искате да четете от него, не се очаква да използвате **StringBufferInputStream** повече. Когато компилирате код с **StringBufferInputStream** конструктор получавате съобщение, което казва да не го използвате повече. Вместо това се очаква да използвате **StringReader**. Обаче ако искате да правите форматилен вход от паметта както в секция 3, налага се да използвате **DataInputStream** – няма “**DataReader**” да го замести – и **DataInputStream** конструкторът изисква **InputStream** аргумент. Така че нямате избор освен използването на остателяния **StringBufferInputStream** клас. Компилаторът предупреждава че използвате остатял клас, но няма какво друго да се направи.²

Секция 4 е достатъчно праволинейно преработена от старата библиотека за новата, без изненади. В секция 5 се налага да използвате всичките стари потокови класове понеже **DataOutputStream** и **DataInputStream** ги изискват и няма алтернативи. Не получавате никакво съобщение по време на компилация. Ако потокът е остатял (ще се изоставя – б.пр.) неговият конструктор обикновено предизвиква предупреждение по време на компилация, но в **DataInputStream** само методът **readLine()** е остатял, понеже се очаква да използвате **BufferedReader** за **readLine()** (но **DataInputStream** за всички останал форматирани вход).

Ако сравните секция 5 със съответната от **IOStreamDemo.java**, ще забележите че в нея версия данните са написани преди текста. Това е поради бъг допуснат в Java 1.1, който се показва със следващата програма:

```
//: c10:IOBug.java
// Java 1.1 (and higher?) IO Bug
import java.io.*;

public class IOBug {
    public static void main(String[] args)
        throws Exception {
        DataOutputStream out =
            new DataOutputStream(
                new BufferedOutputStream(
                    new FileOutputStream("Data.txt")));
        out.writeDouble(3.14159);
        out.writeBytes("That was the value of pi\n");
        out.writeBytes("This is pi/2:\n");
        out.writeDouble(3.14159/2);
        out.close();

        DataInputStream in =
            new DataInputStream(
                new BufferedInputStream(
                    new FileInputStream("Data.txt")));
        BufferedReader inbr =
```

² Perhaps by the time you read this, the bug will be fixed.

```

new BufferedReader(
    new InputStreamReader(in));
// The doubles written BEFORE the line of text
// read back correctly:
System.out.println(in.readDouble());
// Read the lines of text:
System.out.println(inbr.readLine());
System.out.println(inbr.readLine());
// Trying to read the doubles after the line
// produces an end-of-file exception:
System.out.println(in.readDouble());
}
} //:~

```

Видимо нищо след написано след извикването на **writeBytes()** не може да се извлече обратно. Това е доста ограничаваща грешка и може да се надяваме че ще бъде отстранена по времето, когато четете това. Ще пуснете горната програма за да я изprobвате; ако не получите изключение и стойностите се извеждат правилно, работата е наред.

Генератор на справки

```

//: c10:CrossReference.java
// Generates cross-reference listing of tokens
import java.util.*;
import java.io.*;

class CrossReference {
    // Comparator to ignore case:
    static class NoCase implements Comparator {
        public int compare(Object o1, Object o2) {
            String s1 = (String) o1;
            String s2 = (String) o2;
            return s1.compareToIgnoreCase(s2);
        }
    }
    static void process(BufferedReader r)
        throws IOException {
        TreeMap map = new TreeMap(new NoCase());
        String line;
        int lineno = 0;
        // Build map, reading a line at a time:
        while ((line = r.readLine()) != null) {
            ++lineno;
            // Read each token:
            String delim =
                " `~!@#$%^&*()_-_=+\\" | {}();:<,>/?`"
                + "0123456789";
            StringTokenizer tokens =
                new StringTokenizer(line, delim);
            while (tokens.hasMoreTokens()) {
                String token = tokens.nextToken();
                if (!map.containsKey(token)) {
                    // Add token and empty list to map:
                    map.put(token, new LinkedList());
                }
                // See if this line is in there already:
                LinkedList lines =

```

```

        (LinkedList) map.get(token);
        if (lines.isEmpty() || ((Integer)lines.getLast()).intValue() != lineno) {
            // Add line number to list:
            lines.addLast(new Integer(lineno));
        }
    }
}

// Output:
Iterator p = map.entrySet().iterator();
while (p.hasNext()) {
    Map.Entry e = (Map.Entry) p.next();
    System.out.print(e.getKey() + ": ");
    LinkedList lines = (LinkedList)e.getValue();
    for (int i = 0; i < lines.size(); ++i) {
        if (i > 0)
            System.out.print(", ");
        System.out.print(lines.get(i));
    }
    System.out.println();
}
}

public static void
main(String[] args) throws IOException {
    // Process each file:
    for (int i = 0; i < args.length; ++i) {
        System.out.println("File: " + args[i]);
        process(new BufferedReader(
            new FileReader(args[i])));
        System.out.println("=====");
    }
}
}
} //:~

```

Пренасочване на стандартния IO

Java 1.1 е добавил методи в класа **System**, които позволяват да се пренасочват стандартния входен, изходен и за грешки потоци чрез извикване на статични методи:

setIn(InputStream)
setOut(PrintStream)
setErr(PrintStream)

Пренасочването на изхода е особено полезно, когато много информация пробягва по вашия екран без да може да я проследите. Пренасочването на входа е ценно когато имате да пускате програма с повтарящ се потребителски вход при всяко пускане. Ето прост пример за използване на тези методи:

```

//: c10:Redirecting.java
// Demonstrates the use of redirection for
// standard IO in Java 1.1
import java.io.*;

class Redirecting {
    public static void main(String[] args) {
        try {

```

```

BufferedInputStream in =
    new BufferedInputStream(
        new FileInputStream(
            "Redirecting.java"));
// Produces deprecation message:
PrintStream out =
    new PrintStream(
        new BufferedOutputStream(
            new FileOutputStream("test.out")));
System.setIn(in);
System.setOut(out);
System.setErr(out);

BufferedReader br =
    new BufferedReader(
        new InputStreamReader(System.in)));
String s;
while((s = br.readLine()) != null)
    System.out.println(s);
out.close(); // Remember this!
} catch(IOException e) {
    e.printStackTrace();
}
}
} ///:~

```

Тази програма насочва стандартния вход да е от файл, а стандартния изход и стандартната грешка към друг файл.

Това е друг пример където предупреждението за остатялост е неизбежно. Съобщението което ще получите с помощта на флага **-deprecation** е:

*Note: The constructor java.io.PrintStream(java.io.OutputStream)
has been deprecated.*

Обаче и **System.setOut()** и **System.setErr()** изискват **PrintStream** обект за аргумент, така че се принуждавате да използвате **PrintStream** конструктор. Може да се чудите дали Java 1.1 смята за остатял целия **PrintStream** щом е остатял конструктора, докато проектантите на библиотеката, по същото време когато са решили това, са добавили и нови методи към **System** изискващи **PrintStream** заместо **PrintWriter**, който е новият и предпочитан заместител. Това е мистерия.

Компресия

Java 1.1 също е добавил класове за четене и писане на файлове в компресиран формат. Те обгръщат съществуващите IO за получаване на възможност за компресиране/декомпресиране на данните.

Един аспект на тези Java 1.1 класове изпъква: Те не са извлечени от **Reader** и **Writer**, а вместо това са част от **InputStream** и **OutputStream** йерархийте. Така може да се наложи да смесите двата типа потоци. (Помните че може да използвате **InputStreamReader** и **OutputStreamWriter** за лесно преминаване от единия в другия и обратно.)

Java 1.1 Клас за компресия	Функция
CheckedInputStream	GetChecksum() дава контролна сума

Java 1.1 Клас за компресия	Функция
	за всеки InputStream (не само декомпресия)
CheckedOutputStream	GetCheckSum() дава контролна сума за всеки OutputStream (не само декомпресия)
DeflaterOutputStream	Базов за класовете за компресия
ZipOutputStream	DeflaterOutputStream който компресира във формата на Zip файл
GZIPOutputStream	DeflaterOutputStream който компресира във формата на GZIP файл
InflaterInputStream	Базов за класовете за декомпресия
ZipInputStream	DeflaterInputStream който декомпресира от Zip формат
GZIPInputStream	DeflaterInputStream който декомпресира от GZIP формат

Макар и да има много алгоритми за компресия, Zip и GZIP са вероятно най-използваните. Така че лесно може да работите с компресираните си данни чрез множеството налични инструменти.

Проста компресия с GZIP

GZIP интерфейсът е прост и затова може би по-подходящ когато имате едно парче данни за компресиране (в противоположност на множество различни парчета). Ето пример за компресиране на един файл:

```
//: c10:ZipCompress.java
// Uses Java 1.1 Zip compression to compress
// any number of files whose names are passed
// on the command line.
import java.io.*;
import java.util.*;
import java.util.zip.*;

public class ZipCompress {
    public static void main(String[] args) {
        try {
            FileOutputStream f =
                new FileOutputStream("test.zip");
            CheckedOutputStream csum =
                new CheckedOutputStream(
                    f, new Adler32());
            ZipOutputStream out =
                new ZipOutputStream(
                    new BufferedOutputStream(csum));
            out.setComment("A test of Java Zipping");
            // Can't read the above comment, though
            for(int i = 0; i < args.length; i++) {
                System.out.println(
                    "Writing file " + args[i]);
                BufferedReader in =
                    new BufferedReader(
                        new FileReader(args[i]));
                out.putNextEntry(new ZipEntry(args[i]));
                int c;
                while((c = in.read()) != -1)
                    out.write(c);
                in.close();
            }
            out.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

int c;
while((c = in.read()) != -1)
    out.write(c);
in.close();
}
out.close();
// Checksum valid only after the file
// has been closed!
System.out.println("Checksum: " +
    csum.getChecksum().getValue());
// Now extract the files:
System.out.println("Reading file");
FileInputStream fi =
    new FileInputStream("test.zip");
CheckedInputStream csumi =
    new CheckedInputStream(
        fi, new Adler32());
ZipInputStream in2 =
    new ZipInputStream(
        new BufferedInputStream(csumi));
ZipEntry ze;
System.out.println("Checksum: " +
    csumi.getChecksum().getValue());
while((ze = in2.getNextEntry()) != null) {
    System.out.println("Reading file " + ze);
    int x;
    while((x = in2.read()) != -1)
        System.out.write(x);
}
in2.close();
// Alternative way to open and read
// zip files:
ZipFile zf = new ZipFile("test.zip");
Enumeration e = zf.entries();
while(e.hasMoreElements()) {
    ZipEntry ze2 = (ZipEntry)e.nextElement();
    System.out.println("File: " + ze2);
    // ... and extract the data as before
}
} catch(Exception e) {
    e.printStackTrace();
}
}
}
} //:~

```

Използването на класовете е праволинейно – просто обгръщате изходния си поток с **GZIPOutputStream** или **ZipOutputStream** и входният си поток с **GZIPInputStream** или **ZipInputStream**. Всичко друго е обикновено IO четене и писане. Това обаче е хубав пример за случай, когато сте принудени да смесвате старите и новите IO потоци: **in** използва **Reader** класове, докато конструктора на **GZIPOutputStream** може да приеме само **OutputStream** обект, не **Writer** обект.

МНОГО ФАЙЛОВЕ СЪС Zip

Библиотеката на Java 1.1 която поддържа Zip формата е много по-богата. С нея може лесно да компресирате множество файлове, даже има отделен клас който прави четенето на Zip

файл лесно. Библиотеката използва стандартния Zip формат така че несъмнено е съвместива с данните и инструментите които могат да се изтеглят от Мрежата. Следният пример има същата форма като предишния, но позволява колкото искате аргументи на командния ред. Освен това показва използването на **Checksum** класовете за изчисление и проверка на контролна сума за файловете. Има два **Checksum** типа: **Adler32** (който е по-бърз) и **CRC32** (който е по-бавен но мъничко по-акуратен).

```
//: c10:ZipCompress.java
// Uses Java 1.1 Zip compression to compress
// any number of files whose names are passed
// on the command line.
import java.io.*;
import java.util.*;
import java.util.zip.*;

public class ZipCompress {
    public static void main(String[] args) {
        try {
            FileOutputStream f =
                new FileOutputStream("test.zip");
            CheckedOutputStream csum =
                new CheckedOutputStream(
                    f, new Adler32());
            ZipOutputStream out =
                new ZipOutputStream(
                    new BufferedOutputStream(csum));
            out.setComment("A test of Java Zipping");
            // Can't read the above comment, though
            for(int i = 0; i < args.length; i++) {
                System.out.println(
                    "Writing file " + args[i]);
                BufferedReader in =
                    new BufferedReader(
                        new FileReader(args[i]));
                out.putNextEntry(new ZipEntry(args[i]));
                int c;
                while((c = in.read()) != -1)
                    out.write(c);
                in.close();
            }
            out.close();
            // Checksum valid only after the file
            // has been closed!
            System.out.println("Checksum: " +
                csum.getChecksum().getValue());
            // Now extract the files:
            System.out.println("Reading file");
            FileInputStream fi =
                new FileInputStream("test.zip");
            CheckedInputStream csumi =
                new CheckedInputStream(
                    fi, new Adler32());
            ZipInputStream in2 =
                new ZipInputStream(
                    new BufferedInputStream(csumi));
            ZipEntry ze;
            System.out.println("Checksum: " +
```

```

csumi.getChecksum().getValue());
while((ze = in2.getNextEntry()) != null) {
    System.out.println("Reading file " + ze);
    int x;
    while((x = in2.read()) != -1)
        System.out.write(x);
}
in2.close();
// Alternative way to open and read
// zip files:
ZipFile zf = new ZipFile("test.zip");
Enumeration e = zf.entries();
while(e.hasMoreElements()) {
    ZipEntry ze2 = (ZipEntry)e.nextElement();
    System.out.println("File: " + ze2);
    // ... and extract the data as before
}
} catch(Exception e) {
    e.printStackTrace();
}
}
}
} //:~

```

За да добавите всеки файл към архива трябва да извикате **putNextEntry()** и да му дадете **ZipEntry** обект. **ZipEntry** съдържа богат интерфейс който позволява да добавите всичките данни налични в случая в Zip файла: име, дължини компресиран и некомпресиран, дата, CRC сума, допълнителни данни, коментар, метод на компресия и път ако има. Макар и Zip форматът да има начин за даване на парола, това не се поддържа в библиотеката на Java за Zip. И макар че **CheckedInputStream** и **CheckedOutputStream** поддържат и **Adler32** и **CRC32** контролни суми, класът **ZipEntry** поддържа само интерфейс за CRC. Това е ограничение произтичащо от подлежащия Zip формат, но би могло да ви отклони от използването на побързия **Adler32**.

За извлечане на файлове **ZipInputStream** има **getNextEntry()** метод който връща следващия **ZipEntry** ако има такъв. Като по-сбита алтернатива може да прочетете файла със **ZipFile** обект, който има метод **entries()** с връщане на **Enumeration** към **ZipEntries**-ите.

За да четете контролната сума трябва да имате някакъв достъп до асоциирания **Checksum** обект. Тук се запазват манипулятори към **CheckedOutputStream** и **CheckedInputStream**, но също би могъл да бъде и манипулятор на **Checksum** обект.

Объркващ метод в Zip потоците е **setComment()**. Както е показано по-горе, може да се слага коментар когато се пише файл, но няма начин да се възстанови той в **ZipInputStream**. Коментарите изглежда се поддържат за всяка порция данни чрез **ZipEntry**.

Разбира се не сте ограничени до файлове когато използвате **GZIP** или **Zip** библиотеките – може да компресирате каквите и да е данни, включително за изпращане по Интернет.

Java archive (jar) утилитито

Zip форматът се използва също и в Java 1.1 JAR (Java ARchive) файловия формат, който е начин да се събере група файлове в един компресиран файл, точно както Zip. Само че както всичко в Java, JAR файловете са многоплатформени, така че не трябва да се беспокоите по въпросите за преносимост. Може да се включват и звукови файлове и образи, както и файлове с класове.

JAR файловете са в частност полезни когато имате работа с Internet. Преди JAR файловете вашият Web браузър трябва да прави множество поръчки към Web сървъра за да получи всичките файлове които съставят даден аплет. Освен това всеки файл беше некомпресиран. Чрез комбинирането на всички необходими за аплета файлове в единствен JAR файл остава необходим само един достъп и освен това обменът е по-бърз поради компресията. И всяка единица в JAR файла може да бъде цифрово подписана за сигурност (вижте Java документацията за подробности).

Един JAR файл се състои от множество зипнати файлове задедно с "манифест" който ги описва. (Може да създадете ваш собствен файл-манифест; иначе **jar** програмата ще го направи.) Може да намерите повече за JAR манифестите в онлайн документацията.

jar утилитито което идва с JDK на Sun автоматично компресира файлове по ваш избор. Вика се с команден ред:

```
| jar (options) destination (manifest) inputfile(s)
```

Опциите са просто колекция от букви (не са необходими чертички или други подобни). Те са:

c	Създава нов или празен архив.
t	Дава съдържанието.
x	Изважда всички файлове
x file	Изважда споменатия файл
f	Казва: "Ще ви дам името на файла." Ако не използвате това, jar предполага че ще чете от стандартния вход, или, ако създава файл, ще го праща на стандартния изход.
m	Казва че първият аргумент ще бъде името на потребителския манифестов файл
v	Създава изход описващ какво прави jar
o	Само запомня файловете; не ги компресира (използвайте го да създава JAR файл който може да сложите във вашия път към класовете)
M	Не създава автоматично манифестен файл

Ако файловете които се вкарват в JAR файла включват поддиректория, тя се добавя автоматично, включително всичките поддиректории и т.н. Информацията за пътя също се запазва.

Ето някои типични начини за викане на **jar**:

```
| jar cf myJarFile.jar *.class
```

Това създава JAR файл наречен **myJarFile.jar** който съдържа всичките класови файлове в текущата поддиректория, заедно с автоматично генериран манифестен файл.

```
| jar cmf myJarFile.jar myManifestFile.mf *.class
```

Като предишния пример, но с добавяне на създаден от потребителя манифест наречен **myManifestFile.mf**.

```
| jar tf myJarFile.jar
```

Дава съдържание за **myJarFile.jar**.

```
| jar tvf myJarFile.jar
```

Добавя флаг за извеждане на по-подробни сведения за **myJarFile.jar**.

```
| jar cvf myApp.jar audio classes image
```

Предполага се, че **audio**, **classes** и **image** са поддиректории, това комбинира всичко в поддиректориите във файл, наречен **myApp.jar**. Що се добавя флаг за повече обратна връзка докато програмата **jar** работи.

Ако създадете JAR чрез опцията **O**, този файл може да се сложи в CLASSPATH:

```
| CLASSPATH="lib1.jar;lib2.jar;"
```

Тогава Java може да търси **lib1.jar** и **lib2.jar** за файлове с класове.

Инструментът **jar** не е толкова полезен какото **zip** утилитито. Например не може да добавяте или обновявате файлове в JAR файл; може да създадете JAR файлове само "на празно място". Също не може да премествате файлове в JAR файл, изтривайки ги след вмъкването. Обаче един JAR файл създаден на някаква платформа ще бъде прозрачно четим от **jar** инструмента на която и да е друга платформа (проблем, който понякога вади душата на **zip** ютилитата).

Както ще видите в глава 13, JAR файлове се използват и в пакета Java Beans.

Сериализация на обекти

Java 1.1 е добавил интересна черта, наречена **сериализация на обекти** която позволява всеки обект който реализира **Serializable** интерфейса да се превърне в последователност от байтове, от която после може напълно да се възстанови оригиналния обект. Това е вярно даже и през мрежи, което значи че механизъмът компенсира автоматично разликите в операционните системи. Тоест може да създадете обект на Windows машина, да го сериализирате, да го изпратите през мрежата на Unix машина където той ще бъде коректно реконструиран. Не е необходимо да се беспокоите за данните типове на двете системи, подреждането на байтовете или други детайли.

Сама по себе си сериализацията на обекти е интересна с това, че позволява да се реализира лека устойчивост. Запомнете че обектовата устойчивост означава, че обектът няма време на живот колкото е времето на изпълнение на програмата – обектът съществува и между извикванията на програмата. Чрез вземане на сериализиран обект и записването му на диск, а после възстановяването му когато програмата се пусне пак, може да се постигне устойчивост. Причината тя да се нарича "лека" е че не може просто да я посочите чрез някаква ключова дума като "persistent" и да оставите системата да се грижи за детайлите (макар че това може би ще стане в бъдеще). Вместо това трябва явно да се сериализира и де-сериализира обектът в програмата.

Обектовата сериализация бе добавена в езика за поддържане на две главни черти. *Remote method invocation* (RMI) на Java 1.1 позволява обекти които живеят на друга машина да действат като са на вашата машина. Когато се изпращат съобщения на далечните обекти, сериализацията е необходима за изпращане на аргументите и връщането на стойности. RMI се разглежда в глава 15.

Сериализацията е също необходима за Java Beans, въведени с Java 1.1. Когато се използва Bean, неговата информация обикновено се задава по време на проектирането. Тази информация трябва да се запомни и после използва когато се пуска програмата; обектовата сериализация изпълнява тази задача.

Сериализирането на обект е доста просто, ако обектът реализира **Serializable** интерфейс (този интерфейс е само флаг и няма методи). В Java 1.1 много стандартни библиотечни класове са променени така че да могат да се сериализират, включително всички обвивки на примитивните типове, всичките класове-колекции и много други. Даже **Class** обекти може да се сериализират. (Виж глава 11 за такива неща.)

За да се сериализира обект трябва да се създаде някакъв **OutputStream** обект и после да се обгърне с **ObjectOutputStream** обект. В тази точка просто трябва да извикате **writeObject()** и вашият обект е сериализиран и изпратен в **OutputStream**. За да се направи обратното, обгръщате **InputStream** с **ObjectInputStream** и викате **readObject()**. Това което пристига обратно е, както обикновено, ъпкастнат манипулатор на **Object**, така че трябва с даункастинг да оправите нещата.

Един особено умен аспект на сериализацията на обекти е че тя не само записва даден обект, но проследява съдържащите се в него манипулатори и записва и тези обекти, и проследява всеки от съдържащите се в тези обекти манипулатори и т.н. Това понякога се нарича "паяжина от обекти" с която отделен обект може да бъде свързан и тя включва масиви от манипулатори на обекти както и обекти-членове. Ако трябваше да се обслужва собствена потребителска схема на сериализация, поддържането на кода обклужващ това множество връзки би било изумяващо. Обаче обектовата сериализация в Java го прави безуспорно, без съмнение използвайки оптимизиран алгоритъм. Следващият пример изprobва сериализационния механизъм създавайки "червей" от свързани обекти, всеки от които има връзка с обект от следващия член на червея, както и с масив от манипулатори към обекти в друг обект, **Data**:

```
//: c10:Worm.java
// Demonstrates object serialization in Java 1.1
import java.io.*;

class Data implements Serializable {
    private int i;
    Data(int x) { i = x; }
    public String toString() {
        return Integer.toString(i);
    }
}

public class Worm implements Serializable {
    // Generate a random int value:
    private static int r() {
        return (int)(Math.random() * 10);
    }
    private Data[] d = {
        new Data(r()), new Data(r()), new Data(r())
    };
    private Worm next;
    private char c;
    // Value of i == number of segments
    Worm(int i, char x) {
        System.out.println(" Worm constructor: " + i);
        c = x;
        if(--i > 0)
            next = new Worm(i, (char)(x + 1));
    }
    Worm() {
        System.out.println("Default constructor");
    }
    public String toString() {
        String s = ":" + c + "(";
        for(int i = 0; i < d.length; i++)
            s += d[i].toString();
        s += ")";
        if(next != null)
```

```

    s += next.toString();
    return s;
}
public static void main(String[] args) {
    Worm w = new Worm(6, 'a');
    System.out.println("w = " + w);
    try {
        ObjectOutputStream out =
            new ObjectOutputStream(
                new FileOutputStream("worm.out"));
        out.writeObject("Worm storage");
        out.writeObject(w);
        out.close(); // Also flushes output
        ObjectInputStream in =
            new ObjectInputStream(
                new FileInputStream("worm.out"));
        String s = (String)in.readObject();
        Worm w2 = (Worm)in.readObject();
        System.out.println(s + ", w2 = " + w2);
    } catch(Exception e) {
        e.printStackTrace();
    }
    try {
        ByteArrayOutputStream bout =
            new ByteArrayOutputStream();
        ObjectOutputStream out =
            new ObjectOutputStream(bout);
        out.writeObject("Worm storage");
        out.writeObject(w);
        out.flush();
        ObjectInputStream in =
            new ObjectInputStream(
                new ByteArrayInputStream(
                    bout.toByteArray()));
        String s = (String)in.readObject();
        Worm w3 = (Worm)in.readObject();
        System.out.println(s + ", w3 = " + w3);
    } catch(Exception e) {
        e.printStackTrace();
    }
}
} //:~

```

За да се направи интересно, масивът от **Data** обекти вътре в **Worm** се инициализира със случаини числа. (По този начин се избягва подозрението, че компилаторът държи някакви мета-данни.) Всеки сегмент на **Worm** е отбелян с **char** който автоматично се генерира в процеса на рекурсивната генерация на свързания списък от **Wormове**. Когато създавате **Worm**, казвате на конструктора колко дълъг искате да бъде. За да направите **next** манипулятор (следващия-б.пр.) той вика **Worm** конструктор с дължина по-малка с единица и т.н. Последния **next** манипулятор е оставен като **null**, показвайки края на **Worm** (червя-б.пр.).

Работата е в това да се създаде нещо достатъчно сложно, което не би могло лесно да се сериализира (по друг начин-б.пр.). Актът на сериализация, обаче, е доста прост. Веднъж като се създаде **ObjectOutputStream** от някакъв друг поток, **writeObject()** сериализира обекта. Забележете викането на **writeObject()** за **String**, също така. Може също да пишете всичките примитивни типове данни използвайки същите методи като **DataOutputStream** (те споделят един и същ интерфейс).

Има два отделни **try** блока които изглеждат подобни. Първият пише и чете файл, а вторият, за разнообразие, чете и пише **ByteArray**. Може да четете и пишете обект използвайки сериализация към всеки **DataInputStream** или **DataOutputStream** включително, както ще видите в главата за мрежите, през мрежа. Изходът от едно пускане е:

```
Worm constructor: 6  
Worm constructor: 5  
Worm constructor: 4  
Worm constructor: 3  
Worm constructor: 2  
Worm constructor: 1  
w = :a(262):b(100):c(396):d(480):e(316):f(398)  
Worm storage, w2 = :a(262):b(100):c(396):d(480):e(316):f(398)  
Worm storage, w3 = :a(262):b(100):c(396):d(480):e(316):f(398)
```

Може да видите, че възстановеният обект действително съдържа всички връзки от оригиналния обект.

Забележете че не се вика конструктор, нито даже и такъв по подразбиране, в процеса на десериализация на **Serializable** обект. Целият обект се възстановява от данните от **InputStream**.

Сериализацията на обекти е друга черта на Java 1.1 която не е част от новите **Reader** и **Writer** йерархии, а използва старите **InputStream** и **OutputStream** йерархии. Така може да се получат ситуации, при които сте принудени да смесите йерархийте.

Намиране на класа

Може би се чудите какво ли ще е необходимо за възстановяването на обект от неговото сериализирано състояние. Да кажем например че сте сериализирали обект и сте го изпратили през мрежа на друга машина. Би ли могла програма на далечната машина да реконструира обекта използвайки само данните от файла?

Най-добрият начин да се отговори на този въпрос е (както обикновено) чрез експеримент. Следният файл отива в поддиректорията за тази глава:

```
//: c10:Alien.java  
// A serializable class  
import java.io.*;  
  
public class Alien implements Serializable {  
} //:~
```

Файлът който създава и сериализира **Alien** обект отива в същата директория:

```
//: c10:FreezeAlien.java  
// Create a serialized output file  
import java.io.*;  
  
public class FreezeAlien {  
    public static void main(String[] args)  
        throws Exception {  
        ObjectOutputStream out =  
            new ObjectOutputStream(  
                new FileOutputStream("file.x"));  
        Alien zorcon = new Alien();  
        out.writeObject(zorcon);  
    }  
}
```

```
| } //:/~
```

Наместо да хваща и обработва изключения, тази програма възприема бързия и мръсен подход да разкарва изключенията като ги подава чак вън от **main()**, така че за тях се съобщава на командния ред (т.е. от ОС-б.пр.).

Щом програмата се компилира и пусне, копирайте получения файл **file.x** в поддиректория наречена **xfiles**, където отива и следния код:

```
//: c10:xfiles,ThawAlien.java
// Try to recover a serialized file without the
// class of object that's stored in that file.
package c10.xfiles;
import java.io.*;

public class ThawAlien {
    public static void main(String[] args)
        throws Exception {
        ObjectInputStream in =
            new ObjectInputStream(
                new FileInputStream("file.x"));
        Object mystery = in.readObject();
        System.out.println(
            mystery.getClass().toString());
    }
} //:/~
```

Тази програма отваря файла и чете обекта **mystery** успешно. Ако обаче се опитате да намерите нещо за обекта – което изисква **Class** обектът за **Alien** – виртуалната машина (JVM) не може да намери **Alien.class** (докато той не се случи на Classpath, където не бива да бъде в този пример). Получавате **ClassNotFoundException**. (Още веднъж, всякакви свидетелства за живота на alien (извънземни-б.пр.) изчезват преди да бъдат проверени!)

Ако ще правите нещо с обект който е реконструиран, трябва да осигурите че JVM може да намери съответния **.class** файл или локално на пътя за класовете или някъде в Internet.

Управление на сериализацията

Както се вижда, нормалният сериализационен механизъм е тривиален за използване. А ако има специални нужди? Оже би имате специални изисквания към сигурността и не искате да сериализирате някои части от вашия обект, а може би просто няма смисъл някакъв подобект да се сериализира и изпраща понеже при използване ще се генерира отново.

Може да управлявате процеса на сериализация чрез използване на **Externalizable** интерфейса вместо **Serializable** интерфейса. Интерфейсът **Externalizable** разширява **Serializable** интерфейса и добавя два метода, **writeExternal()** и **readExternal()**, които автоматично се викат по време на сериализацията и обратния процес (съответно-б.пр.) и позволяват да се направят специалните неща.

Следният пример показва прости реализации на интерфейсните методи на **Externalizable**. Забележете че **Blip1** и **Blip2** са почти идентични с изключение на малка разлика (вижте дали ще я откриете като четете кода):

```
//: c10:Blips.java
// Simple use of Externalizable & a pitfall
import java.io.*;
import java.util.*;
```

```

class Blip1 implements Externalizable {
    public Blip1() {
        System.out.println("Blip1 Constructor");
    }
    public void writeExternal(ObjectOutput out)
        throws IOException {
        System.out.println("Blip1.writeExternal");
    }
    public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException {
        System.out.println("Blip1.readExternal");
    }
}

class Blip2 implements Externalizable {
    Blip2() {
        System.out.println("Blip2 Constructor");
    }
    public void writeExternal(ObjectOutput out)
        throws IOException {
        System.out.println("Blip2.writeExternal");
    }
    public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException {
        System.out.println("Blip2.readExternal");
    }
}

public class Blips {
    public static void main(String[] args) {
        System.out.println("Constructing objects:");
        Blip1 b1 = new Blip1();
        Blip2 b2 = new Blip2();
        try {
            ObjectOutputStream o =
                new ObjectOutputStream(
                    new FileOutputStream("Blips.out"));
            System.out.println("Saving objects:");
            o.writeObject(b1);
            o.writeObject(b2);
            o.close();
            // Now get them back:
            ObjectInputStream in =
                new ObjectInputStream(
                    new FileInputStream("Blips.out"));
            System.out.println("Recovering b1:");
            b1 = (Blip1)in.readObject();
            // OOPS! Throws an exception:
            // System.out.println("Recovering b2:");
            // b2 = (Blip2)in.readObject();
            } catch(Exception e) {
                e.printStackTrace();
            }
        }
    } ///:~
}

```

Изходът от програмата е:

```
Constructing objects:  
Blip1 Constructor  
Blip2 Constructor  
Saving objects:  
Blip1.writeExternal  
Blip2.writeExternal  
Recovering b1:  
Blip1 Constructor  
Blip1.readExternal
```

Причината че **Blip2** обекта не е възстановен е че когато се прави това възниква изключение. Можете ли да забележите разликата между **Blip1** и **Blip2**? Конструктора на **Blip1** е **public**, докато този на **Blip2** не е, а това предизвиква изключение при възстановяването. Опитайте да направите конструктора на **Blip2** да е **public** като махнете `!!`. Коментарите за да видите коректните резултати.

Когато **b1** е възстановено вика се конструктора по подразбиране на **Blip1**. Това е различно от възстановяването на **Serializable** обект, където обектът се прави изцяло от запомнените негови битове, без извикване на конструктор(и). При **Externalizable** обект всичко с конструкторите си става както обикновено (включително инициализацията в точката на дефиниране на полетата), а **тогава readExternal()** се вика. Трябва да сте предупредени за това – в частност че конструкторът по подразбиране винаги играе – за да направите коректно поведението на вашите **Externalizable** обекти.

Ето пример който показва какво трябва да се направи за пълно запомняне и възстановяване на **Externalizable** обект:

```
//: c10:Blip3.java  
// Reconstructing an externalizable object  
import java.io.*;  
import java.util.*;  
  
class Blip3 implements Externalizable {  
    int i;  
    String s; // No initialization  
    public Blip3() {  
        System.out.println("Blip3 Constructor");  
        // s, i not initialized  
    }  
    public Blip3(String x, int a) {  
        System.out.println("Blip3(String x, int a)");  
        s = x;  
        i = a;  
        // s & i initialized only in non-default  
        // constructor.  
    }  
    public String toString() { return s + i; }  
    public void writeExternal(ObjectOutput out)  
        throws IOException {  
        System.out.println("Blip3.writeExternal");  
        // You must do this:  
        out.writeObject(s); out.writeInt(i);  
    }  
    public void readExternal(ObjectInput in)  
        throws IOException, ClassNotFoundException {  
        System.out.println("Blip3.readExternal");  
        // You must do this:
```

```

s = (String)in.readObject();
i =in.readInt();
}
public static void main(String[] args) {
    System.out.println("Constructing objects:");
    Blip3 b3 = new Blip3("A String ", 47);
    System.out.println(b3.toString());
    try {
        ObjectOutputStream o =
            new ObjectOutputStream(
                new FileOutputStream("Blip3.out"));
        System.out.println("Saving object:");
        o.writeObject(b3);
        o.close();
        // Now get it back:
        ObjectInputStream in =
            new ObjectInputStream(
                new FileInputStream("Blip3.out"));
        System.out.println("Recovering b3:");
        b3 = (Blip3)in.readObject();
        System.out.println(b3.toString());
    } catch(Exception e) {
        e.printStackTrace();
    }
}
} //:~

```

Полетата **s** и **i** се инициализират във втория конструктор, не в този по подразбиране. Това значи че ако не инициализирате **s** и **i** в **readExternal**, ще бъде **null** (понеже паметта на това място се нулира първо при създаването на обекта). Ако изкоментирате двата реда в кода следващи фразата "You must do this" и пуснете програмата, ще видите че когато обектът е възстановен, **s** е **null** и **i** е нула.

Ако наследявате от **Externalizable** обект, типично ще викате версите от базовия клас на **writeExternal()** и **readExternal()** за правилно запомняне и възстановяване на компонентите.

Така че за да станат нещата както трябва необходимо е не само да напишете важните данни на обекта чрез **writeExternal()** метода (няма поведение по подразбиране което да пише който и да е член на **Externalizable** обект), но също трябва и да възстановите въпросните данни чрез **readExternal()** метода. Това може да бъде малко смущаващо отначало понеже поведението по подразбиране при конструиране на **Externalizable** обект може да създаде впечатление, че някакъв вид запомняне и възстановяване става автоматично. Това не става.

Ключовата дума `transient`

В процеса на управление на сериализацията би могло да се случи конкретен обект да не е желателно да бъде запазен и изваждан от сериализационния механизъм на Java автоматично. Това обикновено е случаят, когато подобектът носи важна информация, която не бихте искали да се сериализира, като например парола. Даже тази информация да е **private** в обекта, веднъж сериализирана тя може да стане достъпна за някой който е чел файла или подслушал предаването по мрежата.

Един начин да се предотврати сериализацията на важни ваши обекти е да се използва **Externalizable**, както беше показано. Тогава нищо не се сериализира и трябва да посочите кое да бъде сериализирано явно в **writeExternal()**.

Ако работите със **Serializable** обект, обаче, сериализацията става автоматично. За да се управлява това, може да включвате и изключвате сериализацията поле по поле чрез

ключовата дума **transient**, която казва "Не се занимавай със запазването и възстановяването на това - аз ще се погрижа."

Например да вземем **Login** обект който пази информация за конкретна сесия. Да кажем, че след като е потвърдено влизането, трябва да се запазят данни, но без паролата. Най-лесният начин да се направи това е да се приложи **Serializable** и да се направи полето **password** да бъде **transient**. Ето как изглежда това:

```
//: c10:Logon.java
// Demonstrates the "transient" keyword
import java.io.*;
import java.util.*;

class Logon implements Serializable {
    private Date date = new Date();
    private String username;
    private transient String password;
    Logon(String name, String pwd) {
        username = name;
        password = pwd;
    }
    public String toString() {
        String pwd =
            (password == null) ? "(n/a)" : password;
        return "logon info: \n  +
               "username: " + username +
               "\n  date: " + date.toString() +
               "\n  password: " + pwd;
    }
    public static void main(String[] args) {
        Logon a = new Logon("Hulk", "myLittlePony");
        System.out.println( "logon a = " + a);
        try {
            ObjectOutputStream o =
                new ObjectOutputStream(
                    new FileOutputStream("Logon.out"));
            o.writeObject(a);
            o.close();
            // Delay:
            int seconds = 5;
            long t = System.currentTimeMillis()
                     + seconds * 1000;
            while(System.currentTimeMillis() < t)
                ;
            // Now get them back:
            ObjectInputStream in =
                new ObjectInputStream(
                    new FileInputStream("Logon.out"));
            System.out.println(
                "Recovering object at " + new Date());
            a = (Logon)in.readObject();
            System.out.println( "logon a = " + a);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
} ///:~
```

Може да се види че полетата **date** и **username** са обикновени (не **transient**) и като такива са сериализирани автоматично. **password** обаче е **transient**, така че не е запомняно на диска; също и сериализационния механизъм не прави опит да го възстановява. Изходът е:

```
logon a = logon info:  
    username: Hulk  
    date: Sun Mar 23 18:25:53 PST 1997  
    password: myLittlePony  
Recovering object at Sun Mar 23 18:25:59 PST 1997  
logon a = logon info:  
    username: Hulk  
    date: Sun Mar 23 18:25:53 PST 1997  
    password: (n/a)
```

Когато обектът е възстановен, полето **password** е **null**. Забележете че **toString()** трябва да провери за **null** стойност на **password** понеже ако се опитате да монтирате **String** обект чрез претоварения '+' оператор и тай намери **null** манипулатор, ще получите **NullPointerException**. (По-нови версии на Java биха могли да имат код за избягване на този проблем.)

Може също да видите че полето **date** е запазено на диска и възстановено от него и не е запълвано наново (от системната дата - б.пр.).

Тъй като **Externalizable** обекти не запомнят никой двои полета на диска по подразбиране, ключовата дума **transient** е за използване само със **Serializable** обекти.

Алтернатива на **Externalizable**

Ако не горите от желание да прилагате **Externalizable** интерфейс, има друг подход. Може да приложите **Serializable** интерфейс и да добавите (забележете че казвам "добавите" а не "подписнете" или "реализирате") методи наречени **writeObject()** и **readObject()** които автоматично ще бъдат викани когато обектите биват сериализирани или десериализирани, респективно. Тоест ако дадете тези методи, теще бъдат използвани вместо стандартната сериализация.

Тези методи трябва да имат точно следните сигнатури:

```
private void  
    writeObject(ObjectOutputStream stream)  
        throws IOException;  
  
private void  
    readObject(ObjectInputStream stream)  
        throws IOException, ClassNotFoundException
```

От гледна точка на дизайна нещата стават съвсем странни тук. Преди всичко, може да се помисли, че понеже тези методи не са част от базов клас или **Serializable** интерфейс, те трябва да бъдат дефинирани в техни собствени интерфейси. Но забележете че те са дефинирани като **private**, което значи че трябва да се викат само от други членове на техния клас. Обаче на практика не ги викате от членове на класа, ами **writeObject()** и **readObject()** методите на **ObjectOutputStream** и **ObjectInputStream** обекти викат **writeObject()** и **readObject()** методите на вашите обекти. (Забележете моето страшно въздържане да навляза в остра критика тук заради използването на едни и същи имена на методи. С една дума: смущаващо.) Може да се чудите защо **ObjectOutputStream** и **ObjectInputStream** обектите имат достъп до **private** методи на вашия клас. Може само да предполагаме, че това е част от магията на сериализацията.

Във всеки случай, всичко дефинирано в **interface** е автоматично **public** така че ако **writeObject()** и **readObject()** трябва да бъдат **private**, те не могат да бъдат част от **interface**.

След като трябва да спазите сигнатурите точно, ефектът е същият като от реализацията на **interface**.

Би изглеждало че когато викате **ObjectOutputStream.writeObject()**, **Serializable** обектът който подавате е разпитван (чрез размисление, няма съмнение) за да се види дали реализира свой собствен **writeObject()**. Ако да, нормалният процес на сериализация се пропуска и се вика **writeObject()**. Същата ситуация за **readObject()**.

Има една друга особеност. Вътрешните ви **writeObject()** може да изберете да използвате **writeObject()** действието по подразбиране викайки **defaultWriteObject()**. Подобно, в **readObject()** може да извикате **defaultReadObject()**. Ето прост пример който показва как може да се управлява запазването и възстановяването на **Serializable** обект:

```
//: c10:SerialCtl.java
// Controlling serialization by adding your own
// writeObject() and readObject() methods.
import java.io.*;

public class SerialCtl implements Serializable {
    String a;
    transient String b;
    public SerialCtl(String aa, String bb) {
        a = "Not Transient: " + aa;
        b = "Transient: " + bb;
    }
    public String toString() {
        return a + "\n" + b;
    }
    private void
        writeObject(ObjectOutputStream stream)
        throws IOException {
        stream.defaultWriteObject();
        stream.writeObject(b);
    }
    private void
        readObject(ObjectInputStream stream)
        throws IOException, ClassNotFoundException {
        stream.defaultReadObject();
        b = (String)stream.readObject();
    }
    public static void main(String[] args) {
        SerialCtl sc =
            new SerialCtl("Test1", "Test2");
        System.out.println("Before:\n" + sc);
        ByteArrayOutputStream buf =
            new ByteArrayOutputStream();
        try {
            ObjectOutputStream o =
                new ObjectOutputStream(buf);
            o.writeObject(sc);
            // Now get it back:
            ObjectInputStream in =
                new ObjectInputStream(
                    new ByteArrayInputStream(
                        buf.toByteArray()));
            SerialCtl sc2 = (SerialCtl)in.readObject();
            System.out.println("After:\n" + sc2);
        }
    }
}
```

```
    } catch(Exception e) {
        e.printStackTrace();
    }
}
} //:/~
```

В този пример едното **String** поле е обикновено а другото **transient**, за да се види че не-**transient** полето се запазва чрез **defaultWriteObject()** метода и **transient** полето се запазва и възстановява явно. Полетата се инициализират в конструктора вместо в точката на дефиницията им за да се докаже, че те не са инициализирани автоматично по някакъв механизъм по време на десериализацията.

Ако се гответе да използвате механизма по подразбиране за да пишете не-**transient** частите от вашия обект, трябва да извикате **defaultWriteObject()** като първа операция във **writeObject()** и **defaultReadObject()** като първа операция в **readObject()**. Това са странни викания на методи. Сякаш, например, викате **defaultWriteObject()** за **ObjectOutputStream** без аргументи, и все пак някакси той се оглежда наоколо и знае необходимите манипулатори за да напише не-**transient** частите. Призрачно.

Запазването и възстановяването на **transient** обекти използва по-познат код. Все пак да помислим какво става тук. В **main()** се създава **SerialCtl** обект, после се сериализира в **ObjectOutputStream**. (Забележете в този случай, че се използва буфер вместо файл – и това може за **ObjectOutputStream**.) Сериилизацията става на реда:

```
o.writeObject(sc);
```

Методът **writeObject()** трябва да провери **sc** дали има собствен **writeObject()** метод. (Не чрез преглед на интерфейса – няма такъв – или типа на класа, но чрез истински лов на метода използвайки рефлексия.) Ако има, той бива използван. Подобен подход е в сила и за **readObject()**. Може би това е единствения практичен начин намерен за решаване на промлема, но наистина е странен.

Промяна на версиите

Възможно е да поискате да промените версията на сериализуем клас (обекти от оригиналния клас може да са запазени в база данни, например). Това се поддържа, но вероятно ще го използвате в съвсем специални случаи, а освен това изисква по-голяма дълбочина на познанието, та няма да го разглеждаме тук. JDK1.1 HTML документите които може да свалите от Sun (и които могат да са част от вашия Java пакет с онлайн документи) покрива тази тема достатъчно пълно.

Използване на устойчивостта

Твърде привлекателно е да използвате сериализацията за запазване на данни за състоянието на програма и възстановяването на същото състояние впоследствие. Но преди да се направи това, трябва да се отговори на някои въпроси. Какво ще стане ако запазите два обекта и двата съдържащи манипулатор към трети? Когато възстановявате двата обекта в оригиналното им състояние, само една появява на третия обект ли ще има? Какво ще стане, ако сериализирате вашите обекти в различни файлове и ги десериализирате в различни места на програмата?

Ето пример, който показва проблема:

```
//: c10:MyWorld.java
import java.io.*;
import java.util.*;

class House implements Serializable {}
```

```

class Animal implements Serializable {
    String name;
    House preferredHouse;
    Animal(String nm, House h) {
        name = nm;
        preferredHouse = h;
    }
    public String toString() {
        return name + "(" + super.toString() +
            "), " + preferredHouse + "\n";
    }
}

public class MyWorld {
    public static void main(String[] args) {
        House house = new House();
        ArrayList animals = new ArrayList();
        animals.add(
            new Animal("Bosco the dog", house));
        animals.add(
            new Animal("Ralph the hamster", house));
        animals.add(
            new Animal("Fronk the cat", house));
        System.out.println("animals: " + animals);

        try {
            ByteArrayOutputStream buf1 =
                new ByteArrayOutputStream();
            ObjectOutputStream o1 =
                new ObjectOutputStream(buf1);
            o1.writeObject(animals);
            o1.writeObject(animals); // Write a 2nd set
            // Write to a different stream:
            ByteArrayOutputStream buf2 =
                new ByteArrayOutputStream();
            ObjectOutputStream o2 =
                new ObjectOutputStream(buf2);
            o2.writeObject(animals);
            // Now get them back:
            ObjectInputStream in1 =
                new ObjectInputStream(
                    new ByteArrayInputStream(
                        buf1.toByteArray()));
            ObjectInputStream in2 =
                new ObjectInputStream(
                    new ByteArrayInputStream(
                        buf2.toByteArray()));
            ArrayList animals1 = (ArrayList)in1.readObject();
            ArrayList animals2 = (ArrayList)in1.readObject();
            ArrayList animals3 = (ArrayList)in2.readObject();
            System.out.println("animals1: " + animals1);
            System.out.println("animals2: " + animals2);
            System.out.println("animals3: " + animals3);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}

```

```
    }  
} //:/~
```

Едното интересно нещо тук е че може да използвате сериализацията към байтово поле за правене на "дълбоко копие" от всякакви обекти които са **Serializable**. (Дълбоко копие значи че се дублицира цялата паяжина от обекти, а не само основния обект и манипулаторите в него.) Копирането е разгледано с дълбочина в глава 12.

Animal съдържа полета от тип **House**. В **main()** един **ArrayList** от тези **Animals** е създаден и после сериализиран в два отделни потока. Когато бъдат десериализирани и изведени, виждат се следните резултати от едно пускане (обектите ще бъдат в различни места на паметта при различните пускания):

```
animals: (Bosco the dog(Animal@1cc76c), House@1cc769  
, Ralph the hamster(Animal@1cc76d), House@1cc769  
, Fronk the cat(Animal@1cc76e), House@1cc769  
)  
animals1: (Bosco the dog(Animal@1cca0c), House@1cca16  
, Ralph the hamster(Animal@1cca17), House@1cca16  
, Fronk the cat(Animal@1cca1b), House@1cca16  
)  
animals2: (Bosco the dog(Animal@1cca0c), House@1cca16  
, Ralph the hamster(Animal@1cca17), House@1cca16  
, Fronk the cat(Animal@1cca1b), House@1cca16  
)  
animals3: (Bosco the dog(Animal@1cca52), House@1cca5c  
, Ralph the hamster(Animal@1cca5d), House@1cca5c  
, Fronk the cat(Animal@1cca61), House@1cca5c  
)
```

Разбира се очаква се десериализираните обекти да имат различни адреси от оригиналите си. Но забележете че в **animals1** и **animals2** се появяват едни и същи адреси, включително позоваванията на **House** обект който двата си споделят. От друга страна, когато **animals3** се възстановява, няма начин системата да знае че обектите във втория поток са синоними на обекти в първия, така че тя прави напълно отделна система обекти.

Докато сериализирате всичко в един поток ще може да възстановите същата система обекти която сте записали, без нежелано дублициране на обекти. Разбира се, бихте могли да промените състоянието на обектите през времето между записването на първия и последния, но това си е ваша отговорност – обектите ще бъдат записани в състоянието в което са в момента (и с каквото връзки са с други обекти) на сериализацията им.

Най-сигурното нещо по въпроса е да се направи "атомарна" операция. Ако сериализирате някакви неща, свършите някои работи, сериализирате още и т.н., тогава няма да запазите системата сигурно. Вместо това, сложете всичките обекти които отразяват състоянието на системата в единствена колекция и запишете тази колекция в една операция. После може да я възстановите с единствено викане на метод, също така.

Следният пример е за въображаема CAD система която демонстрира подхода. Освен това се разглежда и въпросът за използване на **static** полета – ако погледнете в документацията ще видите че **Class** е **Serializable**, така че трябва да е лесно да се запомнят **static** полета просто чрез сериализация на **Class** обект. Това изглежда смислен подход, във всеки случай.

```
//: c10:CADState.java  
// Saving and restoring the state of a  
// pretend CAD system.  
import java.io.*;  
import java.util.*;
```

```

abstract class Shape implements Serializable {
    public static final int
        RED = 1, BLUE = 2, GREEN = 3;
    private int xPos, yPos, dimension;
    private static Random r = new Random();
    private static int counter = 0;
    abstract public void setColor(int newColor);
    abstract public int getColor();
    public Shape(int xVal, int yVal, int dim) {
        xPos = xVal;
        yPos = yVal;
        dimension = dim;
    }
    public String toString() {
        return getClass().toString() +
            " color(" + getColor() +
            ") xPos(" + xPos +
            ") yPos(" + yPos +
            ") dim(" + dimension + ")\n";
    }
    public static Shape randomFactory() {
        int xVal = r.nextInt() % 100;
        int yVal = r.nextInt() % 100;
        int dim = r.nextInt() % 100;
        switch(counter++ % 3) {
            default:
            case 0: return new Circle(xVal, yVal, dim);
            case 1: return new Square(xVal, yVal, dim);
            case 2: return new Line(xVal, yVal, dim);
        }
    }
}

class Circle extends Shape {
    private static int color = RED;
    public Circle(int xVal, int yVal, int dim) {
        super(xVal, yVal, dim);
    }
    public void setColor(int newColor) {
        color = newColor;
    }
    public int getColor() {
        return color;
    }
}

class Square extends Shape {
    private static int color;
    public Square(int xVal, int yVal, int dim) {
        super(xVal, yVal, dim);
        color = RED;
    }
    public void setColor(int newColor) {
        color = newColor;
    }
    public int getColor() {
        return color;
    }
}

```

```

        }

}

class Line extends Shape {
    private static int color = RED;
    public static void
    serializeStaticState(ObjectOutputStream os)
        throws IOException {
        os.writeInt(color);
    }
    public static void
    deserializeStaticState(ObjectInputStream os)
        throws IOException {
        color = os.readInt();
    }
    public Line(int xVal, int yVal, int dim) {
        super(xVal, yVal, dim);
    }
    public void setColor(int newColor) {
        color = newColor;
    }
    public int getColor() {
        return color;
    }
}

public class CADState {
    public static void main(String[] args)
        throws Exception {
        ArrayList shapeTypes, shapes;
        if(args.length == 0) {
            shapeTypes = new ArrayList();
            shapes = new ArrayList();
            // Add handles to the class objects:
            shapeTypes.add(Circle.class);
            shapeTypes.add(Square.class);
            shapeTypes.add(Line.class);
            // Make some shapes:
            for(int i = 0; i < 10; i++)
                shapes.add(Shape.randomFactory());
            // Set all the static colors to GREEN:
            for(int i = 0; i < 10; i++)
                ((Shape)shapes.get(i))
                    .setColor(Shape.GREEN);
            // Save the state vector:
            ObjectOutputStream out =
                new ObjectOutputStream(
                    new FileOutputStream("CADState.out"));
            out.writeObject(shapeTypes);
            Line.serializeStaticState(out);
            out.writeObject(shapes);
        } else { // There's a command-line argument
            ObjectInputStream in =
                new ObjectInputStream(
                    new FileInputStream(args[0]));
            // Read in the same order they were written:
            shapeTypes = (ArrayList)in.readObject();
        }
    }
}

```

```

        Line.deserializeStaticState(in);
        shapes = (ArrayList)in.readObject();
    }
    // Display the shapes:
    System.out.println(shapes);
}
} //:~

```

Shape класът **implements Serializable**, така че всичко, което е наследило от **Shape** е автоматично **Serializable** също така. Всеки **Shape** съдържа данни, а всеки извлечен **Shape** клас съдържа **static** поле което определя цвета на всички **Shape**ове от него тип. (Слагането на **static** поле в базовия клас ще даде само едно поле, понеже **static** полетата не се дублицират в извлечените класове.) Методите на базовия клас могат да бъдат подтиснати за да се зададе цвят за различните типове (**static** методите не се свързват динамично, така че тези са нормални методи). **randomFactory()** методът създава различен **Shape** всеки път, когато се вика, използвайки случаен стойности за данните на **Shape**.

Circle и **Square** са праволинейни разширения на **Shape**; единствената разлика е че **Circle** инициализира **color** в точката на дефиницията а **Square** го инициализира в конструктора. Ще дискутираме **Line** по-късно.

В **main()**, един **ArrayList** се използва за **Class** обектите и друг за формите. Ако не зададете аргумент на командния ред **shapeTypes ArrayList** се създава и **Class** обекти се добавят, а после **shapes ArrayList** се създава и **Shape** се добавят. После всички **static color** стойности се поставят да бъдат **GREEN**, накрая всичко се сериализира във файла **CADState.out**.

Ако дадете аргумент на командния ред (предполагајмо **CADState.out**), този файл се отваря и използва за възстановяване на състоянието на програмата. В двете ситуации, резултиращият **ArrayList** от **Shape**ове се извежда. Резултатите от едно пускане са:

```

>java CADState
(class Circle color(3) xPos(-51) yPos(-99) dim(38)
, class Square color(3) xPos(2) yPos(61) dim(-46)
, class Line color(3) xPos(51) yPos(73) dim(64)
, class Circle color(3) xPos(-70) yPos(1) dim(16)
, class Square color(3) xPos(3) yPos(94) dim(-36)
, class Line color(3) xPos(-84) yPos(-21) dim(-35)
, class Circle color(3) xPos(-75) yPos(-43) dim(22)
, class Square color(3) xPos(81) yPos(30) dim(-45)
, class Line color(3) xPos(-29) yPos(92) dim(17)
, class Circle color(3) xPos(17) yPos(90) dim(-76)
)

>java CADState CADState.out
(class Circle color(1) xPos(-51) yPos(-99) dim(38)
, class Square color(0) xPos(2) yPos(61) dim(-46)
, class Line color(3) xPos(51) yPos(73) dim(64)
, class Circle color(1) xPos(-70) yPos(1) dim(16)
, class Square color(0) xPos(3) yPos(94) dim(-36)
, class Line color(3) xPos(-84) yPos(-21) dim(-35)
, class Circle color(1) xPos(-75) yPos(-43) dim(22)
, class Square color(0) xPos(81) yPos(30) dim(-45)
, class Line color(3) xPos(-29) yPos(92) dim(17)
, class Circle color(1) xPos(17) yPos(90) dim(-76)
)

```

Може да се види че стойностите **xPos**, **yPos**, и **dim** са били запазени и възстановени всичките успешно, но има нещо нередно с възстановяването на **static** информацията. Навсякъде е

вкарано '3', но не се възстановява същото. **Circle** имат стойност 1 (**RED**, което е дефиницията), а **Square**-те имат стойност 0 (помнете, те се инициализират в конструктора). Сякаш **static**-те не са се сериализирали въобще! Това е така – макар и **Class** да е **Serializable**, той не прави каквото се очаква. Така че ако искате да сериализирате **static**, трябва сами да го правите.

За това са **serializeStaticState()** и **deserializeStaticState()** **static** методите в **Line**. Може да се види, че те явно се викат в процесите на запазване и възстановяване. (Забележете че трябва да се поддържа редът на запазване и възстановяване.) Така за да се направи **CADState.java** да работи коректно трябва (1) Да се добави **serializeStaticState()** и **deserializeStaticState()** към фигураните, (2) Да се махне **ArrayList shapeTypes** и всичкият свързан с него код, (3) Да се добавят извиквания на съответните методи във фигураните.

Друго нещо за което може да се наложи да мислите е сигурността, понеже сериализацията също запазва **private** данни. Ако сигурността е важна, такива полета ще се отбележат с **transient**. Но тогава ще трябва да си намерите сигурен начин (кодиране и пр. - бел.пр.) за запазване на стойностите, така че **private** променливите да могат да се поставят както трябва после.

Проверка на стила

В тази секция ще погледнем по-завършен пример за използване на Java IO. Този проект е пряко полезен понеже изпълнява проверка дали вашите файлове отговарят на Java стила на писане както може да се намери на www.JavaSoft.com. Отваря всеки **.java** файл в текущата директория и извлича имената на класове и идентификаторите, после показва ако нещо не отговаря на Java стила.

За да работи програмата коректно, трябва първо да построяте хранилище за имена на класове, което да съдържа всички имена от Java библиотеката. Това се прави чрез преминаване на всичкия сурс във всичките поддиректории на Java библиотеката и пускане на **ClassScanner** за всяка поддиректория. Давайки като аргументи имената на файловете-хранилища (с един и същ път и име всеки път) и **-a** опция на командния ред за отбелязване че имената на класове ще се добавят в хранилището.

За да се използва програмата за проверка на код, дайте име на файл и хранилище за използване и я пуснете. Тя ще провери всички имена на файлове и идентификатори в текущата директория и ще ви каже кои не следват типичния за Java стил на капитализация.

Трябва да знаете, че програмата не е перфектна; понякога ще показва че има проблеми, но при преглед ще установите, че няма такива. Това е малко ядосващо, но е много по-добре като цяло, отколкото да се намерят всички тези имена непосредствено от кода.

Обяснението непосредствено следва листинга:

```
//: c10:ClassScanner.java
// Scans all files in directory for classes
// and identifiers, to check capitalization.
// Assumes properly compiling code listings.
// Doesn't do everything right, but is a very
// useful aid.
import java.io.*;
import java.util.*;

class MultiStringMap extends HashMap {
    public void add(String key, String value) {
        if(!containsKey(key))
            put(key, new ArrayList());
```

```

        ((ArrayList)get(key)).add(value);
    }
    public ArrayList getArrayList(String key) {
        if(!containsKey(key)) {
            System.err.println(
                "ERROR: can't find key: " + key);
            System.exit(1);
        }
        return (ArrayList)get(key);
    }
    public void printValues(PrintStream p) {
        Iterator k = keySet().iterator();
        while(k.hasNext()) {
            String oneKey = (String)k.next();
            ArrayList val = getArrayList(oneKey);
            for(int i = 0; i < val.size(); i++)
                p.println((String)val.get(i));
        }
    }
}

public class ClassScanner {
    private File path;
    private String[] fileList;
    private Properties classes = new Properties();
    private MultiStringMap
        classMap = new MultiStringMap(),
        identMap = new MultiStringMap();
    private StreamTokenizer in;
    public ClassScanner() {
        path = new File(".");
        fileList = path.list(new JavaFilter());
        for(int i = 0; i < fileList.length; i++) {
            System.out.println(fileList[i]);
            scanListing(fileList[i]);
        }
    }
    void scanListing(String fname) {
        try {
            in = new StreamTokenizer(
                new BufferedReader(
                    new FileReader(fname)));
            // Doesn't seem to work:
            // in.slashStarComments(true);
            // in.slashSlashComments(true);
            in.ordinaryChar('/');
            in.ordinaryChar('.');
            in.wordChars('_', '_');
            in.eolIsSignificant(true);
            while(in.nextToken() != StreamTokenizer.TT_EOF) {
                if(in.ttype == '/')
                    eatComments();
                else if(in.ttype ==
                    StreamTokenizer.TT_WORD) {
                    if(in.sval.equals("class") ||
                        in.sval.equals("interface")) {

```

```

// Get class name:
    while(in.nextToken() != StreamTokenizer.TT_EOF
        && in.ttype != StreamTokenizer.TT_WORD)
    ;
    classes.put(in.sval, in.sval);
    classMap.add(fname, in.sval);
}
if(in.sval.equals("import") || in.sval.equals("package"))
    discardLine();
else // It's an identifier or keyword
    identMap.add(fname, in.sval);
}
}
} catch(IOException e) {
    e.printStackTrace();
}
}

void discardLine() {
try {
    while(in.nextToken() != StreamTokenizer.TT_EOF
        && in.ttype != StreamTokenizer.TT_EOL)
        ; // Throw away tokens to end of line
} catch(IOException e) {
    e.printStackTrace();
}
}

// StreamTokenizer's comment removal seemed
// to be broken. This extracts them:
void eatComments() {
try {
    if(in.nextToken() != StreamTokenizer.TT_EOF) {
        if(in.ttype == '/')
            discardLine();
        else if(in.ttype != '*')
            in.pushBack();
        else
            while(true) {
                if(in.nextToken() ==
                    StreamTokenizer.TT_EOF)
                    break;
                if(in.ttype == '*')
                    if(in.nextToken() !=
                        StreamTokenizer.TT_EOF
                        && in.ttype == '/')
                        break;
            }
    }
} catch(IOException e) {
    e.printStackTrace();
}
}

```

```

public String() classNames() {
    String() result = new String(classes.size());
    Iterator e = classes.keySet().iterator();
    int i = 0;
    while(e.hasNext())
        result(i++) = (String)e.next();
    return result;
}
public void checkClassNames() {
    Iterator files = classMap.keySet().iterator();
    while(files.hasNext()) {
        String file = (String)files.next();
        ArrayList cls = classMap.getArrayList(file);
        for(int i = 0; i < cls.size(); i++) {
            String className =
                (String)cls.get(i);
            if(Character.isLowerCase(
                className.charAt(0)))
                System.out.println(
                    "class capitalization error, file: "
                    + file + ", class: "
                    + className);
        }
    }
}
public void checkIdentNames() {
    Iterator files = identMap.keySet().iterator();
    ArrayList reportSet = new ArrayList();
    while(files.hasNext()) {
        String file = (String)files.next();
        ArrayList ids = identMap.getArrayList(file);
        for(int i = 0; i < ids.size(); i++) {
            String id =
                (String)ids.get(i);
            if(!classes.contains(id)) {
                // Ignore identifiers of length 3 or
                // longer that are all uppercase
                // (probably static final values):
                if(id.length() >= 3 &&
                   id.equals(
                       id.toUpperCase()))
                    continue;
                // Check to see if first char is upper:
                if(Character.isUpperCase(id.charAt(0))){
                    if(reportSet.indexOf(file + id)
                        == -1){ // Not reported yet
                        reportSet.add(file + id);
                        System.out.println(
                            "Ident capitalization error in:"
                            + file + ", ident: " + id);
                    }
                }
            }
        }
    }
    static final String usage =

```

```

"Usage: \n" +
"ClassScanner classnames -a\n" +
"\tAdds all the class names in this \n" +
"\tdirectory to the repository file \n" +
"\tcalled 'classnames'\n" +
"ClassScanner classnames\n" +
"\tChecks all the java files in this \n" +
"\tdirectory for capitalization errors, \n" +
"\tusing the repository file 'classnames'";
private static void usage() {
    System.err.println(usage);
    System.exit(1);
}
public static void main(String[] args) {
    if(args.length < 1 || args.length > 2)
        usage();
    ClassScanner c = new ClassScanner();
    File old = new File(args[0]);
    if(old.exists()) {
        try {
            // Try to open an existing
            // properties file:
            InputStream oldlist =
                new BufferedInputStream(
                    new FileInputStream(old));
            c.classes.load(oldlist);
            oldlist.close();
        } catch(IOException e) {
            System.err.println("Could not open "
                + old + " for reading");
            System.exit(1);
        }
    }
    if(args.length == 1) {
        c.checkclassNames();
        c.checkIdentNames();
    }
    // Write the class names to a repository:
    if(args.length == 2) {
        if(!args[1].equals("-a"))
            usage();
        try {
            BufferedOutputStream out =
                new BufferedOutputStream(
                    new FileOutputStream(args[0]));
            c.classes.save(out,
                "Classes found by ClassScanner.java");
            out.close();
        } catch(IOException e) {
            System.err.println(
                "Could not write " + args[0]);
            System.exit(1);
        }
    }
}
}

```

```

class JavaFilter implements FilenameFilter {
    public boolean accept(File dir, String name) {
        // Strip path information:
        String f = new File(name).getName();
        return f.trim().endsWith(".java");
    }
} //:~

```

Класът **MultiStringMap** е инструмент който позволява да се проектира група стрингове върху отделен ключ. Както в предишния пример, използва се **HashMap** (този път с наследяване) с ключа като единствен стринг който се проектира в **ArrayList** стойност. Методът **add()** просто проверява дали вече има такъв ключ в **HashMap**, ако няма го слага там. Методът **getArrayList()** дава **ArrayList** за конкретен ключ, а **printValues()**, който основно е полезен за тестване, извежда всички стойности **ArrayList** по **ArrayList**.

За да се опрости живота, имената от стандартните Java библиотеки са пъхнати в **Properties** обект (от стандартната Java библиотека). Помнете че **Properties** обектът е **HashMap** който съдържа само **String** обекти и за ключа, и за стойността. Обаче той може да бъде запазван на диск и възстановяван от там с едно извикване на метод, така че е идеален за склад на имена. Фактически се нуждаем само от списък с имена и **HashMap** не може да приеме **null** за ключ или стойност. Така че един и същ обек ще се използва и за стойностите, и за ключовете.

За класовете във файловете от конкретна директория се използват два, two **MultiStringMap**: **classMap** и **identMap**. Също когато програмата тръгва тя товари склада за имена в **Properties** обект наречен **classes**, а когато се намери ново име на клас, то се добавя също към **classes** както и към **classMap**. По този начин **classMap** може да бъде използван за преминаване по всички класове в локалната директория, а **classes** може да бъде използван за проверка дали текущият токен е име на клас (което показва че започва дефиниция на обект или метод, така че се грабват следващите токени – до точка и запетая – и се слагат в **identMap**).

Конструкторът по подразбиране на **ClassScanner** създава списък от файлови имена (чрез **JavaFilter** реализацията на **FilenameFilter**, както е описано в глава 10). После вика **scanListing()** за всяко класово име.

Вътре в **scanListing()** сорсовия файл е отворен и се превръща в **StreamTokenizer**. По документация чрез даване на **true** на **slashStarComments()** и **slashSlashComments()** се очаква да се махнат тези коментари, но това май не е точно така (не работи в Java 1.0). Вместо това тези редове се изкоментират и се извличат от друг метод. За да стане това '/' трябва да бъде хванато като обикновен знак вместо да се остави **StreamTokenizer** да го погълне като част от коментар, а **ordinaryChar()** методът казва на **StreamTokenizer** да направи това. Това също е вярно за точки ('.'), понеже искаме извикванията на методи да са разпаднати на отделни идентификатори. Обаче подчертаващото тире, което обикновено се третира от **StreamTokenizer** като отделен знак, ще се остави като част от идентификатора, понеже се появява в такива **static final** стойности като **TT_EOF** и т.н., използвани в същата тази програма. Методът **wordChars()** взема количество знаци които искате да се добавят към онези които се оставят вътре в обработвания токен като една дума. Накрая, когато обработваме едноредов коментар или пренебрегваме ред трябва да знаем къде е знакът за край на ред, та чрез викане на **eolIsSignificant(true)** eol ще се покаже наместо да бъде погълнат от **StreamTokenizer**.

Останалото от **scanListing()** чете и реагира на токените до края на файла, означаванс връщане от **nextToken()** на **final static** стойност **StreamTokenizer.TT_EOF**.

Ако токенът е '/' това потенциално е коментар, така че **eatComments()** се вика да се разправя с него. Единствената друга ситуация от която сме заинтересовани тук е когато е в дума, като има няколко специални случая.

Ако думата е **class** или **interface** тогава следващият токен представя име на клас или интерфейс и се слага в **classes** и **classMap**. Ако думата е **import** или **package**, не искаме останалата част от реда. Всичко друго трябва да е идентификатор (от който се интересуваме) или ключова дума (от която не се интересуваме, но в редки случаи са изцяло с малки букви, така че не е лошо да се вкарат, за да не развалят работата). Всички се добавят в **identMap**.

Методът **discardLine()** е прост инструмент който следи за край на ред. Забележете че всеки път когато имате нов токен трябва да проверите за край на файла.

Методът **eatComments()** се вика винаги когато в основния цикъл на преглеждане се срещне наклонена черта. Това обаче не значи непременно че е намерен коментар, така че трябва да се види следващия токен за да се види дали е наклонена черта (в който случай редът се пренебрегва) или звезда. Но ако не е от тези, това значи че чертата трябва да се вкара обратно в главния цикъл! За щастие методът **pushBack()** позволява да “се вкара обратно” текущия токен на входния поток така че когато главният цикъл вика **nextToken()** той ще го намери бутнат обратно.

За удобство методът **classNames()** дава масив от всичките имена в колекцията **classes**. Този метод не се използва в програмата но е полезен при тестване.

Следващите два метода са където фактически се прави проверката. В **checkclassNames()** имената на класове се вземат от **classMap** (който, помнете, съдържа имената само от тази директория, организирани по файловото име така, че то може да бъде изведено заедно с грешното име на клас). Това се прави чрез слагане на всеки асоцииран **ArrayList** и преглеждането му, за да се види дали първият знак е малка буква. Ако да, извежда се съответното съобщение за грешка.

В **checkIdentNames()** има подобен подход: всяко име на идентификатор се взема от **identMap**. Ако името не е в списъка на **classes**, счита се че е идентификатор или ключова дума. Проверява се един специален случай: ако дължината на идентификатора е 3 или повече и всичките знаци са главни букви, този идентификатор се игнорира понеже вероятно е **static final** стойност като например **PI_EOF**. Разбира се, това не е перфектен алгоритъм, но той предполага че ще забележите в края на краишата всички думи само от главни букви.

Вместо да докладва всеки идентификатор който започва с главна буква, този метод следи кои вече са били докладвани в **ArrayList** наречен **reportSet()**. Това третира **ArrayList** като “множество” което ви казва дали елементът вече е в множеството. Елементът се получава чрез конкатенация на файловото име и идентификатора. Ако елементът не е в множеството, добавя се и после се докладва.

Останалата част от листинга включва **main()**, който е зает с обработката на аргументите на командния ред и установяване дали искате да правите склад за имена на класове от стандартна Java библиотека или да проверявате валидността на написан от вас код. В двата случая се прави **ClassScanner** обект.

Дали правите склад или използвате такъв, трябва да отворите съществуващия склад. Чрез проведене на **File** обект и проверка за съществуване може да решите дали да отворите файла и **load()** списъка **Properties classes** вътре в **ClassScanner**. (Класовете от склада се добавят към, а не подтискат класовете от конструктора на **ClassScanner**.) Ако дадете един аргумент на командния ред това значи че искате да направите проверка на имената на класове и идентификатори, но ако дадете два аргумента (вторият “**-a**”) правите склад за имена на класове. В този случай се отваря файл за изход и **Properties.save()** се използва за писане на списъка във файл, заедно със стринг, който съдържа заглавна информация за файла.

Резюме

Потоковата библиотека на Java види се удовлетворява основните изисквания: може да се направи четене и писане от/на конзолата, файл, блок памет, даже през Internet (както ще видите в глава 15). Възможно е (чрез наследяване от **InputStream** и **OutputStream**) да се създадат нови типове от входни и изходни обекти. Може даже да се добави разширяемост към някои видове обекти възприемани от поток чрез редефиниране на **toString()** метод който автоматично се вика когато подавате обект на метод който очаква **String** (Ограниченната "автоматична конверсия на типовете" в Java).

Има въпроси на които не е отговорено с документацията и дизайна на IO потоковата библиотека. Например би било приятно да може да се каже че искате да се изхвърли изключение когато се презаписва файл, отворен за извеждане – някои операционни системи позволяват да определите че ще се отваря файл за писане, но само ако още не съществува. В Java изглежда се очаква да използвате **File** обек за определяне дали файлът съществува, понеже ако го отворите като **FileOutputStream** или **FileWriter** той винаги ще бъде презаписан. Чрез представяне и на пътя, и на файловете класът **File** също предполага беден дизайн чрез нарушаване на максимата "Не се опитвай да правиш твърде много неща в един клас."

IO потоковата библиотека докарва смесени чувства. Тя прави повечето от нещата и е преносима. Но ако още не разбираете декораторският подход, дизайнът не е интуитивен, така че има допълнителна работа за ученето и предаването му. Той също не е завършен: няма поддръжка на вида форматиран изход което се поддържа в почти всички други IO пакети на езици. (Това не бе поправено в Java 1.1, който пропусна възможността да промени дизайна на библиотеката напълно, а вместо това добави даже повече специални случаи и сложност.) Промените на IO библиотеката в Java 1.1 не бяха замествания, а по-скоро добавки, изглежда, че проектантите на библиотеката не можаха да решат кое ще се изоставя и кое ще се предпочита, с резултат ядосващо много съобщения за използване на оstarели неща, показващи противоречията в дизайна на библиотеката.

Обаче като веднъж разберете декораторите и започнете да използвате библиотеката в ситуации, които изискват гъвкавост, може да започнете да печелите от този дизайн, в която точка неговата цена в добавъчни линии може да не ви дразни толкова много вече.

Упражнения

1. Отворете файл така че да може да го четете ред по ред. Четете всеки ред като **String** и сложете този **String** обект в **ArrayList**. Изведете всички редове в **ArrayList** в обратен ред.
2. Променете упражнение 1 така че името на файла да се дава като аргумент на командния ред.
3. Променете упражнение 2 също да отваря текстов файл така че да може да пишете текст в него. Напишете редовете в **ArrayList**, заедно с номерата им, във файла.
4. Променете упражнение 2 да промените всички редове в **ArrayList** да бъдат само с главни букви и изведете резултата на **System.out**.
5. Променете упражнение 2 да вземе допълнителни аргументи от думи които да намери във файла. Изведете всички редове в които има думите.

6. В **Blips.java**, копирайте файла и го преименувайте на **BlipCheck.java** и преименувайте класа **Blip2** на **BlipCheck** (правейки го **public** в това време). Махнете всички `//!` във файла и изпълнете програмата в този вид. После изкоментирайте конструктора по подразбиране на **BlipCheck**. Пуснете програмата и обясните защо работи.
7. В **Blip3.java** изкоментирайте двета реда след фразата "You must do this:" и пуснете програмата. Обясните резултата и защо той се различава от този с двете линии.
8. Превърнете **SortedWordCount.java** програмата да използва Java 1.1 IO потоци.
9. Поправете програмата **CADState.java** както е описано в текста.
10. (Intermediate) В глава 7, намерете **GreenhouseControls.java** примера, който се състои от три файла. В **GreenhouseControls.java** вътрешният клас **Restart()** има твърдо вградена система от събития. Променете програмата така, че да чете събитията и техните относителни времена от текстов файл. (Challenging: Използвайте factory метод от глава 16 за постройка на събитията.)

11: Идентификация на типа по време на изпълнение

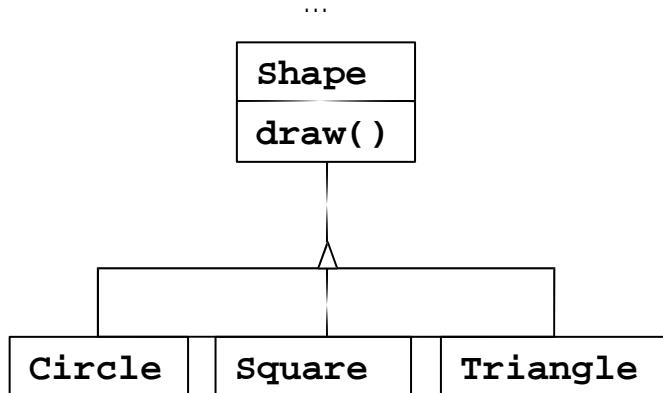
Идеята за това (RTTI) изглежда доста прости отначало: дава се възможност да се намери точният тип на обекта имайки само манипулятор към базовия тип.

Обаче, нуждата от RTTI разкрива лабиринт от интересни (и често обвръщащи) въпроси на ОО дизайн и поставя фундаментални въпроси относно структурирането на програмите.

Тази глава поглежда към начините по които Java подволява да се открие информация за обектите и класовете по време на изпълнение. Има две форми: "традиционното" RTTI, което предполага че всички типове са налични по време на компилация и по време на изпълнение и "reflection" механизма в Java 1.1, който позволява да се открие информация единствено по време на изпълнение. "Традиционното" RTTI ще се разгледа първо, следвано от дискусия за рефлексията.

Нуждата от RTTI

Да видим познатия сега пример с йерархия. Родов е базовият клас **Shape**, а специфичните извлечени класове са **Circle**, **Square** и **Triangle**:



Това е типична диаграма на йерархия на класове, с базовия клас най-отгоре и разрастваща се надолу с извлечените класове. Нормалната цел на ОО програмирането е да може вашият код да манипулира манипуляторите на базовия тип (**Shape**, в този случай), така че ако решите да разширите програмата с нов клас (**Rhombooid**, извлечен от **Shape**, например), кодът да не се засяга. В този пример динамично свързан метод в интерфейса на **Shape** е **draw()**, така че намерението е клиент-програмистът да вика **draw()** чрез родов **Shape** манипулятор. **draw()** е подтиснат във всичките извлечени класове, а понеже е динамично свързан метод, правилното

поведение ще е налице даже и да се вика чрез родовия **Shape** манипулятор. Това е полиморфизъм.

Изобщо, създавате специфичен обект (**Circle**, **Square** или **Triangle**), ъпкаствате го към **Shape** (забравяйки специфичния тип на обекта) и използвате този анонимен **Shape** манипулятор в останалата част на програмата.

Като кратък преглед на полиморфизма и ъпкастинга, бихте могли да кодирате горното така: (Вижте стр 63 ако имате проблеми с пускането на тази програма.)

```
//: c11:Shapes.java
package c11;
import java.util.*;

interface Shape {
    void draw();
}

class Circle implements Shape {
    public void draw() {
        System.out.println("Circle.draw()");
    }
}

class Square implements Shape {
    public void draw() {
        System.out.println("Square.draw()");
    }
}

class Triangle implements Shape {
    public void draw() {
        System.out.println("Triangle.draw()");
    }
}

public class Shapes {
    public static void main(String[] args) {
        ArrayList s = new ArrayList();
        s.add(new Circle());
        s.add(new Square());
        s.add(new Triangle());
        Iterator e = s.iterator();
        while(e.hasNext())
            ((Shape)e.next()).draw();
    }
} //:~
```

Базовият клас би могъл да бъде кодиран като **interface**, **abstract** клас, или обикновен клас. Понеже **Shape** няма конкретни членове (тоест такива с дефиниции) и не се очаква някога да създадете конкретен **Shape** обект, най-подходящото и гъвкаво представяне е **interface**. То също е и по-чисто защото нямате многобройните **abstract** ключови думи да се моткат наоколо.

Всеки от извлечените класове поддържа метода на базовия клас **draw** така че да се получи различно поведение. В **main()**, специфичните типове на **Shape** се създават и добавят към **ArrayList**. Това е мястото където става ъпкастът понеже **ArrayList** държи само **Objects**. Понеже всичко в Java (с изключение на примитивите) е **Object**, **ArrayList** може също да държи

Shape обекти. Но поради ъпкастът към **Object**, губи се всяка възможна специфична информация, включително фактът че обектите са **shape**ове. За **ArrayList** те са си **Objects**.

В точката в която вземате елемент от **ArrayList** с **next()**, нещата стават малко състени. Понеже **ArrayList** държи само **Objects**, **next()** естествено дава манипулятор към **Object**. Но ние знаем че той в действително е манипулятор на **Shape** и искаме да пратим **Shape** съобщения към него обект. Така че е необходим каст към **Shape** чрез традиционния начин “**(Shape)**”. Това е най-основната форма на RTTI, понеже в Java всички кастове се проверяват по време на изпълнение за коректност. Това е точно каквото значи RTTI: по време на изпълнение се намира типа на обекта.

В този случай кастът на RTTI е само частичен: **Object** е каст към **Shape**, а не по целия път до **Circle**, **Square** или **Triangle**. Това е защото единственото нещо което знаем в тази точка е че **ArrayList** е пълен с **Shape**ове. По време на компилация това е наложено по ваши собствени правила, но по време на изпълнение кастът го осигурява.

Сега се намесва полиморфизъмът и точният метод за **Shape** се определя по това дали конкретният манипулятор е за **Circle**, **Square** или **Triangle**. И изобщо, това е както трябва да бъде; вие искате вашия код да знае колкото е възможно по-малко от спецификата на типовете на обектите и само да се оправя с общото представяне на фамилия от обекти (в този случай, **Shape**). Като резултат вашият код ще е по-лесен за четене, писане и поддръжка, дизайнът — по-лесен за реализация, разбиране и промяна. Така че полиморфизъмът е обща цел в ООП.

Ами ако имате да решавате специфичен проблем, което по-лесно може да стане със знаенето на конкретния тип на манипулятора? Например да предположим че искате да позволите на вашите потребители да засветляват всичките фигури от конкретен тип чрез проведенето им морави. По този начин те биха могли да открият всичките триъгълници на екрана засветлявайки ги. Това е което свършва RTTI: може да питате за каъв точно тип се отнася даден манипулятор на **Shape**.

Обектът **Class**

За да се разбере как RTTI работи в Java първо трябва да се знае как информацията се представя по време на изпълнение. Това става чрез специален обект наречен обект **Class**, който съдържа информация за класа. (Това понякога се нарича мета-клас.) Фактически, обектът **Class** се използва за създаване на всички “нормални” обекти от вашия клас.

За всеки клас който във вашата програма има обект **Class**. Тоест всеки път когато пишете нов клас се създава единственный обект **Class** (и се запомня, съвсем правилно, в идентично наименован **.class** файл). По време на изпълнение, когато искате да направите обект от този клас, Java Virtual Machine (JVM) първо проверява дали обект **Class** за въпросния тип е натоварен. Ако не, JVM го товари (в паметта-б.пр.) от **.class** файл със същото име. Така една Java програма не е напълно натоварена преди да започне, което е различно от всички традиционни езици.

Щом веднъж **Class** обектът за конкретния тип е в паметта, той се използва за създаване на всички обекти от този тип.

Ако това изглежда мъгливо или не му вярвате, ето демонстрационна програма за доказателство:

```
//: c11:SweetShop.java
// Examination of the way the class loader works

class Candy {
    static {
        System.out.println("Loading Candy");
```

```

    }
}

class Gum {
    static {
        System.out.println("Loading Gum");
    }
}

class Cookie {
    static {
        System.out.println("Loading Cookie");
    }
}

public class SweetShop {
    public static void main(String[] args) {
        System.out.println("inside main");
        new Candy();
        System.out.println("After creating Candy");
        try {
            Class.forName("Gum");
        } catch(ClassNotFoundException e) {
            e.printStackTrace();
        }
        System.out.println(
            "After Class.forName(\"Gum\")");
        new Cookie();
        System.out.println("After creating Cookie");
    }
}
//:~

```

Всеки от класовете **Candy**, **Gum** и **Cookie** има **static** клауза която се изпълнява когато класът се зареди (в паметта-б.пр.) за пръв път. Извежда се съобщение че се товари съответния клас. В **main()** създаванията на обекти са обкръжени с оператори за извеждане за да се проследи създаването.

Един особено интересен ред е:

```
| Class.forName("Gum");
```

Този метод е **static** член на **Class** (към който принадлежат всички обекти от **Class**). **Class** обектът е като всеки друг и затова може да се вземе манипулятор към него и да се прави нещо с него. (Така прави и товарачът (в паметта - лоудър - б.пр.).) Един от начините да се вземе манипулятор към **Class** обект е **forName()**, който взема **String** съдържащ името (да се внимава с правописа и капитализацията!) на конкретен клас към който искате манипулятор. Той връща **Class** манипулятор.

Изходът от тази програма за една JVM е:

```

inside main
Loading Candy
After creating Candy
Loading Gum
After Class.forName("Gum")
Loading Cookie
After creating Cookie

```

Може да видите че всеки обект **Class** се товари само когато е необходим и че инициализацията на **static** се изпълнява при товаренето на класа.

Доста интересно, друга JVM дава:

```
| Loading Candy  
| Loading Cookie  
| inside main  
| After creating Candy  
| Loading Gum  
| After Class.forName("Gum")  
| After creating Cookie
```

Изглежда че тази JVM предвижда необходимостта от **Candy** и **Cookie** чрез преглеждане на кода в **main()**, но не е могла да види **Gum** понеже той е бил създаден чрез извикване на **forName()** и не чрез по-типичното извикване на **new**. Докато тази JVM дава желаните ефекти понеже товари класовете преди да има нужда от тях, не е ясно сигурно дали такова поведение е коректно.

Класни литерали

В Java 1.1 има втори начин за произвеждане на манипулятор към обект **Class**: чрез **класен литерал**. В горната програма това би изглеждало така:

```
| Gum.class;
```

Което не само е по-просто, но също и по-безопасно понеже се проверява по време на компилация. Понеже елиминира извикването на метод, така също е и по-ефективно.

Класният литерал работи както с обикновени класове, така и с интерфейси, масиви, примитиви. В добавка, има стандартно поле **TYPE** което съществува за всички обгръщащи примитиви класове. Полето **TYPE** дава манипулятор към **Class** обект за асоциирания примитивен тип, както тук:

... е еквивалентно на ...	
boolean.class	Boolean.TYPE
char.class	Character.TYPE
byte.class	Byte.TYPE
short.class	Short.TYPE
int.class	Integer.TYPE
long.class	Long.TYPE
float.class	Float.TYPE
double.class	Double.TYPE
void.class	Void.TYPE

Проверка преди каст

До тук сте видели следните форми на RTTI:

1. Класическия каст, напр. **(Shape)**, който използва RTTI за да осигури коректността на каста и изхвърли **ClassCastException** ако кастът е лош.
2. Обектът **Class** представящ типа на вашия обект. Обектът **Class** може да даде полезна информация по време на изпълнение.

В C++ класическият каст **"(Shape)"** не правят RTTI. Той просто казва на компилатора да третира указателя като на обект от посочения тип. В Java, където се прави проверка на типовете, този каст се нарича често "безопасен даункаст на типа." Причината за термина "downcast" е историческа: представянето на диаграмата. Ако се превръща **Circle** в **Shape** е

ъпкаст, тогава от **Shape** към **Circle** ще е даункаст. Обаче вие знаете че **Circle** също е **Shape**, а компилаторът свободно разрешава ъпкаст, но не знаете дали **Shape** е непременно **Circle**, така че компилаторът не прави такъв каст докато не го напишете явно.

Има и трета форма на RTTI в Java. Това е ключовата дума **instanceof** която казва дали обектът е екземпляр от някакъв конкретен тип. Тя връща **boolean** така че се използва като въпрос:

```
if(x instanceof Dog)
  ((Dog)x).bark();
```

Горния **if** оператор проверява дали **x** принадлежи на класа **Dog** преди превръщането на **x** към **Dog**. Важно е да се използва **instanceof** преди даункаст, когато няма друга информация за типа; иначе ще се случи **ClassCastException**.

Обикновено ще ловувате за един тип (триъгълници за да се направят морави, например), но следната програма показва как да се покрият всички обекти чрез **instanceof**.

```
//: c11:petcount:PetCount.java
// Using instanceof
package c11.petcount;
import java.util.*;

class Pet {}
class Dog extends Pet {}
class Pug extends Dog {}
class Cat extends Pet {}
class Rodent extends Pet {}
class Gerbil extends Rodent {}
class Hamster extends Rodent {}

class Counter { int i; }

public class PetCount {
  static String[] typenames = {
    "Pet", "Dog", "Pug", "Cat",
    "Rodent", "Gerbil", "Hamster",
  };
  public static void main(String[] args) {
    ArrayList pets = new ArrayList();
    try {
      Class[] petTypes = {
        Class.forName("c11.petcount.Dog"),
        Class.forName("c11.petcount.Pug"),
        Class.forName("c11.petcount.Cat"),
        Class.forName("c11.petcount.Rodent"),
        Class.forName("c11.petcount.Gerbil"),
        Class.forName("c11.petcount.Hamster"),
      };
      for(int i = 0; i < 15; i++)
        pets.add(
          petTypes[
            (int)(Math.random()*petTypes.length)]
          .newInstance());
    } catch(InstantiationException e) {}
    catch(IllegalAccessException e) {}
    catch(ClassNotFoundException e) {}
    HashMap h = new HashMap();
    for(int i = 0; i < typenames.length; i++)
```

```

h.put(typenames(i), new Counter());
for(int i = 0; i < pets.size(); i++) {
    Object o = pets.get(i);
    if(o instanceof Pet)
        ((Counter)h.get("Pet")).i++;
    if(o instanceof Dog)
        ((Counter)h.get("Dog")).i++;
    if(o instanceof Pug)
        ((Counter)h.get("Pug")).i++;
    if(o instanceof Cat)
        ((Counter)h.get("Cat")).i++;
    if(o instanceof Rodent)
        ((Counter)h.get("Rodent")).i++;
    if(o instanceof Gerbil)
        ((Counter)h.get("Gerbil")).i++;
    if(o instanceof Hamster)
        ((Counter)h.get("Hamster")).i++;
}
for(int i = 0; i < pets.size(); i++)
    System.out.println(
        pets.get(i).getClass().toString());
for(int i = 0; i < typenames.length; i++)
    System.out.println(
        typenames(i) + " quantity: " +
        ((Counter)h.get(typenames(i))).i);
}
} //:~

```

Има малко тясно ограничение върху **instanceof** в Java 1.0: Може да го сравнявате с именуван тип само, а не с **Class** обект. В примера по-горе е ясно че би било досадно да се пишат всички тези **instanceof** изрази. Но в Java 1.0 няма начин умно да се автоматизира това чрез **ArrayList** от **Class** обекти и сравняване. Това не е така голямо ограничение, както може да се помисли, понеже накрая ще се бламира проектът, ако трябва да напишете всички тези **instanceof** изрази.

Разбира се този пример е нарочен – вие вероятно бихте сложили **static** даннов член във всеки тип и инкрементирати го в конструктора ще се пази информация за броя. Бихте правили така ако имахте сурса и можехте да го променяте. Тъй като това не е обикновения случай, RTTI може да дойде твърде на място.

Използване на класни литерали

It's interesting to see how the **PetCount.java** example can be rewritten using Java 1.1 class literals. The result is cleaner in many ways:

```

//: c11:petcount2:PetCount2.java
// Using Java 1.1 class literals
package c11.petcount2;
import java.util.*;

class Pet {}
class Dog extends Pet {}
class Pug extends Dog {}
class Cat extends Pet {}
class Rodent extends Pet {}
class Gerbil extends Rodent {}
class Hamster extends Rodent {}

```

```

class Counter { int i; }

public class PetCount2 {
    public static void main(String[] args) {
        ArrayList pets = new ArrayList();
        Class[] petTypes = {
            // Class literals work in Java 1.1+ only:
            Pet.class,
            Dog.class,
            Pug.class,
            Cat.class,
            Rodent.class,
            Gerbil.class,
            Hamster.class,
        };
        try {
            for(int i = 0; i < 15; i++) {
                // Offset by one to eliminate Pet.class:
                int rnd = 1 + (int)(Math.random() * (petTypes.length - 1));
                pets.add(
                    petTypes(rnd).newInstance());
            }
        } catch(InstantiationException e) {}
        catch(IllegalAccessException e) {}
        HashMap h = new HashMap();
        for(int i = 0; i < petTypes.length; i++)
            h.put(petTypes(i).toString(),
                  new Counter());
        for(int i = 0; i < pets.size(); i++) {
            Object o = pets.get(i);
            if(o instanceof Pet)
                ((Counter)h.get(
                    "class c11.petcount2.Pet")).i++;
            if(o instanceof Dog)
                ((Counter)h.get(
                    "class c11.petcount2.Dog")).i++;
            if(o instanceof Pug)
                ((Counter)h.get(
                    "class c11.petcount2.Pug")).i++;
            if(o instanceof Cat)
                ((Counter)h.get(
                    "class c11.petcount2.Cat")).i++;
            if(o instanceof Rodent)
                ((Counter)h.get(
                    "class c11.petcount2.Rodent")).i++;
            if(o instanceof Gerbil)
                ((Counter)h.get(
                    "class c11.petcount2.Gerbil")).i++;
            if(o instanceof Hamster)
                ((Counter)h.get(
                    "class c11.petcount2.Hamster")).i++;
        }
        for(int i = 0; i < pets.size(); i++)
            System.out.println(
                pets.get(i).getClass().toString());
    }
}

```

```

Iterator keys = h.keySet().iterator();
while(keys.hasNext()) {
    String nm = (String)keys.next();
    Counter cnt = (Counter)h.get(nm);
    System.out.println(
        nm.substring(nm.lastIndexOf('.') + 1) +
        " quantity: " + cnt.i);
}
}
} //:~

```

Масивът **typenames** е махнат понеже имената ще се вземат като стрингове от обект **Class**. Забележете допълнителната работа за това: името на класа не е, например, **Gerbil**, а е **c11.petcount2.Gerbil** понеже се включва името на пакета. Забележете също че системата може да прави разлика между класове и интерфейси.

Може също да видите че създаването на **petTypes** не е необходимо да бъде в **try** блок понеже се изчислява по време на компилация и няма да изхвърли изключение, за разлика от **Class.forName()**.

Когато **Pet** са създадени динамично, може да видите че генераторът на случаини числа е ограничен между 1 и **petTypes.length** и не включва нула. Така е защото нулата се отнася за **Pet.class** и по предположение родовия **Pet** обект не е интересен. Понеже **Pet.class** е част от **petTypes** резултатът е че висчките "петс" се броят.

Динамично instanceof

Java 1.1 е добавил **isInstance** метод в класа **Class**. Това позволява динамично да се вика **instanceof** оператора, което може да се прави само статично в Java 1.0 (както беше показано). Така всички онези досадни **instanceof** оператори могат да се махнат в **PetCount** примера:

```

//: c11:petcount3:PetCount3.java
// Using Java 1.1 isInstance()
package c11.petcount3;
import java.util.*;

class Pet {}
class Dog extends Pet {}
class Pug extends Dog {}
class Cat extends Pet {}
class Rodent extends Pet {}
class Gerbil extends Rodent {}
class Hamster extends Rodent {}

class Counter { int i; }

public class PetCount3 {
    public static void main(String[] args) {
        ArrayList pets = new ArrayList();
        Class[] petTypes = {
            Pet.class,
            Dog.class,
            Pug.class,
            Cat.class,
            Rodent.class,
            Gerbil.class,

```

```

Hamster.class,
};

try {
for(int i = 0; i < 15; i++) {
    // Offset by one to eliminate Pet.class:
    int rnd = 1 + (int)(Math.random() * (petTypes.length - 1));
    pets.add(
        petTypes(rnd).newInstance());
}

} catch(InstantiationException e) {}
catch(IllegalAccessException e) {}

HashMap h = new HashMap();
for(int i = 0; i < petTypes.length; i++)
    h.put(petTypes(i).toString(),
        new Counter());

for(int i = 0; i < pets.size(); i++) {
    Object o = pets.get(i);
    // Using isInstance to eliminate individual
    // instanceof expressions:
    for (int j = 0; j < petTypes.length; ++j)
        if (petTypes(j).isInstance(o)) {
            String key = petTypes(j).toString();
            ((Counter)h.get(key)).i++;
        }
}

for(int i = 0; i < pets.size(); i++)
    System.out.println(
        pets.get(i).getClass().toString());
Iterator keys = h.keySet().iterator();
while(keys.hasNext()) {
    String nm = (String)keys.next();
    Counter cnt = (Counter)h.get(nm);
    System.out.println(
        nm.substring(nm.lastIndexOf('.') + 1) +
        " quantity: " + cnt.i);
}
}
}
} ///:~

```

Може да видите че методът **isInstance()** в Java 1.1 е елиминиран нуждата от **instanceof** изрази. Още това значи че може да добавяте нови типове "петс" просто чрез промяна на **petTypes** масива; Останалата част от програмата не иска промени (каквото бяха необходими с **instanceof** изразите).

Синтаксис на RTTI

Java прави своето RTTI чрез обекта **Class**, даже ако правите нещо от рода на каст. Класът **Class** също има някои пътища за използване на RTTI.

Първо, трябва да се сдобиете с манипулатор към съответния **Class** обект. Единият начин за това, както беше показано в предишния пример, е да се използва стринг и **Class.forName()** метода. Това е удобно понеже не е необходим обект от дадения тип за да се получи **Class** манипулатор. Обаче ако вече има обект от типа от който се интересувате, може да вземете **Class** манипулатора чрез викане на метод който е част от **Object** кореновия клас: **getClass()**.

Това връща **Class** манипулятор представляващ фактическия тип на обекта. **Class** има някои интересни и понякога полезни методи, демонстрирани в следния пример:

```
//: c11:ToyTest.java
// Testing class Class

interface HasBatteries {}
interface Waterproof {}
interface ShootsThings {}

class Toy {
    // Comment out the following default
    // constructor to see
    // NoSuchMethodError from (*1*)
    Toy() {}
    Toy(int i) {}
}

class FancyToy extends Toy
    implements HasBatteries,
               Waterproof, ShootsThings {
    FancyToy() { super(1); }
}

public class ToyTest {
    public static void main(String[] args) {
        Class c = null;
        try {
            c = Class.forName("FancyToy");
        } catch(ClassNotFoundException e) {}
        printInfo(c);
        Class[] faces = c.getInterfaces();
        for(int i = 0; i < faces.length; i++)
            printInfo(faces(i));
        Class cy = c.getSuperclass();
        Object o = null;
        try {
            // Requires default constructor:
            o = cy.newInstance(); // (*1*)
        } catch(InstantiationException e) {}
        catch(IllegalAccessException e) {}
        printInfo(o.getClass());
    }
    static void printInfo(Class cc) {
        System.out.println(
            "Class name: " + cc.getName() +
            " is interface? (" +
            cc.isInterface() + ")");
    }
} //:~
```

Може да видите че **class FancyToy** е доста сложен, като наследява от **Toy** и **implements interface**-и на **HasBatteries**, **Waterproof** и **ShootsThings**. В **main()** се създава манипулятор на **Class** и се инициализира на **FancyToy Class** чрез **forName()** вътре в подходящ **try** блок.

Методът **Class.getInterfaces()** връща масив от **Class** обекти представлящи интерфейсите които се съдържат в **Class** обекта от който се интересуваме.

Ако имате **Class** обект може също да го питате за директния му базов клас чрез **getSuperclass()**. Това, разбира се, връща **Class** манипулатор който може да изследвате по-нататък. Това значи че, по време на изпълнение, може да получите цялата йерархия на класа.

Методът **newInstance()** на **Class** може, отначало, да изглежда просто друг начин да **clone()** обект. Обаче може да създадете нов обект чрез **newInstance()** без съществуващ обект, както се вижда тук, понеже няма обект **Toy**, само **су**, което е манипулатор към **Class** обекта на **y**. Това е начин да се реализира “виртуален конструктор,” който позволява да се каже “Не знам точно какъв тип си, но вземи че се създай правилно.” В примера по-горе, **су** е точно **Class** манипулатор без никаква допълнителна информация известна по време на компилиация. И когато създавате нов екземпляр, получавате **Object** манипулатор. Но този манипулатор сочи **Toy** обект. Разбира се, преди да може да изпращате съобщения различни от тези за **Object**, ще трябва да го изследвате малко и да направите някакъв кастинг. Освен това класът кийто се създава с **newInstance()** трябва да има конструктор по подразбиране. Няма начин да се използва **newInstance()** за създаване на обекти без конструктор по подразбиране, така че това може да бъде малко ограничаващо в Java 1. Обаче API на рефлексията в Java 1.1 (обсъждан в следващата секция) позволява динамично да се използва всякаакъв конструктор в един клас.

Последния метод в листинга е **printInfo()**, който взема **Class** манипулатор и името с **getName()**, после гледа дали е интерфейс с **isInterface()**.

Изходът от тази програма е:

```
Class name: FancyToy is interface? (false)
Class name: HasBatteries is interface? (true)
Class name: Waterproof is interface? (true)
Class name: ShootsThings is interface? (true)
Class name: Toy is interface? (false)
```

Така, с **Class** обекта може да намерите всичко което искате да знаете за обект.

Рефлексия: информация за клас по време на изпълнение

Ако не знаете точния тип на обект, RTTI ще ви го каже. Обаче има ограничение: типът трябва да е известен по време на компилиация за да може да използвате RTTI и да правите нещо полезно с информацията. С други думи казано, компилаторът трябва да знае всички класове с които работите за RTTI.

Отначало това не изглежда силно ограничение, но да предположим че имате манипулатор кийто е извън вашето програмно пространство. Фактически класът на този обект е недостъпен за вашата програма даже по време на компилиация. Да предположим че сте взели байтове от дисков файл или мрежова връзка и знаете че те представляват клас. Понеже компилаторът не знае за класа когато компилира кода, как би могъл да използува този клас?

В традиционните програмни среди това изглежда като изсмукан от пръсти сценарий. Но ако преминем към по-голям програмен свят виждаме важни случаи когато това се случва. Първият е компонентно-базираното програмиране където строите проекти чрез *Rapid Application Development* (RAD) в инструмент за производство на приложения. Това е визуалният подход към правене на програми (които виждате на екрана като форми) чрез местене на икони които представлят компоненти във формата. Тези компоненти после се конфигурират чрез поставяне на някои стойности по време на компилиация. Тази конфигурация по време на проектирането изиска от всеки компонент да може да се прави екземпляр и той да показва част от себе си и да позволява стойностите му да бъдат четени и поставяни. Освен това компонентите които

поддържат GUI събитията трябва да показват информация за методите така че RAD средата да може да асистира на програмиста в подтискането на тези методи за обработка на събитията. Рефлексията дава механизъм за откриване на достъпните методи и имената им. Java 1.1 дава начин за компонентно програмиране посредством Java Beans (описани в глава 13).

Друга примамлива мотивация за намиране на информация по време на изпълнение е възможността да се създават и активират обекти отдалечно и много платформено през мрежа. Това е наречено *Remote Method Invocation* (RMI) и позволява на Java програма (версия 1.1 и по-висока) да има разпределени по много машини обекти. Това разпределение може да се появи по много причини: може би правите задача с интензивни изчисления и искате да я накъсате и да дадете някои парчета на машини които циклят напразно за да ускорите нещата. В някои ситуации може да искате кода който обработва някои конкретни типове задачи (напр. "Бизнес Правила" в многоредична клиент/сървър архитектура) да е на определена машина така че тя да стане общо хранилище за тези данни та да може да се прamenят лесно и да важат за всички потребители. (Това е интересно изследване понеже машина съществува само за да прови промените лесни!) Паралелно с всичко това, разпределената обработка също поддържа специализиран хардуер който може да е добър за отделна задача – обръщане на матрици, например – но неподходящ и твърде скъп за програмни системи с общо предназначение.

В Java 1.1 класът **Class** (описан по-рано в тази глава) е разширен за да поддържа концепцията за рефлексия, та има допълнителна библиотека, **java.lang.reflect**, с класовете **Field**, **Method** и **Constructor** (всеки от които прилага **Member interface**). Обекти от тези типове се създават от JVM по време на изпълнение за да представят членовете на непознат клас. После може да използвате **Constructor**ите за създаване на нови обекти, методите **get()** и **set()** за четене и промяна на полетата асоциирани с **Field** обекти, метод **invoke()** за викане на метод асоцииран с **Method** обект. Освен това може да викате методите за удобство **getFields()**, **getMethods()**, **getConstructors()** и т.н., за да върнат масиви от обекти представлящи полета, методи и конструктори. (Може да намерите повече чрез разглеждане на **Class** във вашата онлайн документация.) Така информацията за класа за анонимен обект може напълно да се извлече по време на изпълнение, не е необходимо нищо да се знае по време на компилация.

Важно е да се разбере че няма нищо магическо в рефлексията. Когато използвате рефлексията за работа с обект от неизвестен тип JVM просто ще погледне обекта и ще види дали принадлежи към определен клас (точно както обикновения RTTI) но след това, преди да може да направи каквото и да е друго нещо, **Class** обектът трябва да бъде натоварен. Така **.class** файлът за конкретния тип трябва все пак да бъде достъпен за JVM, било на локалната машина или през мрежата. Така че истинската разлика между RTTI и рефлексията е че с RTTI компилаторът отваря и разглежда **.class** файла по време на компилация. С други думи, може да викате всички методи на обект по "нормален" начин. С рефлексията **.class** файлът е недостъпен по време на компилация; той се отваря и използва от програмната среда по време на изпълнение.

Екстрактор на методите

Рядко ще става нужда да се използват инструментите на рефлексията; те са в езика за поддръжка на други черти на Java като сериализацията на обекти (описана в глава 10), Java Beans и RMI (описано по-късно в тази книга). Има обаче случаи когато възможността динамично да се извлича информация за клас е твърде полезна. Един извънредно полезен инструмент е екстракторът на информация за методите. Както се спомена по-горе, гледането дефиницията на клас в сорса позволява да се видят само дефинираните или подписаните методи **само в тази дефиниция**. Но може да има дузини други които са дошли чрез базовите класове. Да се намерят дефинициите им е както досадно, така и отнемащо много време. За

щастие рефлексията дава начин да се напише прост инструмент който автоматично ще показва целия интерфейс. Ето как работи той:

```
//: c11>ShowMethods.java
// Using Java 1.1 reflection to show all the
// methods of a class, even if the methods are
// defined in the base class.
import java.lang.reflect.*;

public class ShowMethods {
    static final String usage =
        "usage: \n" +
        "ShowMethods qualified.class.name\n" +
        "To show all methods in class or: \n" +
        "ShowMethods qualified.class.name word\n" +
        "To search for methods involving 'word'";
    public static void main(String[] args) {
        if(args.length < 1) {
            System.out.println(usage);
            System.exit(0);
        }
        try {
            Class c = Class.forName(args[0]);
            Method[] m = c.getMethods();
            Constructor[] ctor = c.getConstructors();
            if(args.length == 1) {
                for (int i = 0; i < m.length; i++)
                    System.out.println(m[i].toString());
                for (int i = 0; i < ctor.length; i++)
                    System.out.println(ctor[i].toString());
            }
            else {
                for (int i = 0; i < m.length; i++)
                    if(m[i].toString()
                        .indexOf(args[1])!= -1)
                        System.out.println(m[i].toString());
                for (int i = 0; i < ctor.length; i++)
                    if(ctor[i].toString()
                        .indexOf(args[1])!= -1)
                        System.out.println(ctor[i].toString());
            }
        } catch (ClassNotFoundException e) {
            System.out.println("No such class: " + e);
        }
    }
} ///:~
```

Методите на **Class getMethods()** и **getConstructors()** връщат масив от **Method** и **Constructor**, респективно. Всеки от тези класове има методи за по-нататъшна дисекция на имена, аргументи и връщани стойности на методите, които те представят. Но може също просто да използвате **toString()**, както е направено тук, за да се получи **String** с цялата сигнатурата на метода. Тостаналата част от кода е за извличане на информация от командния ред, определяне дали дадена сигнатурата совпада с вашия стринг (чрез **indexOf()**) и извеждане на резултатите.

Това показва рефлексия в действие, понеже резултатът даван от **Class.forName()** не може да бъде известен по време на компилация и затова цялата информация за сигнатурите бива

извлечена по време на изпълнение. Ако изследвате вашата онлайн документация за рефлексията, вие ще видите че има достатъчна поддръжка за извикване на метод който е напълно непознат по време на компилация. Още веднъж, това е нещо което єдва ли ще искате да правите сами – поддръжката е налице заради Java и така програмната среда може да работи с Java Beans – но е интересно.

Интересен експеримент е да се пусне **java ShowMethods ShowMethods**. Това дава листинг който включва **public** конструктор по подразбиране, даже и да сте видели от сорса че не е дефиниран такъв. Конструкторът който виждате е този автоматично конструиран от компилатора. Ако после направите **ShowMethods** не-**public** клас (тоест, приятелски), конструкторът по подразбиране вече не се показва в изходния листинг. На синтезирания конструктор по подразбиране автоматично е даден достъп като на класа.

Изходът за **ShowMethods** е все още малко уморителен. Например има една част произведена чрез извикване на **java ShowMethods java.lang.String**:

```
public boolean
    java.lang.String.startsWith(java.lang.String,int)
public boolean
    java.lang.String.startsWith(java.lang.String)
public boolean
    java.lang.String.endsWith(java.lang.String)
```

Би било даже по-добре квалификатори като **java.lang** да можеха да се изрязват. Класът **StreamTokenizer** въведен в предишната глава може да помогне да се реши този проблем:

```
//: c11>ShowMethodsClean.java
// ShowMethods with the qualifiers stripped
// to make the results easier to read
import java.lang.reflect.*;
import java.io.*;

public class ShowMethodsClean {
    static final String usage =
        "usage: \n" +
        "ShowMethodsClean qualified.class.name\n" +
        "To show all methods in class or: \n" +
        "ShowMethodsClean qualif.class.name word\n" +
        "To search for methods involving 'word'";
    public static void main(String[] args) {
        if(args.length < 1) {
            System.out.println(usage);
            System.exit(0);
        }
        try {
            Class c = Class.forName(args[0]);
            Method[] m = c.getMethods();
            Constructor[] ctor = c.getConstructors();
            // Convert to an array of cleaned Strings:
            String[] n =
                new String(m.length + ctor.length);
            for(int i = 0; i < m.length; i++) {
                String s = m[i].toString();
                n[i] = StripQualifiers.strip(s);
            }
            for(int i = 0; i < ctor.length; i++) {
                String s = ctor[i].toString();
                n[i + m.length] =
```

```

        StripQualifiers.strip(s);
    }
    if(args.length == 1)
        for (int i = 0; i < n.length; i++)
            System.out.println(n(i));
    else
        for (int i = 0; i < n.length; i++)
            if(n(i).indexOf(args(1))!= -1)
                System.out.println(n(i));
} catch (ClassNotFoundException e) {
    System.out.println("No such class: " + e);
}
}

class StripQualifiers {
    private StreamTokenizer st;
    public StripQualifiers(String qualified) {
        st = new StreamTokenizer(
            new StringReader(qualified));
        st.ordinaryChar(' '); // Keep the spaces
    }
    public String getNext() {
        String s = null;
        try {
            if(st.nextToken() != StreamTokenizer.TT_EOF) {
                switch(st.ttype) {
                    case StreamTokenizer.TT_EOL:
                        s = null;
                        break;
                    case StreamTokenizer.TT_NUMBER:
                        s = Double.toString(st.nval);
                        break;
                    case StreamTokenizer.TT_WORD:
                        s = new String(st.sval);
                        break;
                    default: // single character in ttype
                        s = String.valueOf((char)st.ttype);
                }
            }
        } catch(IOException e) {
            System.out.println(e);
        }
        return s;
    }
    public static String strip(String qualified) {
        StripQualifiers sq =
            new StripQualifiers(qualified);
        String s = "", si;
        while((si = sq.getNext()) != null) {
            int lastDot = si.lastIndexOf('.');
            if(lastDot != -1)
                si = si.substring(lastDot + 1);
            s += si;
        }
        return s;
    }
}

```

```
| }  
| } //:/~
```

Класът **ShowMethodsClean** е доста приличен на **ShowMethods**, освен че взема масиви от **Method** и **Constructor** и ги превръща в единствен масив от **String**. Всеки от тези **String** обекти по-късно се дава на **StripQualifiers.Strip()** за махане на всичката квалификация на метода. Както може да видите, това използва **StreamTokenizer** и **String** манипулация за свършване на тази работа.

Този инструмент може реално да спести време докато програмирате, когато се интересувате от конкретен член и не щете да се разхождате в документацията на класовата иерархия, или ако не знаете дали класът може да прави нещо с, да кажем, **Color** обекти.

Глава 13 съдържа GUI версия на тази програма, така че може да я оставите пусната докато пишете програми, за бърз оглед.

Резюме

RTTI позволява да се получи информация за типа от анонимен манипулатор на базов клас. Така то е готово за неправилна употреба от новака понеже може да има смисъл преди полиморфните методи. За много хора с процедурна ориентация, it's difficult not to organize their programs into sets ое трудно да реорганизират програмите си в множества от **switch** оператори. Те биха направили това чрез RTTI и биха загубили по този начин важната черта полиморфизъм на кода и поддръжката. Намерението в Java е да се използват полиморфни извиквания на методи в кода, а да се използва RTTI само когато е необходимо.

Обаче използването на полиморфизма изисква сорса да е на разположение, понеже в някой момент се открива, че не разполагаме с някой необходим метод. Ако базовият клас идва от библиотека или въобще принадлежи на някой друг, решението на проблема е RTTI: Може да наследите нов тип и да добавите вашия допълнителен метод. На друго място в кода може да откриете точния тип и да извикате въпросния метод. Това не премахва полиморфизма или разширяемостта на кода на програмата понеже добавянето на нов тип няма да изисква издавянето на превключващи оператори в програмата. Обаче когато добавяте код в главното тяло на програмата който изисква вашата нова черта, трябва да използвате RTTI за откриване на точния тип.

Слагането на черта в базовия клас би могло да значи че, за употреба в някакъв специален слас, всички класове извлечени от въпросния клас трябва да имат някаква част от метода. Това прави интерфейса по-малко ясен и дразни с подтискането на абстрактните методи когато наследявате от този базов клас. Например да вземем класова иерархия представяща музикални инструменти. Да кажем че искате да настроите всички инструменти в оркестъра. Едната възможност е да използвате **ClearSpitValve()** метод в базовия клас **Instrument**, но това е смущаващо понеже предполага че **Percussion** и **Electronic** инструментите също се настройват с червячета. RTTI дава много по-смислено решение в този случай понеже можете да сложите метода в специфичен клас (**Wind** в този случай), където е подходящо. Обаче по-подходящо решение е да сложите **prepareInstrument()** метод в базовия клас, но може да не видите това когато се сблъскате с проблема и погрешно да мислите че трябва да се използва RTTI.

Накрая, RTTI понякога ще решава проблеми с ефективността. Ако вашият код използва масово полиморфизъм, но излезе че един от вашите обекти реагира на това с твърде ниска ефикасност, може да намерите типа с RTTI и да напишете код зависещ от случая за повишаване на ефективността.

Упражнения

1. Напишете код който взема обект и рекурсивно печата всички класове в йерархията му.
2. В **ToyTest.java**, изкоментирайте конструктора по подразбиране на **Toy** и обяснете какво се случва.
3. Създайте нов тип колекция която използва **ArrayList**. Хванете типа на първия вкаран обект, а после позволете на потребителя да вкарва обекти от само него тип от този момент нататък.
4. Напишете програма да определи дали масив от **char** е примитивен тип или същински обект.
5. Реализирайте **clearSpitValve()** както е описано в тази глава.
6. Реализирайте **rotate(Shape)** метода описан в тази глава, така че да проверява дали върти **Circle** (и, ако да, не изпълнява операцията).

12: Подаване и връщане на обекти

Вече трябва да се чувствате достатъчно удобно с мисълта че когато “предавате” обект фактически предавате манипулатор.

В много програмни езици, ако не във всички, може да се използва “обичаен” начин да се предават обекти и повечето време той работи добре. Но винаги, изглежда, идва момент когато се налага да правите нещо необично и нещата стават малко по-сложни (в случая на C++, доста сложни). Java не е изключение, важно е да знаете какво точно става при предаването на обекти. Тази глава ще хвърли светлина върху този въпрос.

Друг начин да се постави въпроса, ако извате от съответно снабден език, е “Има ли в Java указатели?” Твърди се че указателите са сложни и пр. И затова лоши, а понеже Java е изцяло доброта и светлина and и ще премахне вашите земни програмистки тегла, той вероятно не може да има такива неща. Обаче по-точно е да се каже че в Java има указатели; разбира се всеки идентификатор на обект в Java (освен за примитивите) е един от тези указатели, но използването им е ограничавано и контролирано не само от компилатора, но и от операционната среда по време на изпълнение. Или с други думи казано, Java има указатели, но не аритметика с указателите. Те са които аз нарекох “манипулатори,” вие можете да ги мислите като “безопасни указатели,” не много различно от безопасните ножици в основното училище - те не са остри и не може да се порежете без голямо усилие, но понякога могат да бъдат бавни и досадни.

Подаване на манипулатори

Когато подавате манипулатор на метод, още сочите към същия обект. Прост експеримент демонстрира това: (Виж стр. 63 ако имате проблеми с пускането на тази програма.)

```
//: c12:PassHandles.java
// Passing handles around
package c12;

public class PassHandles {
    static void f(PassHandles h) {
        System.out.println("h inside f(): " + h);
    }
    public static void main(String[] args) {
        PassHandles p = new PassHandles();
        System.out.println("p inside main(): " + p);
        f(p);
    }
} ///:~
```

Методът **toString()** е автоматично извикан в операторите за извеждане, а **PassHandles** наследява директно от **Object** без предефиниране на **toString()**. Така, версията на **toString()** от **Object** се използва, което извежда класа на обекта следван от адреса на който се

намира обекта (не манипулатора, а фактическия адрес в паметта). Изходът изглежда като този:

```
| p inside main(): PassHandles@1653748  
| h inside f(): PassHandles@1653748
```

Може да видите че както **p** така и **h** се отнасят за един и същ обект. Това е много по-ефективно от дублицирането на **PassHandles** обекта така че да можете да подадете аргумент на метод. Но това поражда важен въпрос.

Псевдоними

Псевдонимите означават, че повече от един указател сочи към един обект, както в горния пример. Проблемът с псевдонимите се случва когато някой пише в него обект. Ако собствениците на другите манипулатори към обекта не очакват той да се променя, те ще бъдат изненадани. Това може да бъде демонстрирано с прост пример:

```
//: c12:Alias1.java  
// Aliasing two handles to one object  
  
public class Alias1 {  
    int i;  
    Alias1(int ii) { i = ii; }  
    public static void main(String[] args) {  
        Alias1 x = new Alias1(7);  
        Alias1 y = x; // Assign the handle  
        System.out.println("x: " + x.i);  
        System.out.println("y: " + y.i);  
        System.out.println("Incrementing x");  
        x.i++;  
        System.out.println("x: " + x.i);  
        System.out.println("y: " + y.i);  
    }  
} ///:~
```

В реда:

```
| Alias1 y = x; // Assign the handle
```

нов **Alias1** манипулатор се създава, но вместо да се насочи към пресния обект създаден с **new**, той се приравнява на съществуващия манипулатор. Така че съдържанието на манипулатора **x**, който е адрес на обекта **x** към който сочи, е приравнен на **y**, а с това и **x** и **y** сочат един и същ обект. Така че когато **i**-то на **x** се увеличава в оператора:

```
| x.i++;
```

i-то на **y** също ще бъде засегнато. Това може да бъде видяно в изхода:

```
x: 7  
y: 7  
Incrementing x  
x: 8  
y: 8
```

Едно добро решение на случая е просто той да не се състои: не правете алиасинг съзнателно в един обхват. Вашият код ще бъде по-лесен за четене и тестване. Обаче когато давате манипулатор като аргумент – което е начина по който се предполага да се работи в Java – понеже манипулаторът който автоматично се създава локално може да модифицира “външния обект” (обектът който е създаден извън обхвата на метода). Ето пример:

```
//: c12:Alias2.java
// Method calls implicitly alias their
// arguments.

public class Alias2 {
    int i;
    Alias2(int ii) { i = ii; }
    static void f(Alias2 handle) {
        handle.i++;
    }
    public static void main(String[] args) {
        Alias2 x = new Alias2(7);
        System.out.println("x: " + x.i);
        System.out.println("Calling f(x)");
        f(x);
        System.out.println("x: " + x.i);
    }
} ///:~
```

Изходът е:

```
x: 7
Calling f(x)
x: 8
```

Методът променя аргумента си, външния обект. Когато се случи нещо от този род, вие трябва да решите смислено ли е, дали потребителят го обаква и дали ще се създават проблеми.

Изобщо, методи се викат за да върнат стойност и/или промяна на състоянието на обекта за който методът е извикан. (Методът е как „пращате съобщение“ към него обект.) Много по-малко разпространено е да се вика метод заради промяна на аргументите му; това се нарича „викане на метод заради странични ефекти.“ Така, ако го правите, потребителят трябва да е добре инструктирани предупреден за възможни изненади. Поради смущението и капаните, много по-добре е да избегвате страничните ефекти.

Ако искате да промените аргумент по време на работа на метод и не възнамерявате да променяте външния аргумент, ще предпазите последния чрез създаване на копие в метода. Това е тема на по-голямата част от тази глава.

Създаване на локални копия

Да припомним: всички аргументи в Java се подават чрез подаване на манипулатори. Тоест, когато давате „обект,” в действителност давате само манипулятор към обект който живее извън метода, така че ако направите промени с него манипулятор, променяте външния обект. Освен това:

- ◆ Автоматично се създават псевдоними при предаването на аргументите.
- ◆ Няма локални обекти, а само локални манипулатори.
- ◆ Манипуляторите имат обхвати, обектите нямат.
- ◆ В Java не стои въпросът с времето на живот на обектите.
- ◆ Няма поддръжка от езика (напр. като `const`) за предотвратяване модифицирането на обект (за предпазване от отрицателните ефекти на псевдонимите).

Ако само четете информация от обект и не го променяте, даването на манипулятор е най-ефикасната форма на подаване на аргументи. Това е добре; най-ефикасният начин да се

направят нещата е и начин по подразбиране. Обаче понякога става нужда да се третират обекти както ако бяха “локални” така че промените да засягат само локалното копие и да не засягат външния обект. Много програмни езици поддържат възможността за правене на локално копие на външния обект вътре в метода.¹ Java не прави така, но дава възможност да се постигне такъв ефект.

Предаване по стойност

Това извежда на преден план терминологичен въпрос, който винаги изглежда добре за аргумент. Терминът е “предаване по стойност,” а значението зависи от това как схващате работата на програмата. Общото значение е че вземате локално копие от това което подавате, но реалният въпрос е какво мислите за това, което подавате. Като дойде ред на значението “предаване по стойност,” има две доволно различни становища:

1. Java дава всичко по стойност. Когато давате примитиви на метод, имате различно копие на примитива. Когато давате манипулятор на метод, имате копие на манипулятора. Ergo, всичко се предава по стойност. Разбира се, предположението е че винаги мислите (и правите) да се предават манипулятори, но изглежда като дизайнът на Java е изминал голям път нанатък позволявайки ви (товечето време) да работите с манипулятор. Тоест, изглежда да мислите за манипулятора като за “обекта,” понеже всяко извикване на метод работи с обекта.
2. Java подава примитивите по стойност (няма аргумент), но обектите се предават с позоваване. Това е световната координатна система където манипуляторите са псевдоними на обектите, така че не мислите за подаване на манипулятори, ами казвате “Аз предавам обект.” Доколкото не се работи с локално копие на обекта като го давате за аргумент, обектите явно не се предават по стойност. Види се има поддръжка за тази гледна точка от Sun, понеже една от “резервираните но не реализирани” ключови думи е **byvalue**. (Никой не знае, обаче, дали тя ще види бялелия свят някой ден.)

Като сме огледали добре и двете и сме казали “Зависи от това как гледате на манипулятора,” ще се опитам да проследя въпроса в останалата част на главата. В края на краишата това не е толкова важно – важното е да разберете, че подаването на манипулятор може да промени извикващия обект неочаквано.

Клониране на обекти

Най-вероятната причина да правите локално копие е ако искате да променяте копието и не искате да се промения извикващия обект. Ако пожелаете да правите локално копие, просто викате метода **clone()** да направи това. Този метод е дефиниран като **protected** в базовия клас **Object** и трябва да го подтиснете като **public** в който и да е извлечен клас който искате да може да се клонира. Например стандартния библиотечен клас **ArrayList** поддържа **clone()**, така че може да извикаме **clone()** за **ArrayList**:

```
//: c12:Cloning.java
// The clone() operation works for only a few
// items in the standard Java library.
import java.util.*;

class Int {
    private int i;
    public Int(int ii) { i = ii; }
    public void increment() { i++; }
```

¹ In C, which generally handles small bits of data, the default is pass-by-value. C++ had to follow this form, but with objects pass-by-value isn't usually the most efficient way. In addition, coding classes to support pass-by-value in C++ is a big headache.

```

public String toString() {
    return Integer.toString(i);
}
}

public class Cloning {
    public static void main(String[] args) {
        ArrayList v = new ArrayList();
        for(int i = 0; i < 10; i++)
            v.add(new Int(i));
        System.out.println("v: " + v);
        ArrayList v2 = (ArrayList)v.clone();
        // Increment all v2's elements:
        for(Iterator e = v2.iterator();
            e.hasNext();)
            ((Int)e.next()).increment();
        // See if it changed v's elements:
        System.out.println("v: " + v);
    }
} ///:~

```

Методът **clone()** прави **Object**, който после трябва да бъде преобразуван в правилния тип. Този пример показва как методът **clone()** на **ArrayList** не се опитва автоматично да клонира всеки съдържан в **ArrayList** обект – стария **ArrayList** и клонинга **ArrayList** са псевдоними на един и същ обект. Това често се нарича **плитко копиране**, понеже се копира само “повърхностната” порция на обект. Фактически обектът се състои от тази “повърхност” плюс обектите към които сочат манипулаторите в обекта, плюс всички обекти към които тези обекти сочат и т.н. Това често се споменава като “паяжина от обекти.” Копирането на цялата суматоха се нарича **дълбоко копиране**.

Може да видите ефекта от плиткото копиране в изхода, където операциите изпълнени над **v2** засягат **v**:

```

v: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
v: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

```

Да не се прилага **clone()** към обектите съдържани в **ArrayList** е вероятно добро предположение, понеже не е сигурно, че всички те са клонирани.²

Правене на клас клонирам

Макар че клониращия метод да принадлежи към базовия за всички клас **Object**, клонирането не е автоматично налице за всеки клас.³ Това може да изглежда антиинтуитивно на идеята че всички наследени методи са достъпни в извлечения клас. Клонирането в Java върви срещу

² This is not the dictionary spelling of the word, but it's what is used in the Java library, so I've used it here, too, in some hopes of reducing confusion.

³ You can apparently create a simple counter-example to this statement, like this:

```

public class Cloneit implements Cloneable {
    public static void main (String[] args)
        throws CloneNotSupportedException {
        Cloneit a = new Cloneit();
        Cloneit b = (Cloneit)a.clone();
    }
}

```

However, this only works because **main()** is a method of **Cloneit** and thus has permission to call the **protected** base-class method **clone()**. If you call it from a different class, it won't compile.

тази идея; ако искате да се клонира в клас, трябва да добавите специално код за да стане това.

Използване на трик с **protected**

За да се предотврати клонируемостта по подразбиране във всеки клас, методът **clone()** е **protected** в базовия клас **Object**. Това значи не само недостъпност за клиент-програмиста който просто използва класа (без да наследява от него), но също и че не може да се вика **clone()** чрез манипулятор към базовия клас. (Макар и това да би могло да бъде полезно в някои ситуации, като тази да се клонира куп **Objects** полиморфно.) Това е ефективно начин да се даде, по време на компилация, информация че вашият обект не е клонирам – и достатъчно неприятно многото класове в стандартната Java библиотека не са клонирани. Така, ако напишете:

```
Integer x = new Integer(1);
x = x.clone();
```

ще получите, по време на компилация, съобщение за грешка, казващо че **clone()** не е достъпен (понеже **Integer** не го поддържа и се вика този по подразбиране – **protected** версията).

Ако, обаче, сте в клас извлечен от **Object** (както всички класове са), имате позволението да викате **Object.clone()** понеже той е **protected** и вие сте в наследника. Базовият клас **clone()** има полезна функционалност – прави фактическо побитово копие на обекта на извлечения клас, съгласно с общия смисъл на операцията клониране. Обаче тогава трябва да направите вашата клонираща операция **public** за да бъде достъпна. Така че двата ключови въпроса при клонирането са: практически винаги се вика **super.clone()** и се прави **public**.

Вероятно бихте искали да поддържате **clone()** в по-нататък извлечени класове, иначе вашият (сега **public**) **clone()** ще се използва, а това може да не е точното нещо (макар и, понеже **Object.clone()** прави копие на фактическия обект, боже и да бъде). Трикът с **protected** работи еднократно, първият път когато наследите от клас който не е клонирам и го направите клонирам. Във всички класове наследени от вашия клас методът **clone()** е достъпен понеже е невъзможно в Java да се намали достъпът на метод при наследяване. Тоест, щом класът е клонирам, всичко извлечено от него е клонирамо, ако не използвате механизма (описан покъсно) за "изключване" на клонирането.

Прилагане на интерфейса **Cloneable**

Има още едно нещо за да се осигури клонируемостта на обект: да се приложи **Cloneable interface**. Този **interface** е малко странен понеже е празен!

```
interface Cloneable {}
```

Причината за прилагане на този празен **interface** очевидно не е че възнамерявате да правите ъпкаст към **Cloneable** и да викате някой от неговите методи. Използването на **interface** тук е малко като "hack" понеже се използва не по предназначение. Прилагането на **Cloneable interface** работи като флаг, вграден в типа на класа.

Има две причини за съществуването на **Cloneable interface**. Първо, бихте могли да имате ъпкастнат манипулятор към базовия клас и да не можете да определите обектът клонирам ли е или не. В този случай можете да използвате ключовата дума **instanceof** (описана в глава 11) за да намерите дали манипуляторът сочи към обект който може да бъде клониран:

```
if(myHandle instanceof Cloneable) // ...
```

Втората причина е допускането, че вероятно не бихте искали всички обекти да бъдат клонирами. Така **Object.clone()** потвърждава дали класът прилага **Cloneable** интерфейс.

Ако не, той изхвърля **CloneNotSupportedException** изключение. Изобщо, принудени сте да implement **Cloneable** като част от поддръжката на клониране (в езика).

Успешно клониране

След като сте разбрали подробностите около използването на метода **clone()**, вие сте в състояние лесно да правите класове, които да могат да се дублицират локално:

```
//: c12:LocalCopy.java
// Creating local copies with clone()
import java.util.*;

class MyObject implements Cloneable {
    int i;
    MyObject(int ii) { i = ii; }
    public Object clone() {
        Object o = null;
        try {
            o = super.clone();
        } catch (CloneNotSupportedException e) {
            System.out.println("MyObject can't clone");
        }
        return o;
    }
    public String toString() {
        return Integer.toString(i);
    }
}

public class LocalCopy {
    static MyObject g(MyObject v) {
        // Passing a handle, modifies outside object:
        v.i++;
        return v;
    }
    static MyObject f(MyObject v) {
        v = (MyObject)v.clone(); // Local copy
        v.i++;
        return v;
    }
    public static void main(String[] args) {
        MyObject a = new MyObject(11);
        MyObject b = g(a);
        // Testing handle equivalence,
        // not object equivalence:
        if(a == b)
            System.out.println("a == b");
        else
            System.out.println("a != b");
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        MyObject c = new MyObject(47);
        MyObject d = f(c);
        if(c == d)
            System.out.println("c == d");
        else
            System.out.println("c != d");
    }
}
```

```

        System.out.println("c != d");
        System.out.println("c = " + c);
        System.out.println("d = " + d);
    }
} //:~

```

Преди всичко, **clone()** трябва да бъде достъпен така че трябва да се направи **public**. Второ, в началото на вашата **clone()** ще викате метода **clone()** на базовия клас. Този **clone()** който се вика тук е онзи предварително дефиниран в **Object**, а вие можете да го извикате понеже е **protected** и с това достъпен за извлечениите класове.

Object.clone() намира колко е голям обекта, алокира достатъчно памет за нов екземпляр и копира всички битове от стария в новия. Това се нарича **побитово копиране** и типично е каквото се очаква да прави един **clone()** метод. Но преди **Object.clone()** да изпълни действията, той проверява дали **Cloneable**, тоест, дали прилага интерфейса **Cloneable**. Ако не, **Object.clone()** изхвърля **CloneNotSupportedException** за да индицира че не може да клонирате обекта. Така че трябва да сложите вашия **super.clone()** в трай-блок, за да хванете изключение, което никога няма да се случи (понеже сте приложили **Cloneable** интерфейс).

В **LocalCopy** двата метода **g()** и **f()** демонстрират разликата между двата подхода към подаването на аргументи. **g()** показва подаването по указател и модифицирането на външния обект и връща указател към него, докато **f()** клонира аргумента, по този начин разделяйки го от оригиналния обект. После може да прави каквото си иска, даже да върне манипулятор към този нов обект без да засегне външния изобщо. Забележете следния някакси любопитно изглеждащ оператор:

```
v = (MyObject)v.clone();
```

Това е където е създадено локалното копие. За да се предотврати смущението от такъв оператор, напомняме че такъв странен програмен идиом е перфектен в Java понеже всичко което има име е фактически манипулятор. Така че манипуляторът **v** се използва за **clone()** на копие от това, към което е насочен, а се връща манипулятор към базовия обект **Object** (понеже е дефинирано по този начин в **Object.clone()**) което после трябва да се преобразува в правилния тип.

В **main()** се тества разликата между двата подхода в двата аргумента. Изходът е:

```

a == b
a = 12
b = 12
c != d
c = 47
d = 48

```

Важно е да се отбележи, че тестването за равенство в Java не гледа съдържанието на сравняваните обекти. Операторите **==** и **!=** просто сравняват съдържанието на манипуляторите. Ако адресите в манипуляторите са едни и същи, манипуляторите сочат един и същ обект и затова са "равни." Така че операциите фактически проверяват дали двата манипулятора са псевдоними на един и същи обект!

Ефектът от **Object.clone()**

Какво всъщност става когато се вика **Object.clone()** та е толкова важно да се извика **super.clone()** когато подтискате **clone()** във ваш клас? Методът **clone()** в кореновия клас е отговорен за алокирането на паметта и побитовото копиране на обектите. Тоест той не просто отделя памет и копира **Object** – фактически се определя точната дължина на копирания обект, отделя се памет и се копира. Понеже всичко това става чрез кода в метода **clone()** определен в кореновия клас (който няма представа какво ще се наследява от него),

може да познаете че процесът използва RTTI за определяне на фактическия обект който ще се клонира. По този начин методът **clone()** може да алокира точното количество памет и да извърши копирането.

Каквото и да правите, първо ще се вика **super.clone()** в процеса на клонирането. Това дава основа на цялата операция чрез създаването на точен дубликат. След тази точка може да направите каквото трябва за да завърши клонирането (което може и да не трябва да е точно копиране - б.пр.).

За да определите какви трябва да бъдат допълнителните операции, трябва да знаете какво точно прави **Object.clone()**. В частност, дали клонира назначението на всички манипулатори? Следния пример проверява това:

```
//: c12:Snake.java
// Tests cloning to see if destination of
// handles are also cloned.

public class Snake implements Cloneable {
    private Snake next;
    private char c;
    // Value of i == number of segments
    Snake(int i, char x) {
        c = x;
        if(--i > 0)
            next = new Snake(i, (char)(x + 1));
    }
    void increment() {
        c++;
        if(next != null)
            next.increment();
    }
    public String toString() {
        String s = ":" + c;
        if(next != null)
            s += next.toString();
        return s;
    }
    public Object clone() {
        Object o = null;
        try {
            o = super.clone();
        } catch (CloneNotSupportedException e) {}
        return o;
    }
    public static void main(String[] args) {
        Snake s = new Snake(5, 'a');
        System.out.println("s = " + s);
        Snake s2 = (Snake)s.clone();
        System.out.println("s2 = " + s2);
        s.increment();
        System.out.println(
            "after s.increment, s2 = " + s2);
    }
} ///:~
```

Snake (змията - б.пр.) е направена от сегменти, всеки от тип **Snake**. Така че това е еднострочно свързан списък. Сегментите се създават рекурсивно, декрементирайки първия аргумент на конструктора докато се достигне нула. За да се даде на всеки сегмент уникатен

белег, вторият аргумент, който е `char`, се инкрементира за всяко рекурсивно викане на конструктора.

Методът `increment()` рекурсивно инкрементира всеки белег, така че може да видите промените, а `toString()` рекурсивно извежда всеки белег. Изходът е:

```
s = :a:b:c:d:e  
s2 = :a:b:c:d:e  
after s.increment, s2 = :a:c:d:e:f
```

Това значи, че само първият сегмент е дублициран от `Object.clone()`, така че се прави плитко копие. Ако искате да се дублицира цялата змия – дълбоко копиране – трябва да извършите допълнителни операции във вашия подтиснат `clone()`.

Типично ще викате `super.clone()` във всеки клас извлечен от клониран клас за да се осигури че всички операции от базовия клас (включително `Object.clone()`) ще се извършат. Това е последвано от явно извикване на `clone()` за всеки манипулятор във вашия обект; иначе те ще станат псевдоними на тези в оригиналния обект. Това е аналогично на начина на викане на конструктори – конструктора на базовия клас първо, после следващият извлечен конструктори така нататък до крайния извлечен конструктор. Разликата е че `clone()` не е конструктор така че нищо не го кара да става автоматично. Трябва да го правите сами.

Клониране на композиран обект

Като се опитвате да правите дълбоко копиране на композиран обект срещате проблем. Трябва да предполагате че методът `clone()` в членовете-обекти на свой ред прави дълбоко копиране на техните манипулятори и така нататък. Това е само обещание. То ефективно значи че за да работи дълбокото копиране трябва или да притежавате всичкия сорс на всичките класове или най-малкото да имате достатъчно сведения за всеки клас за да знаете дали той ще направи дълбокото копиране коректно.

Този пример показва какво ще трябва да направите та дълбоко да копирате композиран обект:

```
//: c12:DeepCopy.java  
// Cloning a composed object  
  
class DepthReading implements Cloneable {  
    private double depth;  
    public DepthReading(double depth) {  
        this.depth = depth;  
    }  
    public Object clone() {  
        Object o = null;  
        try {  
            o = super.clone();  
        } catch (CloneNotSupportedException e) {  
            e.printStackTrace();  
        }  
        return o;  
    }  
}  
  
class TemperatureReading implements Cloneable {  
    private long time;  
    private double temperature;  
    public TemperatureReading(double temperature) {  
        time = System.currentTimeMillis();  
    }
```

```

        this.temperature = temperature;
    }
    public Object clone() {
        Object o = null;
        try {
            o = super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return o;
    }
}

class OceanReading implements Cloneable {
    private DepthReading depth;
    private TemperatureReading temperature;
    public OceanReading(double tdata, double ddata){
        temperature = new TemperatureReading(tdata);
        depth = new DepthReading(ddata);
    }
    public Object clone() {
        OceanReading o = null;
        try {
            o = (OceanReading)super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        // Must clone handles:
        o.depth = (DepthReading)o.depth.clone();
        o.temperature =
            (TemperatureReading)o.temperature.clone();
        return o; // Upcasts back to Object
    }
}

public class DeepCopy {
    public static void main(String[] args) {
        OceanReading reading =
            new OceanReading(33.9, 100.5);
        // Now clone it:
        OceanReading r =
            (OceanReading)reading.clone();
    }
} ///:~

```

DepthReading и **TemperatureReading** са доста подобни; и двата съдържат само примитиви. Затова методът **clone()** може да бъде доста прост: вика **super.clone()** и връща резултата. Забележете че кодът на **clone()** за двата класа е идентичен.

OceanReading е композиран от **DepthReading** и **TemperatureReading** обекти та, за да стане дълбоко копиране, неговият **clone()** трябва да клонира манипуляторите вътре в **OceanReading**. За да стане това резултатът от **super.clone()** трябва да бъде кастнат към **OceanReading** обект (та да може да имате достъп до манипуляторите **depth** и **temperature**).

Дълбоко копиране на **ArrayList**

Нека да се върнем към примера с **ArrayList** от по-рано в тази глава. Този път класът **Int2** е клонирам така че **ArrayList** може да бъде копиран дълбоко:

```
//: c12:AddingClone.java
// You must go through a few gyrations to
// add cloning to your own class.
import java.util.*;

class Int2 implements Cloneable {
    private int i;
    public Int2(int ii) { i = ii; }
    public void increment() { i++; }
    public String toString() {
        return Integer.toString(i);
    }
    public Object clone() {
        Object o = null;
        try {
            o = super.clone();
        } catch (CloneNotSupportedException e) {
            System.out.println("Int2 can't clone");
        }
        return o;
    }
}

// Once it's cloneable, inheritance
// doesn't remove cloneability:
class Int3 extends Int2 {
    private int j; // Automatically duplicated
    public Int3(int i) { super(i); }
}

public class AddingClone {
    public static void main(String[] args) {
        Int2 x = new Int2(10);
        Int2 x2 = (Int2)x.clone();
        x2.increment();
        System.out.println(
            "x = " + x + ", x2 = " + x2);
        // Anything inherited is also cloneable:
        Int3 x3 = new Int3(7);
        x3 = (Int3)x3.clone();

        ArrayList v = new ArrayList();
        for(int i = 0; i < 10; i++ )
            v.add(new Int2(i));
        System.out.println("v: " + v);
        ArrayList v2 = (ArrayList)v.clone();
        // Now clone each element:
        for(int i = 0; i < v.size(); i++)
            v2.set(i, ((Int2)v2.get(i)).clone());
        // Increment all v2's elements:
        for(Iterator e = v2.iterator();
            e.hasNext(); )
```

```

        ((Int2)e.next()).increment();
        // See if it changed v's elements:
        System.out.println("v: " + v);
        System.out.println("v2: " + v2);
    }
} //:~

```

Int3 е наследен от **Int2** и нов примитивен член **int j** е добавен. Може да си помислите че ще трябва да подтиснете **clone()** пак за да осигурите копирането на **j**, но не е така. Когато **clone()** на **Int2** се вика като **clone()** на **Int3**, той вика **Object.clone()**, който определя че работи с **Int3** и дублицира всичките битове в **Int3**. Доколкото не добавяте манипулатори които трябва да се клонират, едно извикване на **Object.clone()** прави цялото необходимо дублициране, без значение колко дълбоко в йерархията е дефиниран **clone()**.

Може да видите какво е необходимо за дълбоко копиране на **ArrayList**: След като е клониран **ArrayList** трябва да преминете по всички сочени от **ArrayList** обекти и да ги клонирате. Нещо подобно трябва да направите за дълбоко копиране на **HashMap**.

Останалата част от примера показва че е станало клонирането понеже, веднъж като е клониран обектът, може да го променяте и оригиналът да остане незасегнат.

Дълбоко копиране чрез сериализация

Със сериализацията на обекти в Java 1.1 (въведена в глава 10) може да сте забелязали че, сериализиран и после десериализиран обект е, фактически, клониран.

Така че защо да не се използва сериализацията за постигане на дълбоко копиране? Ето пример който сравнява двата подхода чрез измерване на времето за изпълнение:

```

//: c12:Compete.java
import java.io.*;

class Thing1 implements Serializable {}
class Thing2 implements Serializable {
    Thing1 o1 = new Thing1();
}

class Thing3 implements Cloneable {
    public Object clone() {
        Object o = null;
        try {
            o = super.clone();
        } catch (CloneNotSupportedException e) {
            System.out.println("Thing3 can't clone");
        }
        return o;
    }
}

class Thing4 implements Cloneable {
    Thing3 o3 = new Thing3();
    public Object clone() {
        Thing4 o = null;
        try {
            o = (Thing4)super.clone();
        } catch (CloneNotSupportedException e) {
            System.out.println("Thing4 can't clone");
        }
        return o;
    }
}

```

```

        }
        // Clone the field, too:
        o.o3 = (Thing3)o3.clone();
        return o;
    }
}

public class Compete {
    static final int SIZE = 5000;
    public static void main(String[] args) {
        Thing2[] a = new Thing2[SIZE];
        for(int i = 0; i < a.length; i++)
            a[i] = new Thing2();
        Thing4[] b = new Thing4[SIZE];
        for(int i = 0; i < b.length; i++)
            b[i] = new Thing4();
        try {
            long t1 = System.currentTimeMillis();
            ByteArrayOutputStream buf =
                new ByteArrayOutputStream();
            ObjectOutputStream o =
                new ObjectOutputStream(buf);
            for(int i = 0; i < a.length; i++)
                o.writeObject(a[i]);
            // Now get copies:
            ObjectInputStream in =
                new ObjectInputStream(
                    new ByteArrayInputStream(
                        buf.toByteArray()));
            Thing2[] c = new Thing2[SIZE];
            for(int i = 0; i < c.length; i++)
                c[i] = (Thing2)in.readObject();
            long t2 = System.currentTimeMillis();
            System.out.println(
                "Duplication via serialization: " +
                (t2 - t1) + " Milliseconds");
            // Now try cloning:
            t1 = System.currentTimeMillis();
            Thing4[] d = new Thing4[SIZE];
            for(int i = 0; i < d.length; i++)
                d[i] = (Thing4)b[i].clone();
            t2 = System.currentTimeMillis();
            System.out.println(
                "Duplication via cloning: " +
                (t2 - t1) + " Milliseconds");
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
} ///:~

```

Thing2 и **Thing4** съдържат обекти-членове, така че е налице някакво дълбоко копиране. Интересно е да се отбележи че докато **Serializable** класовете са лесни за устройване, има много повече работа за да бъдат дублицирани. Клонирането докарва много работа за уреждането на класа, но фактическото дублициране е сравнително просто. Резултатите действително говорят. Ето изход от три различни пускания:

```
Duplication via serialization: 3400 Milliseconds
```

```
Duplication via cloning: 110 Milliseconds
```

```
Duplication via serialization: 3410 Milliseconds
```

```
Duplication via cloning: 110 Milliseconds
```

```
Duplication via serialization: 3520 Milliseconds
```

```
Duplication via cloning: 110 Milliseconds
```

Напук на обевидно огромната разлика във времето за сериализация и клониране, също ще забележите и че сериализационната техника значително варира по продължителност, докато клонирането става за еднакво време всеки път.

Добавяне на клонируемост надолу по иерархията

Ако създавате нов клас неговият базов клас е по подразбиране **Object**, което означава неклонируемост по подразбиране (както ще видите в следващата сесия). Докато не добавите клонируемост явно няма и да я имате. Но може да я добавите в някакъв слой и от там надолу вече ще я има, както това:

```
//: c12:HorrorFlick.java
// You can insert Cloneability at any
// level of inheritance.
import java.util.*;

class Person {}
class Hero extends Person {}
class Scientist extends Person
    implements Cloneable {
    public Object clone() {
        try {
            return super.clone();
        } catch (CloneNotSupportedException e) {
            // this should never happen:
            // It's Cloneable already!
            throw new InternalError();
        }
    }
}
class MadScientist extends Scientist {}

public class HorrorFlick {
    public static void main(String[] args) {
        Person p = new Person();
        Hero h = new Hero();
        Scientist s = new Scientist();
        MadScientist m = new MadScientist();

        // p = (Person)p.clone(); // Compile error
        // h = (Hero)h.clone(); // Compile error
        s = (Scientist)s.clone();
        m = (MadScientist)m.clone();
    }
} ///:~
```

Преди да бъде добавена клонируемост компилаторът ви спираше при опити да клонирате нещо. Когато клонируемост беше добавена за **Scientist**, **Scientist** и всичките му наследници са клонирани.

Зашо този странен дизайн?

Ако всичко това изглежда странна схема, така е, защото тя наистина е такава. Може да се чудите защо е направено по този начин. Какво се крие зад този дизайн? Това което следва не е доказана прецизна история – може би защото търговията около Java го показва като перфектно проектиран език – но тя върви дълго по пътя към обясняването защо нещата са такива, каквито са оказали.

Оригинално Java беше замислена като език за управление на хардуерни устройства и определено не с Internet в замисъла. В език с общо предназначение като този има смисъл да се даде възможност програмистът да клонира всеки обект. Така **clone()** беше сложен в кореновия клас **Object**, но беше **public** метод така че всеки обект можеше да се клонира. Това изглеждаше като най-разумен подход и, най-после, как можеше да навреди?

Добре, когато Java се виждаше вече като чудесен език за програмиране в/за Internet, нещата се промениха. Внезапно изскочиха изискванията за сигурност и, разбира се, тяхната връзка с използването на обекти, и липсата на желание всички обекти да могат да бъдат клонирани от когото и да е. Така че това което се вижда са множеството кръпки по първоначалната праволинейна и прости схема: **clone()** е сега **protected** в **Object**. Трябва да го подтиснете и да **implement Cloneable** и да работите с изключенията.

Нищо не е че трябва да използвате **Cloneable** интерфейса самоако се каните да викате **clone()** на **Object** понеже той проверява по време на изпълнение дали обектът е **Cloneable**. Но за пълнота (и понеже **Cloneable** е празен така или иначе) ще го прилагате.

Управление на клонируемостта

Би могло да се помисли че за премахване на клонируемостта просто трябва да се направи метода **clone()** да бъде **private**, но няма да стане така понеже не може да се направи метод в базовия клас по-**private** в извлечен клас. Така че не е толкова просто. И все пак, необходимо е да се управлява клонируемостта на обектите. Трябва да се обрне внимание на няколко неща в тази връзка:

1. Безразличие. Не правите нищо за да бъде вашият клас клониран, но наследниците му могат да се направят да бъдат ако е необходимо. Това работи само ако **Object.clone()** по подразбиране прави необходимото с полетата във вашия клас.
2. Поддръжка на **clone()**. Следвайте стандартната практика на прилагане на **Cloneable** и подтискане на **clone()**. В подтиснатия **clone()** викате **super.clone()** и прихващате всички изключения (така че подтиснатия **clone()** не изхвърля изключения).
3. Условно поддържане на клонирането. Ако вашият клас съдържа манипулатори които могат да бъдат или да не бъдат клонирани (един пример е клас-колекцията), може да се опитате да клонирате всички тях, а ако изхвърлят изключения – просто да ги отминете. Например да вземем специален вид **ArrayList** който се опитва да клонира всички съдържани в него обекти. Когато пишете такъв **ArrayList**, не знаете че обекти ще сложи вътре клиент-програмистът във вашия **ArrayList**, така че не знаете дали са клонирани.

4. Не прилагайте **Cloneable** ами подтиснете **clone()** като **protected**, давайки коректно поведение три копирането на всички полета. По този начин всеки който наследява този клас може да подтисне **clone()** и извика **super.clone()** за получаване на коректно поведение прикопирането. Забележете че вашата реализация може и трябва да вика **super.clone()** даже ако методът очаква **Cloneable** обект (иначе ще се изхвърли изключение), понеже никой няма да го вика от обект точно с вашия тип. Той ще се вика само от наследен клас, който, за да работи успешно, прилага **Cloneable**.
5. Опитайте се да предотвратите клонирането чрез неприлагане на **Cloneable** и подтискане на **clone()** за изхвърляне на изключение. Това е успешно само ако всеки извлечен клас вика **super.clone()** в своята повторна дефиниция на **clone()**. Иначе програмистът би могъл да го заобиколи.
6. Предотвратете клонирането чрез правене на вашия клас **final**. Ако **clone()** не е бил подтиснат в някой от предшестващите ваши класове, не може и да бъде вече. Ако е бил, подтиснете го и изхвърлете **CloneNotSupportedException**. Правенето на класа **final** е единствения начин за гарантирана невъзможност за клониране. Освен това когато имате работа по сигурността или други случаи когато трябва да се управлява броя на създаваните обекти, ще направите всичките конструктори **private** и ще дадете един или повече специални методи за създаване на обекти. По този начин тези методи могат да въведат ограничения върху броя на обектите и условията при които те се създават. (Частен случай на това е *singleton* шаблона описан в глава 16.)

Ето пример който показва как могат да се реализират различните начини за клониране и после, надолу по йерархията, да бъдат "изключени":

```
//: c12:CheckCloneable.java
// Checking to see if a handle can be cloned

// Can't clone this because it doesn't
// override clone():
class Ordinary {}

// Overrides clone, but doesn't implement
// Cloneable:
class WrongClone extends Ordinary {
    public Object clone()
        throws CloneNotSupportedException {
            return super.clone(); // Throws exception
    }
}

// Does all the right things for cloning:
class IsCloneable extends Ordinary
    implements Cloneable {
    public Object clone()
        throws CloneNotSupportedException {
            return super.clone();
    }
}

// Turn off cloning by throwing the exception:
```

```

class NoMore extends IsCloneable {
    public Object clone()
        throws CloneNotSupportedException {
        throw new CloneNotSupportedException();
    }
}

class TryMore extends NoMore {
    public Object clone()
        throws CloneNotSupportedException {
        // Calls NoMore.clone(), throws exception:
        return super.clone();
    }
}

class BackOn extends NoMore {
    private BackOn duplicate(BackOn b) {
        // Somehow make a copy of b
        // and return that copy. This is a dummy
        // copy, just to make the point:
        return new BackOn();
    }
    public Object clone() {
        // Doesn't call NoMore.clone():
        return duplicate(this);
    }
}

// Can't inherit from this, so can't override
// the clone method like in BackOn:
final class ReallyNoMore extends NoMore {}

public class CheckCloneable {
    static Ordinary tryToClone(Ordinary ord) {
        String id = ord.getClass().getName();
        Ordinary x = null;
        if(ord instanceof Cloneable) {
            try {
                System.out.println("Attempting " + id);
                x = (Ordinary)((IsCloneable)ord).clone();
                System.out.println("Cloned " + id);
            } catch(CloneNotSupportedException e) {
                System.out.println(
                    "Could not clone " + id);
            }
        }
        return x;
    }
    public static void main(String[] args) {
        // Upcasting:
        Ordinary() ord = {
            new IsCloneable(),
            new WrongClone(),
            new NoMore(),
            new TryMore(),
            new BackOn(),
            new ReallyNoMore(),

```

```

};

Ordinary x = new Ordinary();
// This won't compile, since clone() is
// protected in Object:
//! x = (Ordinary)x.clone();
// tryToClone() checks first to see if
// a class implements Cloneable:
for(int i = 0; i < ord.length; i++)
    tryToClone(ord(i));
}
} //:~

```

Първият клас, **Ordinary**, представя видовете кладове които сме срещали по протежение на книгата: без поддръжка на клониране, но както излиза, също и без предотвратяване на клонирането. Но ако имате манипулятор към **Ordinary** обект който може да е бил ъпкастнат от по-извлечен (по-нататък в юерархията - б.пр.) клас, не може да кажете дали е клонирам или не.

Класът **WrongClone** показва некоректен начин за реализиране на клонирането. Той подписка **Object.clone()** и го прави **public**, но не прилага **Cloneable**, та когато се извика **super.clone()** (което резутира в извикване на **Object.clone()**), **CloneNotSupportedException** се изхвърля така че клонирането не работи.

В **IsCloneable** може да видите всички правилни действия свързани с клонирането: **clone()** е подтиснато и **Cloneable** се прилага. Обаче този **clone()** метод и няколко други нататък в примера не прихващат **CloneNotSupportedException**, а вместо това го подават на извикващия метод, та трябва да има трай-хвани блок около него. Във вашия собствен **clone()** метод типично ще прихващате **CloneNotSupportedException** вътре в **clone()** наместо да го подавате нататък. Както ще видите, за този пример е по-информиращо да се предаде изключението.

Класът **NoMore** се опитва да "изключи" по начина желан от дизайнериите на Java: в извлечения клас **clone()** изхвърляте **CloneNotSupportedException**. Методът **clone()** в класа **TryMore** правилно вика **super.clone()**, това довежда **NoMore.clone()**, който изхвърля изключение и предотвратява клонирането.

Ами ако програмистът не следва "правилния" начин за викане на **super.clone()** в подтиснатия метод **clone()**? В **BackOn** може да видите това да се случи. Този клас използва отделен метод **duplicate()** за да направи копие на текущия обект и вика този метод вътре в **clone()** вместо да вика **super.clone()**. Не се изхвърля изключение и новият клас е клонирам. Не може да разчитате на изхвърлянето на изключения за предотвратяването на клонируемостта на клас. Единственото истинско решение е посочено в **ReallyNoMore**, който е **final** и затова не може да бъде наследяван. Това значи че ако **clone()** изхвърля изключение във **final** клас това не може да се промени чрез наследяване и мащабето на клонируемостта е гарантирано. (Не може явно да извикате **Object.clone()** от клас, който има произволно ниво на наследяване; ограничени сте до **super.clone()**, който има достъп само до прекия базов клас.) Така, ако направите обекти които са свързани с въпроси на сигурността, ще вземете да ги направите **final**.

Първия метод който виждате в класа **CheckCloneable** е **tryToClone()**, който взема **Ordinary** обект и проверява дали той е клонирам с **instanceof**. Акое, прави каст към **IsCloneable**, вика **clone()** и прави каст на резултатите обратно към **Ordinary**, прихващайки всички изхвърлени изключения. Забележете използването на идентификация на типа по време на изпълнение (виж глава 11) за извеждане на името на класа, та да видите какво става.

В **main()**, различни типове от **Ordinary** се създават и ъпкастват **Ordinary** в дефиницията на масива. Първите две линии след това създават прост **Ordinary** обект и се опитват да го клонират. Този код обаче няма да се компилира понеже **clone()** е **protected** метод в **Object**.

Останалата част от кода пробягва масива и се опитва да клонира всеки обект, докладвайки за успеха или неуспеха на всяко начинание. Изходът е:

```
Attempting IsCloneable
Cloned IsCloneable
Attempting NoMore
Could not clone NoMore
Attempting TryMore
Could not clone TryMore
Attempting BackOn
Cloned BackOn
Attempting ReallyNoMore
Could not clone ReallyNoMore
```

Като резюме, ако искате клас да бъде клониран:

1. Прилагате **Cloneable** интерфейс.
2. Поддържате **clone()**.
3. Викате **super.clone()** вътре в **clone()**.
4. Хващате изключенията вътре в **clone()**.

Това ще доведе до най-подходящия ефект.

Копи-конструкторът

Клонирането изглежда сложен за организиране процес. Може да изглежда, че е необходима алтернатива. Единият подход който може да ви хареса (особено ако сте C++ програмист) е да се направи специален конструктор, който да дублира класовете. В C++ това се нарича *копи(раш) конструктор*. Отначало всичко изглежда очевидно. Ето пример:

```
//: c12:CopyConstructor.java
// A constructor for copying an object
// of the same type, as an attempt to create
// a local copy.

class FruitQualities {
    private int weight;
    private int color;
    private int firmness;
    private int ripeness;
    private int smell;
    // etc.
    FruitQualities() { // Default constructor
        // do something meaningful...
    }
    // Other constructors:
    // ...
    // Copy constructor:
    FruitQualities(FruitQualities f) {
        weight = f.weight;
        color = f.color;
        firmness = f.firmness;
        ripeness = f.ripeness;
        smell = f.smell;
        // etc.
    }
}
```

```

        }

    }

class Seed {
    // Members...
    Seed() { /* Default constructor */ }
    Seed(Seed s) { /* Copy constructor */ }
}

class Fruit {
    private FruitQualities fq;
    private int seeds;
    private Seed() s;
    Fruit(FruitQualities q, int seedCount) {
        fq = q;
        seeds = seedCount;
        s = new Seed(seeds);
        for(int i = 0; i < seeds; i++)
            s(i) = new Seed();
    }
    // Other constructors:
    // ...
    // Copy constructor:
    Fruit(Fruit f) {
        fq = new FruitQualities(f.fq);
        seeds = f.seeds;
        // Call all Seed copy-constructors:
        for(int i = 0; i < seeds; i++)
            s(i) = new Seed(f.s(i));
        // Other copy-construction activities...
    }
    // To allow derived constructors (or other
    // methods) to put in different qualities:
    protected void addQualities(FruitQualities q) {
        fq = q;
    }
    protected FruitQualities getQualities() {
        return fq;
    }
}

class Tomato extends Fruit {
    Tomato() {
        super(new FruitQualities(), 100);
    }
    Tomato(Tomato t) { // Copy-constructor
        super(t); // Upcast for base copy-constructor
        // Other copy-construction activities...
    }
}

class ZebraQualities extends FruitQualities {
    private int stripedness;
    ZebraQualities() { // Default constructor
        // do something meaningful...
    }
    ZebraQualities(ZebraQualities z) {

```

```

super(z);
stripedness = z.stripedness;
}

}

class GreenZebra extends Tomato {
GreenZebra() {
    addQualities(new ZebraQualities());
}
GreenZebra(GreenZebra g) {
    super(g); // Calls Tomato(Tomato)
    // Restore the right qualities:
    addQualities(new ZebraQualities());
}
void evaluate() {
    ZebraQualities zq =
        (ZebraQualities)getQualities();
    // Do something with the qualities
    // ...
}
}

public class CopyConstructor {
public static void ripen(Tomato t) {
    // Use the "copy constructor":
    t = new Tomato(t);
    System.out.println("In ripen, t is a " +
        t.getClass().getName());
}
public static void slice(Fruit f) {
    f = new Fruit(f); // Hmm... will this work?
    System.out.println("In slice, f is a " +
        f.getClass().getName());
}
public static void main(String[] args) {
    Tomato tomato = new Tomato();
    ripen(tomato); // OK
    slice(tomato); // OOPS!
    GreenZebra g = new GreenZebra();
    ripen(g); // OOPS!
    slice(g); // OOPS!
    g.evaluate();
}
} ///:~

```

Това изглежда малко странно на пръв поглед. Плодът има качества сигурно, но защо просто не се сложат даннови членове които ги представят направо в класа **Fruit**? Има две потенциални причини. Първата е че може да искате бързо и лесно да променяте количествата. Забележете че **Fruit** има **protected** метод **addQualities()** за да позволи на извлечените класове да направят това. (Може да помислите че е логично да има **protected** конструктор на **Fruit** който приема **FruitQualities** аргумент, но конструкторите не се наследяват и той ще бъде безполезен във второто и по-високо ниво на йерархията.) Чрез правене на качества на плодовете отделен клас се получава голяма гъвкавост включително възможността да се променят по време на живота на конкретен обект.

Има втора причина да се направи **FruitQualities** отделен обект в случай че смятате да добавяте количества или да променяте поведението чрез наследяване и полиморфизъм.

Забележете че за **GreenZebra** (което в действителност е вид домати – развъждал съм ги и те са баснословни), конструкторът вика **addQualities()** и му подава **ZebraQualities** обект, който е извлечен от **FruitQualities** така че може да бъде присъединен към **FruitQualities** манипулатора на базовия клас. Разбира се, когато **GreenZebra** използва **FruitQualities** трябва да се направи даункаст към провилния тип (както се вижда в **evaluate()**), но той винаги знае че типът е **ZebraQualities**.

Ще видите също че има **Seed** клас и че **Fruit** (който по дефиниция си има семена) съдържа масив от **Seed**-ове.

Накрая забележете че всеки клас си има копи-конструктор и че всеки копиращ конструктор трябва да се грижи за викането на копиконструктора на базовия клас и членовете необходими за дълбокото копиране. Копи конструкторът е пробван вътре в класа **CopyConstructor**. Методът **ripen()** взима **Tomato** аргумент и изпълнява копиращо конструиране с него за да дублицира обекта:

```
| t = new Tomato(t);
```

докато **slice()** взима по-родовия **Fruit** и така го дублицира:

```
| f = new Fruit(f);
```

Те са пробвани с различни видове **Fruit** в **main()**. Ето изхода:

```
In ripen, t is a Tomato  
In slice, f is a Fruit  
In ripen, t is a Tomato  
In slice, f is a Fruit
```

Ето тук се показва проблемът. След копиращата конструкция която става с **Tomato** вътре в **slice()**, резултатът не е вече **Tomato** обект, а просто **Fruit**. Той е загубил всичко доматено. По-нататък, когато вземате **GreenZebra**, както **ripen()** така и **slice()** се превръщат в **Tomato** и **Fruit**, респективно. Така, за нещастие, схемата на копиращите конструктори не е полезна за нас в Java когато се опитваме да направим локално копие на обект.

Защо работи в C++ и не работи в Java?

Копи конструкторът е основна част от C++, понеже автоматично прави локално копие на обект. И все пак горният пример показва че това не работи в Java. Защо? В Java всичко с което работим е манипулятор, докато в C++ може да имате прилични на манипулятори същности и също може да подадете обект направо. За това е копи-конструкторът в C++: когато искате да вземете обект и да го подадете по стойност, дублицирайки по този начин обекта. Това работи чудесно в C++, но ще запомните че тази схема се проваля в Java, така че не я използвайте.

Класове само за четене

Докато локалното копие произведено от **clone()** дава желаните резултати в съответните случаи, то е и пример на заставяне на програмиста (автора на метода) да бъде отговорен за лошите ефекти на псевдонимите. Какво ще стане ако правите толкова много употребявана и обща библиотека, че не може да се знае дали винаги ще бъде клонирано където трябва? Или по-вероятно, какво ако вие искате да позволите псевдоними заради ефективността – за предотвратяване на ненужна дубликация на обекти – но не искате отрицателната страна на псевдонимите?

Едно решение е да се създаде *неизменяем* обект който принадлежи към класовете само за четене. Може да дефинирате така един клас, че да няма методи които да му променят вътрешното състояние. В такъв клас псевдонимите не могат да попречат понеже вътрешната структура може само да се чете, така че и много места в кода да я четат няма проблеми.

Като прост пример с неизменяеми обекти стандартната библиотека на Java съдържа "обгръщащи" класове за всички примитивни типове. Може вече да сте открили това, че ако искате да запомните `int` в колекция като `ArrayList` (която взема само `Object` манипулатори), може да обградите вашия `int` в класа от стандартната библиотека `Integer`:

```
//: c12:ImmutableInteger.java
// The Integer class cannot be changed
import java.util.*;

public class ImmutableInteger {
    public static void main(String[] args) {
        ArrayList v = new ArrayList();
        for(int i = 0; i < 10; i++)
            v.add(new Integer(i));
        // But how do you change the int
        // inside the Integer?
    }
} ///:~
```

Класът `Integer` (както и всички "обгръщащи" примитиви класове) реализира непроменимост по прост начин: те нямат методи които позволяват да се променя обекта.

Ако се нуждадете от обект който съдържа примитивен тип на който да може да се променя стойността, трябва да го създадете сами. За щастие това е тривиално:

```
//: c12:MutableInteger.java
// A changeable wrapper class
import java.util.*;

class IntValue {
    int n;
    IntValue(int x) { n = x; }
    public String toString() {
        return Integer.toString(n);
    }
}

public class MutableInteger {
    public static void main(String[] args) {
        ArrayList v = new ArrayList();
        for(int i = 0; i < 10; i++)
            v.add(new IntValue(i));
        System.out.println(v);
        for(int i = 0; i < v.size(); i++)
            ((IntValue)v.get(i)).n++;
        System.out.println(v);
    }
} ///:~
```

Забележете че `n` е приятелско за опростяване на кодирането.

`IntValue` може да е даже по-просто ако инициализацията по подразбиране с нула се окаже подходяща (тогава не ви трябва конструктора) и не се грижите за извеждането `toString()`: (тогава не се нуждадете от `toString()`):

```
class IntValue { int n; }
```

Извеждането на елемент и кастингът му са малко тромави, но това е черта на `ArrayList`, не на `IntValue`.

Създаване на класове само за четене

Възможно е да създадете собствен клас само за четене. Ето пример:

```
//: c12:Immutable1.java
// Objects that cannot be modified
// are immune to aliasing.

public class Immutable1 {
    private int data;
    public Immutable1(int initVal) {
        data = initVal;
    }
    public int read() { return data; }
    public boolean nonzero() { return data != 0; }
    public Immutable1 quadruple() {
        return new Immutable1(data * 4);
    }
    static void f(Immutable1 i1) {
        Immutable1 quad = i1.quadruple();
        System.out.println("i1 = " + i1.read());
        System.out.println("quad = " + quad.read());
    }
    public static void main(String[] args) {
        Immutable1 x = new Immutable1(47);
        System.out.println("x = " + x.read());
        f(x);
        System.out.println("x = " + x.read());
    }
} //:~
```

Всички данни са **private** и виждате, че никой от **public** методите не променя данните. Разбира се, методът който изглежда че променя данните е **quadruple()**, но това създава нов **Immutable1** и оставя оригиналния незасегнат.

Методът **f()** взима **Immutable1** обект и изпълнява различни операции с него, а изходът от **main()** демонстрира че **x** няма промяна. Така обектът на **x** може да има много псевдоними без вреда понеже класът **Immutable1** е проектиран да осигури че обектите от него не могат да бъдат променяни.

Недостатъкът на непроменимостта

Отначало създаването на непроменим клас изглежда елегантно решение. Обаче щом се наложи промяна на стойности страдате от допълнителната работа за създаване на нов обект, както и потенциално по-честото събиране на боклука. За някои класове това не е проблем, но за други (като класа **String**) това е забранително скъпо.

Решението е да се създава съпътстващ клас който може да бъде променян. В такъв случай когато правите много промени, може да се превключите към променящия компаньон и отново към непроменимия клас когато свършите.

Горният пример може да бъде променен да показва това:

```
//: c12:Immutable2.java
// A companion class for making changes
// to immutable objects.

class Mutable {
```

```

private int data;
public Mutable(int initVal) {
    data = initVal;
}
public Mutable add(int x) {
    data += x;
    return this;
}
public Mutable multiply(int x) {
    data *= x;
    return this;
}
public Immutable2 makeImmutable2() {
    return new Immutable2(data);
}
}

public class Immutable2 {
    private int data;
    public Immutable2(int initVal) {
        data = initVal;
    }
    public int read() { return data; }
    public boolean nonzero() { return data != 0; }
    public Immutable2 add(int x) {
        return new Immutable2(data + x);
    }
    public Immutable2 multiply(int x) {
        return new Immutable2(data * x);
    }
    public Mutable makeMutable() {
        return new Mutable(data);
    }
    public static Immutable2 modify1(Immutable2 y){
        Immutable2 val = y.add(12);
        val = val.multiply(3);
        val = val.add(11);
        val = val.multiply(2);
        return val;
    }
    // This produces the same result:
    public static Immutable2 modify2(Immutable2 y){
        Mutable m = y.makeMutable();
        m.add(12).multiply(3).add(11).multiply(2);
        return m.makeImmutable2();
    }
    public static void main(String[] args) {
        Immutable2 i2 = new Immutable2(47);
        Immutable2 r1 = modify1(i2);
        Immutable2 r2 = modify2(i2);
        System.out.println("i2 = " + i2.read());
        System.out.println("r1 = " + r1.read());
        System.out.println("r2 = " + r2.read());
    }
} ///:~

```

Immutable2 съдържа методи които, както преди, запазват неизменяемостта на обектите чрез правене на нов обект винаги щом е необходима промяна. Те са **add()** и **multiply()** методите. Съществуващият клас е наречен **Mutable** и също има **add()** и **multiply()** методи, но те променят **Mutable** обекта наместо да правят нов. Освен това **Mutable** няма метод да използва данните си да прави **Immutable2** обект и обратно.

Вата статични метода **modify1()** и **modify2()** показват два различни подхода за получаване на един и същ резултат. В **modify1()** всичко е направено в **Immutable2** и може да видите че четири нови **Immutable2** са създадени при работата. (И всеки път когато **val** се пре-приравнява, предишният обект става боклук.)

В метода **modify2()** може да видите че първо се взема **Immutable2** и се създава **Mutable** от него. (Това е точно като викането на **clone()** както видяхте по-рано, но този път се създава различен тип обект.) Тогава **Mutable** обектът се използва за правене на много модификации без да е необходимо създаването на много нови обекти. Накрая отново се превръща в **Immutable2**. Тук се създават два нови обекта (**Mutable** и резултата **Immutable2**) вместо четири.

Този подход е подходящ когато:

1. Ви трябва неизменяем обект и
2. Често трябва да правите много модификации или
3. Е скъпо да правите нови неизменяеми обекти

Неизменяеми Stringове

Да видим следния код:

```
//: c12:Stringer.java

public class Stringer {
    static String upcase(String s) {
        return s.toUpperCase();
    }
    public static void main(String[] args) {
        String q = new String("howdy");
        System.out.println(q); // howdy
        String qq = upcase(q);
        System.out.println(qq); // HOWDY
        System.out.println(q); // howdy
    }
} //:~
```

Когато **q** се подава в **upcase()** това фактически е копие на манипулатора към **q**. Обектът сочен от този манипулятор си стои в единствено физическо място. Манипуляторите се копират при подаването.

Гледайки дефиницията на **upcase()** може да забележите че подаваният манипулятор има име **s** и съществува само докато тялото на **upcase()** се изпълнява. Когато **upcase()** завърши локалният манипулятор **s** изчезва. **upcase()** връща резултата, който е оригиналният стринг с всички знаци в горен регистър. Разбира се се връща манипулятор към резултата. Но излиза че върнатият манипулятор е за новия обект, а оригиналният **q** е изоставен. Как се случва това?

Неявни константи

Ако кажем:

```
| String s = "asdf";
| String x = Stringer.toUpperCase(s);
```

Искаме ли наистина методът `toUpperCase()` да промени аргумента? Изобщо, не, понеже аргументът изглежда на четящия кода като парче подадена на метода информация, не като нещо за модифициране. Това е важна гаранция, понеже прави кода по-лесен за четене и разбиране.

В C++ наличността на такава гаранция беше достатъчно важна, за да се използва специална ключова дума, `const`, да позволи на програмиста да гарантира че манипуляторът (указател или псевдоним в C++) няма да бъде използван за промяна на оригиналния обект. Но C++ програмистът трябваше да бъде грижлив и да използва `const` навсякъде. Това може да бъде смущаващо и лесно за забравяне.

Претоварване на ‘+’ и `StringBuffer`

Обектите от класа `String` са проектирани да бъдат неизменяеми, чрез показаната технология. Ако разгледате онлайн документацията за класа `String` (която е дадена в резюме по-късно в тази глава), ще видите че всеки метод който променя `String` фактически създава и връща съвсем нов `String` обект съдържащ модификацията. Оригиналният `String` остава незасегнат. Така няма черта в Java подобна на `const` в C++ за поддръжка от страна на компилатора на неизменяемостта на вашите обекти. Ако я искате, трябва да я направите сами, както прави `String`.

Понеже `String` обектите са неизменяеми, може един `String` да има колкото си искате псевдоними. Понеже е само за четене няма как един манипулятор да измени нещо за другите манипулятори. Така че обектът само за четене добре решава проблема с псевдонимите.

Изглежда също възможно да се задоволят всички случаи на промяна като се създава съвсем нов обект съдържащ я, както прави `String`. Обаче за някои операции това не е ефективно. Такъв случай е операторът ‘+’ който е претоварен за `String` обекти. Претоварването значи че се добавя допълнително значение когато се приложи за конкретен клас. (‘+’ и ‘+=’ за `String` са единствените оператори които са претоварени в Java и Java не позволява на програмистът да претоварва които и да било други⁴).

Когато се използва със `String` обекти ‘+’ позволява да се конкатенират `String`ове+:

```
| String s = "abc" + foo + "def" + Integer.toString(47);
```

Бихме могли да си представим как това би могло да работи: `String`ът “abc” би могъл да има метод `append()` който създава нов `String` обект съдържащ “abc” съединен със съдържанието на `foo`. Новият `String` обект после би създал нов `String` в който добавя “def” и така нататък.

Това сигурно ще работи, но то би изисквало създаването на много `String` обекти само за да се съединят `String`овете, а тогава ще има много междуинни `String` обекти които трябва да се оберат като боклук. Подозирам че проектантите на Java са опитали този подход отначало (което е урок по програмно проектиране – нищо не знаете за системата докато не направите някакъв код да работи). Подозирам също че те са открили, че това води до неприемлива производителност.

Решението е променим съпътстващ клас какъвто беше вече показан. За `String` този съпътстващ клас е наречен `StringBuffer` и компилаторът автоматично създава `StringBuffer` за изчисляване на

⁴ C++ allows the programmer to overload operators at will. Because this can often be a complicated process —(see Chapter 10 of my book *Thinking in C++* (Prentice-Hall, 1995).—the Java designers deemed it a “bad” feature that shouldn’t be included in Java. It wasn’t so bad that they didn’t end up doing it themselves, and ironically enough, operator overloading would be much easier to use in Java than in C++.

някои изрази, в частност когато претоварените оператори `+` и `+=` се използват със **String** обекти. Този пример показва какво става:

```
//: c12:ImmutableStrings.java
// Demonstrating StringBuffer

public class ImmutableStrings {
    public static void main(String[] args) {
        String foo = "foo";
        String s = "abc" + foo +
            "def" + Integer.toString(47);
        System.out.println(s);
        // The "equivalent" using StringBuffer:
        StringBuffer sb =
            new StringBuffer("abc"); // Creates String!
        sb.append(foo);
        sb.append("def"); // Creates String!
        sb.append(Integer.toString(47));
        System.out.println(sb);
    }
} ///:~
```

При създаването на **String s** компилаторът прави код грубо еквивалентен на следния използваш **sb**: създава се **StringBuffer** и се използва **append()** за добавяне на нови знаци направо към **StringBuffer** обекта (наместо да се прави ново копие всеки път). Докато това е по-ефективно, нищо не струва че всеки път когато създавате стринг с кавички като **"abc"** и **"def"** компилаторът ги превръща в **String** обекти. Така че може да има повече на брой от очакваното обекти, напук на ефективността постигната от **StringBuffer**.

Класовете **String** и **StringBuffer**

Ето преглед на методите достъпни в **String** и **StringBuffer** така че може да си създадете представа за начина по който си взаимодействват. Табличите не съдържат всеки отделен метод, а само важните за тази дискусия. Методите които са претоварени са резюмирани в един ред.

Първо, класът **String**:

Метод	Аргументи, Претоварване	Употреба
Конструктор	Претоварени: Default, String , StringBuffer , char МАСИВИ, byte МАСИВИ.	Създаване на String обекти.
length()		Брой на знаците в String .
charAt()	int Index	Char -ът в определено място на Stringa .
getChars() , getBytes()	Началото и края откъдето да се копира, масивът в който се копира, индекс в последния.	Копира charове или bytes във външен масив.
toCharArray()		Дава char() съдържащ знаците в Stringa .

Метод	Аргументи, Претоварване	Употреба
equals(), equals-IgnoreCase()	String с който да се сравнява.	Тест за равенство на съдържанията на два Stringa .
compareTo()	String с който да се сравнява.	Резултата негативен, нула или положителен в зависимост от лексикографичното подреждане на String и аргумента. Малките и големи букви не са равни!
regionMatches()	Отместване в този String , другия String и неговото отместване и дължина за сравняване. С оверлоудинг се добавя "пренебрегване на регистра."	Булев резултат индициращ дали обхватът съвпада.
startsWith()	String с който може да започва. Оверлоуд добавя отместване.	Булев резултат индициращ дали String започва с аргумента.
endsWith()	String който може да бъде суфикс на този String .	Булев резултат индициращ дали е суфикс.
indexOf(), lastIndexOf()	Претоварени: char, char и начален индекс, String, String и начален индекс	Връща -1 ако аргументът не е намерен в този String , иначе връща индексът където аргументът започва. lastIndexOf() търси отзад напред.
substring()	Претоварени: Начален индекс, начален индекс и индекс на края.	Връща нов String обект съдържащ специфицилания знаков набор.
concat()	Stringa за конкатениране	Връща нов String обект съдържащ знаците на оригиналния String следвани от знаците на аргумента.
replace()	Старият знак който се търси, новия знак с който да се замени.	Връща нов String обект с направени замествания. Използва стария String ако не е намерено съвпадение.
toLowerCase()		Връща нов String обект с

Метод	Аргументи, Претоварване	Употреба
toUpperCase()		променен регистър на буквите. Изполва стария String ако не са били необходими промени.
trim()		Връща нов String обект с непечатуемите знаци мащнати от края. Използва стария String ако не са били необходими промени.
valueOf()	Претоварени: Object, char(), char() и отмествания и брой, boolean, char, int, long, float, double .	Връща String съдържащ знаковото представяне на аргумента.
intern()		Произвежда единствен String манипулатор за всяка уникална знакова редица.

Може да видите че всеки метод на **String** грижливо връща нов **String** когато е необходимо да се промени съдържанието. Също когато съдържанието не се променя връща се манипулатор към оригиналния **String**. Това спестява памет и допълнителна работа.

Ето класа **StringBuffer**:

Метод	Аргументи,претоварване	Използване
Конструктор	Претоварени: по подразбиране, дължина на създавания буфер, String от който да се създаде.	Създава нов StringBuffer обект.
toString()		Създава String от този StringBuffer .
length()		Брой на знаците в StringBuffer a.
capacity()		Връща текущия брой алокирани шпации.
ensureCapacity()	Цяло индициращо желания капацитет.	Прави StringBuffer способен да побере най-малко желаното количество знаци.
setLength()	Цяло индициращо новата дължина на стринга в буфера.	Реже и разширява предишния стринг. Ако разширява, пълни с нули.
charAt()	Цяло показващо индекса на желания елемент.	Връща char който е на това място в буфера.
setCharAt()	Цяло показващо индекса на желания елемент и новата char стойност на	Модифицира стойността на това място.

Метод	Аргументи, претоварване	Използване
	елемента.	
getChars()	Начало и край от където да се копира, масив в който да се копира, индекс в него.	Копира charове във външен масив. Няма getBytes() като в String .
append()	Претоварени: Object, String, char(), char() с отместване и дължина, boolean, char, int, long, float, double .	Аргументът се превръща в стринг и добавя към края на текущия буфер, увеличавайки го ако е необходимо.
insert()	Претоварени, всеки с пръв аргумент отместването където да започне вмъкването: Object, String, char(), boolean, char, int, long, float, double .	Вторият аргумент се превръща в стринг и се вмъква в текущия буфер започвайки от отместването. Буфера се увеличава ако е необходимо.
reverse()		Редът на знаците в буфера е обърнат.

Най-често използвания метод е **append()**, който се използва и от компилатора за изчисляване на **String** изрази които съдържат `+` и `+=` оператори. Методът **insert()** има подобна форма, а двата метода изпълняват значителна работа със стринговете без да създават нов обект.

Стринговете са специални

До тук видяхме че класът **String** не е просто още един клас в Java. Има множество специални случаи със **String**, не най-маловажният от които е че това е вграден клас и основан за Java. После идва фактът че стринговете в кавички се превръщат в **String** от компилатора и специални претоварени оператори **+** и **+=**. В тази глава видяхте и останалия специален случай: грижливо построена неизменимост с помощта на **StringBuffer** и малко допълнителна магия от компилатора.

Резюме

Понеже всичко е манипулатор в Java и понеже всички обекти се създават на хийпа и се почистват само когато вече не са необходими, вкусът на обработката на обекти се променя, специално когато се подават и връщат обекти. Например в C или C++ ако искате да инициализирате някаква област от паметта в един метод, вероятно ще искате потребителят да даде адреса на тази памет. Иначе ще трябва да се притеснявате за това кой ще очисти после тази памет. Така интерфейсът и разбирамостта на такива методи са усложнени. Но в Java никога не се грижите за съществуването на обекта когато вече не е необходим, понеже за това се грижи боклуко-събирането. Програмата ви може да създаде обекта в точката в която е необходим, не преди това, и няма да се беспокоите за предаването на обекта (като параметър - б.пр.): просто предавате манипулатора. Понякога опростяването което носи това е незабележима, друг път е смайваща.

Обратната страна на цялата тази магия е двойна:

1. Винаги получавате удар по скоростта поради допълнителната работа за управление на паметта (макар и това да може и да е твърде малко) и винаги има малка несигурност относно времето за което нещо може да стартира (понеже може да се активира

събирачът на боклук винаги щом има недостиг на памет). За повечето приложения ползите превишават недостатъците, а особено критичните откъм време на изпълнение части могат да се напишат чрез **native** методи (виж приложение A).

2. Псевдонимите: понякога може по стечие на обстоятелствата да завършите с два манипулятора сочещи към един и същ обект, което е проблем само ако те сочат различен обект. Това е на което трябва да се обрне малко повече внимание, когато е необходимо, **clone()**-ирайте обекта за да предотвратите изненадата на другия манипулятор от промяната. Като алтернатива може да допуснете псевдоними заради ефективността, избягвайки проблемите чрез създаването на неизменяеми обекти чито операции могат да върнат нов обект от същия тип или някакъв друг тип, но никога не променят началния обект така че няма никакви промени за псевдонимите които го сочат.

Някои хора твърдят че клонирането в Java е проектирано през куп за грош, така че прилагат тяхна собствена версия на клонирането⁵ и никога не викат метода **Object.clone()** елиминирайки с това необходимостта да използват **Cloneable** и да прихващат **CloneNotSupportedException**. Това сигурно е резонен подход и понеже **clone()** е поддържан така рядко в стандартната библиотека на Java, види се е по-сигурен също така. Но доколкото не викате **Object.clone()** не е необходимо да прилагате **Cloneable** или да прихващате изключението, така че и това ще е приемливо.

Интересно е да се отбележи че едната “запазена но неизползвана” ключова дума в Java е **byvalue**. След разглеждането на въпросите с псевдонимите и клонирането може да си въобразите че **byvalue** може някой ден да бъде използвана за реализацията на автоматично локално копие в Java. Това би могло да елиминира неприятностите с клонирането и да направи програмирането по-просто и с по-добър резултат.

Упражнения

1. Създайте клас **myString** съдържащ **String** обект който инициализирате в конструктора използвайки аргументите му. Добавете **toString()** метод и метод **concatenate()** който добавя **String** обект към вашия вътрешен стринг. Приложете **clone()** в **myString**. Създайте два **static** всеки да взема **myString** x манипулятор като аргумент и да вика **x.concatenate("test")**, но във втория метод извикайте **clone()** първо. Пробвайте двата метода и покажете различния ефект.
2. Създайте клас наречен **Battery** съдържащ **int** който е номер на батерията (като уникален идентификатор). Направете я кланируема и напишете **toString()** метод. Сега създайте клас наречен **Toy** който съдържа масив от **Battery** и **toString()** който извежда всичките батерии. Напишете **clone()** за **Toy** който автоматично клонира всичките **Battery** обекти. Пробвайте това чрез клониране на **Toy** и извеждане на резултатите.
3. Променете **CheckCloneable.java** така че всички **clone()** методи да хващат **CloneNotSupportedException** заместо да го предават към извикващия метод.
4. Променете **Compete.java** с добавяне на повече член-обекти към **Thing2** и **Thing4** и вижте дали ще може да определите промяната на времената в зависимост от сложността – дали това е проста линейна зависимост или е някаква по-сложна.

⁵ Doug Lea, who was helpful in resolving this issue, suggested this to me, saying [that](#) he simply creates a function called **duplicate()** for each class.

5. Започвайки със **Snake.java**, създайте версия с дълбоко копиране на змията.

13: Създаване на прозорци и аплети

(Следващата бележка не се превежда, понеже вече има 8-ма окончателна версия на книгата - б.пр.)

((Note: please notice that only the code files have been changed to be Swing-compliant, and the prose is essentially the way it was before, and refers to the old programs – that is to say, more work is going to be done here. So please concentrate on the code listings, and if you see changes that should be made, please email the entire file after you've made the improvements to Bruce@EckelObjects.com. Thanks!)))

Основно правило в проектирането е “направи простите неща лесни и трудните - възможни.”

Първоначалната цел на графичния потребителски интерфейс в съответната (GUI) библиотека в Java 1.0 беше да позволи на програмиста да направи GUI който изглежда добре на всички платформи. Тази цел не беше достигната. Вместо това *Abstract Window Toolkit* (AWT) на Java 1.0 дава GUI който изглежда еднакво посредствено на всички системи. Освен това е ограничителен: може да използвате само четири шрифта и не може да достигнете по-сложни елементи на GUI който са налице във вашата операционна система. AWT програмният модел на Java 1.0 е също тромав и не е обектно-ориентиран. Един слушател на моите семинари (който беше в Sun по време на създаването на Java) обясни защо: оригиналния AWT е бил концептуализиран, проектиран и реализиран за един месец. Чудо на производителността а също и урок защо е важно проектирането.

Ситуацията е много подобрена в Java 1.1 Събитийния модел на AWT който взема много по-ясен, обектно ориентиран подход, заедно с добавката на Java Beans, среда за компонентно програмиране, чиято цел е бързо и лесно визуално създаване на програми. Java 2 завършва трансформацията от стария Java 1.0 AWT чрез добавяне на *Java Foundation Classes* (JFC), частта GUI на която е наречена “Swing.” Това е множество от лесни за използване, лесни за разбиране Java Beans които могат да бъдат включени и пускане (както и удобно програмиран¹) за създаване на GUI който може (накрая) да бъде удовлетворителен. Правилото “revision 3” на софтуерната индустрия (продуктът не е добър до версия 3) изглежда валидно също и за програмните езици.

Тази глава не разглежда нищо освен съвременната, Java 2 Swing библиотека, и прави резонното предположение че Swing е крайната библиотека GUI за Java. Ако по някаква причина трябва да използвате оригиналния “стар” AWT (понеже поддържате стар код или стари браузъри), може да намерите това въведение в първото он-лайн издание на книгата на www.BruceEckel.com.

Повечето примери ще показват създаването на аплети, най-вече защото е по-просто и лесно да се разберат примерите по този начин. Освен това ще видите разликите когато се създава

¹ A variation on this is called “the principle of least astonishment,” which essentially says: “don’t surprise the user.”

обикновено приложение с използването на Swing, а също и приложения които са и приложения и аплети, та могат да се пускат както чрез браузър така и от команден ред.

Моля имайте предвид, че това не е гълен справочник на методите в съответните класове. Тази глава само ви позволява да започнете основното. Когато търсите повече от това, вижте с браузера да намерите методите които ви трябват. Има множество книги посветени само на Swing и ще трябва да се обърнете към тях ако искате да промените повезението по подразбиране на Swing.

Както си учене за Swing ще откриете че:

1. Swing е много по-добър програмен модел отколокто вероятно сте виждали в други програмни езици и развойни среди. Java Beans е рамката за тази библиотека.
2. "GUI построители" (визуални програмни среди) са *de rigueur* за всички развойни системи. Java Beans и библиотеката Swing позволява на построителя на GUI да пише код заради вас както си попълвате формите. Това не само много ускорява работата при строене на GUI, но и позволява изprobването на повече възможности за късо време и по този начин предполага по-добър дизайн най-накрая.

Основният аплет

Една от основните цели в Java е създаването на *applets*, които са малки програми работещи в Web браузър. Понеже трябва да бъдат безопасни, аплетите са ограничени във възможностите си. Обаче те са мощен инструмент за поддръжка на клиентската страна, основна задача в Web.

Програмирането в рамките на аплет е толкова ограничително, че често се говори за него като за "пясък," понеже винаги има някой – системата за сигурност на Java – да ви следи. Java предлага цифров подпис така че може да позволите на аплети на които може да се вярва достъп до вашата машина. Може също да излезете от пясъка и да напишете нормабно приложение, в който случай имате достъп до различни черти на ОС. През цялото време писахме нормални приложения до сега, но те бяха за конзола без никакви графични компоненти. Swing може също да се използва за постройка на GUI за обикновени приложения.

Библиотеките често се групират според тяхната функционалност. Някои, например, се използват както са си, "от рафта". Стандартните в Java библиотеката **String** и **ArrayList** класове са примери за този случай. Други библиотеки са проектирани специално да съдържат строителни блокове за постройка на други класове. Някой клас от такава библиотека е *application framework*, чиято цел е да дадат базово множество от класове които изпълняват всички общи цели на приложението, чиято е рамката. Тогава за да се приспособи според нуждите поведението трябва да се наследи от тези класове и да се подтиснат методите, които представляват интерес. Механизмът за управление на рамката ще извика вашите методи когато му е времето. Рамките за приложения са добър пример за "разделяне на нещата които се променят от тези които остават същите," понеже се опитват да ограничат всички уникални части на програмата в подтиснатите методи².

Аплетите се пишат чрез използване на рамка. Наследявате от клас **JApplet** и подтискате методите където е нужно. Има няколко метода които управляват създаването и изпълнението на аплети в страница:

Метод	Операция
init()	Вика се когато аплетът е създаден за пръв път и извършва инициализация, вкл. На разположението на компонентите. Винаги ще подтискате този метод.

² This is an example of the design pattern called the *template method*.

Метод	Операция
start()	Вика се всеки път когато аплетът влиза в полезрението на Web браузър за да започне изпълнението на нормалните си операции (особено онези спрени със stop()). Също се вика след init() .
stop()	Вика се всеки път когато аплетът излиза от полезрението на Web броузър за да позволи спирането на скъпите операции. Също се вика непосредствено преди destroy() .
destroy()	Вика се когато аплетът се маха от страницата за освобождаване на всички ресурси когато аплетът вече не е нужен

С тази информация може да създадете прост аплет:

```
//: c13:Applet1.java
// Very simple applet
import javax.swing.*;
import java.awt.*;

public class Applet1 extends JApplet {
    public void init() {
        getContentPane().add(new JLabel("Applet!"));
    }
} ///:~
```

Забележете че аплетите не е необходимо да имат **main()**. Всичко необходимо е вградено в рамката; слагате всички код за начално установяване в **init()**.

Пускане на аплети от Web браузър

За да се стартира тази програма трябва да я сложите в Web страница и да я разгледате с вашия Web броузър, в който Java е активиран. За да сложите аплет в Web страница слагате специален текст в HTML корса на Web страницата³ за да се каже на страницата как да натовари и пусне аплета.

Този процес беше много простиčък когато Java беше простиčък и всеки се намираше в същото купе и слагаше една и съща поддръжка на Java в броузера си. Така че може би ще се разминете с много простиčък текст HTML в Web страницата, като този:

```
<applet code=Applet1 width=100 height=50>
</applet>
```

После дойдоха браузърските и езикови войни, а всички ние загубихме. След доста време Javasoft разбра че вече не може да се разчита браузърите да поддържат еднакво добре Java, а единственото спасение е добавянето на нещо което отчита механизма на конкретния браузър. Чрез използване на механизма на разширенията (които продавачът на браузъра не може да деактивира – в стремежа да направи състезателен напредък – без да съсипе всичките разширения от трети доставчици) Javasoft гарантира че Java не може да бъде разказан от браузър от антагонистично настроен доставчик.

При Internet Explorer механизъмът на разширяването е ActiveX управление, а при Netscape механизъмът е plug-in. Във вашия html код трябва да сложите текст за поддръжка и на двата. Ето как изглежда най-простата HTML страница за **Applet1**:

³ It is assumed that the reader is familiar with the basics of HTML. It's not too hard to figure out, and there are lots of books and resources.

```
//:! c13:Applet1.html
<html><head><title>Applet1</title></head><hr>
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
    width="100" height="50" align="baseline"
    codebase="http://java.sun.com/products/plugin/1.2.2/jinstall-1_2_2-win.cab#Version=1,2,2,0">
<PARAM NAME="code" VALUE="Applet1.class">
<PARAM NAME="codebase" VALUE=".>
<PARAM NAME="type" VALUE="application/x-java-applet;version=1.2.2">
<COMMENT>
<EMBED type="application/x-java-applet;version=1.2.2" width="200"
    height="200" align="baseline" code="Applet1.class"
    codebase=".>
    pluginspage="http://java.sun.com/products/plugin/1.2/plugin-install.html">
<NOEMBED>
</COMMENT>
    No Java 2 support for APPLET!!
</NOEMBED></EMBED>
</OBJECT>
<hr></body></html>
///:~
```

Някои от резовете бяха твърде дълги и трябваше да се отрежат според страницата. Кодът във версията която разтоварвате (от www.BruceEckel.com) ще работи без да се грижите за това.

Стойността **code** дава името на **.class** файла в който е аплетът. **width** и **height** определят началните размери на аплета (в пиксели, както преди). Има други неща които може да сложите в реда за аплета: място за намиране на други **.class** файлове в Internet (**codebase**), информация за подравняване (**align**), специален идентификатор който позволява на аплетите да комуникират един с друг (**name**) и на програмистите да дават информация която аплетът може да използва. Параметрите имат формата

```
<param name="identifier" value = "information">
```

и може да бъдат толкова на брой колкото е необходимо.

Автоматична генерация на HTML файлове

Има доста аплети в тази глава и би било добре да има и средство за автоматично писане на HTML файл който ги използва всичките, така че лесно да може да ги видите всичките на една страница но без да пишете всичко ръчно. В тази точка на книгата вече имаме технологията за написване на такава програма:

Използване на Appletviewer

JDK на Sun (бесплатно може да го свалите от техния сайт) съдържа инструмент наречен **Appletviewer** който сочи **<applet>** таговете в HTML файла и песка аплетите без да изобразява съпровождащия HTML текст. Понеже той игнорира всичко освен APPLET таговете, може да сложите такива тагове в Java сурса като коментари:

```
// <applet code=MyApplet width=200 height=100>
// </applet>
```

По този начин може да стартирате "**Appletviewer MyApplet.java**" и няма нужда да създавате мънички HTML файлове за нуждите на тестовете. Също може да видите от примера че HTML тагът трябва да бъде вмъкнат в Web страница.

Например може да създадете изкоментирани HTML тагове в **Applet1.java**:

```
//: c13:Applet1b.java
```

```
// Embedding the applet tag for Appletviewer
// <applet code=Applet1b width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;

public class Applet1b extends JApplet {
    public void init() {
        getContentPane().add(new JLabel("Applet!"));
    }
} ///:~
```

Сега може да извикате аплета с команда

```
| Appletviewer Applet1b.java
```

За "простите" аплети в тази книга ще се използва простата форма за лесно тестване (покъсно ще видите друг начин за кодиране който ще позволи да стартирате аплетите от командния ред без използване на Appletviewer).

Тестване на аплети

Може да направите прост тест без да имате каквато и да било връзка с мрежа чрез пускането на вашия Web браузър и отварянето на HTML файла съдържащ аплетския таг. Като се натовари HTML файла браузерът ще открие тага и ще започне лов на **.class** определен от стойността **code**. Разбира се, той гледа CLASSPATH за да намери мястото на лова, а ако вашият **.class** не е на CLASSPATH ще издаде съобщение за грешка на статус-линията - че не е намерил **.class** файла.

Когато поискате да изprobвате това на вашия Web сайт нещата стават мъничко по-сложни. Преди всичко, трябва да имате Web сайт, което за повечето хора значи Internet Service Provider (ISP) на отдалечената машина (понеже аплетът е файл или множество от файлове, ISP-то няма нужда да дава никаква специална поддръжка за Java). После трябва да имате начин да закарате HTML файловете и **.class** файловете в правилната директория (вашата WWW директория) на машината на ISP. Това типично се прави с File Transfer Protocol (FTP) програма, каквито има множество достъпни бесплатно или като шеъруеър. Така че сякаш всичко което трябва да се направи е да закарате с FTP файловете на машината на ISP, после да се свържете със сайта и HTML чрез вашия браузър; ако аплетът идва и работи, тогава всичко е наред, нали?

Ето тук може да се избудалкате. Ако браузерът на клиентската машина не може да намери **.class** файла на сървъра, той ще търси на CLASSPATH на вашата локална машина. Така аплетът може и да не се е натоварил правилно, но всичко изглежда добре поради наличието на файловете на вашата машина. Когато някой друг се включи, обаче, неговият броузър не може да намери файловете. Така че когато тествате осигурете изтриването на **.class** файловете които трябва на вашата машина за да бъде сигурно.

Едно от най-коварните места където ми се случи такова нещо беше когато аз невинно сложих аплета вътре в **package**. След слагането на HTML файла и аплета на сървъра излезе, че пътят на сървъра до файла се обърква от името на пакета. Обаче моят браузър намираше всичко на CLASSPATH. Така че аз бях единствения, който можеше вярно да зареди аплета. Отне време докато открия, че **package** операторът беше причината. Изобщо, ще махнете **package** оператора от аплетите.

Правене на бутона

Правенето на бутона е доста просто: просто викате конструктора на **Button** с етикета, който искате да е на бутона. (Може също да използвате конструктор по подразбиране, ако не искате надпис на бутона, но това не е много полезно.) Обикновено ще искате да създадете манипулятор към бутона за да може да работите с него по-късно.

Button е компонент, както неговото собствено малко прозорче, което автоматично ще де прерисува при осъвременяване. Това значи че не изобразявате явно бутона или някакъв друг управляващ елемент; просто ги слагате във формата и ти оставяте да се грижат за висичките боядисване сами. Така че за слагане на бутона във формата подтискате **init()** вместо **paint()**:

```
//: c13:Button1.java
// Putting buttons on an applet
// <applet code=Button1 width=200 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;

public class Button1 extends JApplet {
    JButton
    b1 = new JButton("Button 1"),
    b2 = new JButton("Button 2");
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(b1);
        cp.add(b2);
    }
} //:~
```

Не е достатъчно да се създае **Button** (или друго такова нещо). Трябва също да извикате метода **Applet add()** за да се сложи той на формата на аплета. Това изглежда много по-просто отколкото е, понеже извикването на **add()** фактически решава, неявно, къде точно да сложи бутона. Управлението на разположението на елементите е прегледано накъсо

Хващане на събитие

Ще забележите, че ако се компилира и пусне горния аплет нищо не се случва като натиснете бутона. Ето тук трябва да се намесите и да напишете код, който да определя какво трябва да се случи. Основата на движението от събития програми, което е много от съдържанието на GUI, привързва събитията към код, който се изпълнява с тяхното настъпване.

С наученото до тук вероятно ще предположите, че има нещо обектно-ориентирано и в тази работа. Например да би могло да се наследи бутона и да се подтисне метъодът "натиснат бутона" (това, както се оказа, е твърде досадно и ограничаващо). Може също да очаквате че има някакъв главен "събитиен" който съдържа метод за всяко събитие на което искате да се реагира.

Преди обектите типичният подход за обработка на събития беше "гигантски switch оператор." Всяко събитие ще има уникален цял номер и в главния цикъл ще сложите превключващ оператор **switch** за въпросната стойност.

Swing в Java 1.0 не използва нищо ОО. Нито пък използва огромен **switch** оператор който използва присвояване на цели числа към събитията. Вместо това трябва да създадете каскадно множество от **if** оператори. С **if** операторите се опитвате да откриете кой обект е

цел на събитието. Тоест ако натиснете бутон конкретен бутон е целта. Нормално това е всичко за което се грижите – ако бутона е цел на събитието, най-вероятно е станало щракване с мишката и в повечето случаи може да се осланяте на това предположение. Обаче събитията може да съдържат и друга информация. Например ако искате да намерите местоположението на пиксела където е станало кликането, така че да може да начертаете линия до него място, обектът **Event** ще съдържа мястото. (Трябва също да сте предупредени че компонентите на Java 1.0 могат да бъдат ограничени във възможните видове генериирани събития, докато Java 1.1 и Swing/JFC компонентите дават пълно множество от събития.)

AWT методът в Java 1.0 където се намират каскадираните **if** оператори се нарича **action()**. Макар че целият Java 1.0 Event модел се счита за остатъл в Java 1.1, все още той широко се използва за прости аплети и системи които не поддържат Java 1.1, така че препоръчвам да го разучите, включително използването на следващия подход с **action()** метода.

action() има два аргумента: първият е от тип **Event** и съдържа всичката информация за събитието кето е довело до това извикване на **action()**. Например това може да е кликане с мишката, обикновено натискане или отпускане на клавиши, такова на специален клавиши, фактът че компонент е взел или закубил фокуса, премествания на мишката, или придвижвания и т.н. Вторият аргумент е обикновено целта на събитието, която често ще игнорирате. Вторият аргумент също е капсулиран в **Event** обект така че се повтаря като аргумент.

Ситуациите където се вика **action()** са крайно ограничени: Когато слагате управляващи органи на екранчето, някой видове контроли (бутони, маркиращи кутийки, падащи списъци, менюта) имат "стандартна реакция" която се случва, та извика **action()** със съответния **Event** обект. Например с бутон методът **action()** се вика когато бутона се натисне и никога в друг случай. Обикновено това е много добре, понеже точно това търсите от бутона. Възможно е обаче да работите с много други видове събития чрез метода **handleEvent()** както ще видите по-късно в тази глава.

Предишния пример може да се разшири да поддържа и натисканията на бутон така:

```
//: c13:Button2.java
// Capturing button presses
// <applet code=Button2 width=200 height=50>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class Button2 extends JApplet {
    JButton
    b1 = new JButton("Button 1"),
    b2 = new JButton("Button 2");
    class BL implements ActionListener {
        public void actionPerformed(ActionEvent e){
            String name =
                ((JButton)e.getSource()).getText();
            getAppletContext().showStatus(name);
        }
    }
    BL al = new BL();
    public void init() {
        b1.addActionListener(al);
        b2.addActionListener(al);
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(b1);
        cp.add(b2);
```

```
    }
} ///:~
```

За да се види какъв е целевия обект трябва да се провери члена **target** на **Event** и да се използва **equals()** метода за проверка дали съвпада с целевия манипулятор който ви интересува. Когато напишете поддръжка за всички обекти които ви интересуват трябва да извикате **super.action(evt, arg)** в **else** оператора накрая, както е показано по-горе. Припомните си от глава 7 (полиморфизъм) че вашия подтиснат метод се вика наместо версията от базовия клас. Обаче последната съдържа код за обработка на всичко, от което вие не се интересувате, та този код няма да се изпълни, ако не го извикате явно. Връщаната стойност показва дали е обработено или не, така че ако си намерите събитието ще връщате **true**, иначе това, което връща **event()** от базовия клас.

За този пример най-простата акция е да се изведе съобщение, че е натиснат бутон. Някои системи дават възможност да изскочи малко прозорче със съобщението в него, но аплетите не поощряват това. Може обаче да сложите съобщението в долния край на прозореца на Web браузера на статус реда чрез извикване на метода **getAppletContext()** на **Applet** за да се получи достъп до браузъра и после **showStatus()** за да се сложи стринг на статус реда.⁴ Може да изведете пълно описание на събитието по същия начин, чрез **getAppletContext().showStatus(evt + "")**. (Празният **String** кара компилатора да превърне **evt** в **String**.) И двете извеждания са наистина ценни само за тестване и дебъгване понеже браузърът може да напише друго върху вашето съобщение.

Често е по-удобно да се кодира **ActionListener** като анонимен вътрешен клас, особено ако има тенденция да бъде използван единствен екземпляр от всеки клас-слушател (на събития - б.пр.). **Button2.java** може да се промени за да използва анонимен вътрешен клас както следва:

```
//: c13:Button2b.java
// Using anonymous inner classes
// <applet code=Button2b width=200 height=50>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class Button2b extends JApplet {
    JButton
        b1 = new JButton("Button 1"),
        b2 = new JButton("Button 2");
    ActionListener al = new ActionListener() {
        public void actionPerformed(ActionEvent e){
            String name =
                ((JButton)e.getSource()).getText();
            getAppletContext().showStatus(name);
        }
    };
    public void init() {
        b1.addActionListener(al);
        b2.addActionListener(al);
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(b1);
        cp.add(b2);
    }
} ///:~
```

⁴ **ShowStatus()** is also a method of Applet, so you can call it directly, without calling **getAppletContext()**.

Текстови полета

Едно **TextField** е един ред, който позволява на потребителя да въвежда и редактира текст. **TextField** е наследено от **TextComponent**, което позволява да се селектира текста, да се вземе като **String**, да се зададе (въведе), а също да се зададе дали **TextField** ще може да се редактира, заедно с други методи, които може да намерите във вашата он-лайн документация. Следния пример демонстрира някаква част от функционалността на **TextField**; може да се убедите че имената на методите са доста очевидни:

```
//: c13:TextField1.java
// Using the text field control
// <applet code=TextField1 width=350 height=75>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class TextField1 extends JApplet {
    JButton
        b1 = new JButton("Get Text"),
        b2 = new JButton("Set Text");
    JTextField
        t = new JTextField("Starting text: ", 30);
    String s = new String();
    ActionListener a1 = new ActionListener() {
        public void actionPerformed(ActionEvent e){
            getAppletContext().showStatus(t.getText());
            s = t.getSelectedText();
            if(s == null)
                s = t.getText();
            t.setEditable(true);
        }
    };
    ActionListener a2 = new ActionListener() {
        public void actionPerformed(ActionEvent e){
            t.setText("Inserted by Button 2: " + s);
            t.setEditable(false);
        }
    };
    public void init() {
        b1.addActionListener(a1);
        b2.addActionListener(a2);
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(b1);
        cp.add(b2);
        cp.add(t);
    }
} ///:~
```

Има няколко начина да се конструира **TextField**; този показан тук дава начален стринг и задава дължината на полето в знаци.

Натискането на бутон 1 взема текста селектиран с мишката или целия текст и го дава на **String s**. Позволява също полето да бъде редактирано. Натискането на бутон 2 пуска съобщение и **s** в текстовото поле и предотвратява редактируемостта на полето (макар и все още да може да се селектира текста). Редактируемостта на текста се управлява с даването на **setEditable()** или **true** или **false**.

Текстови площи

TextArea е подобно на **TextField** с тази разлика, че допуска много редове и има значително по-голяма функционалност. Към това което може да се прави с **TextField** се добавя добавянето на текст и вмъкването на текст в определена точка. Сякаш тази функционалност би била полезна и за **TextField** също, та е малко смущаващо да се опита да се разбере как е направена разликата. Би могло да се помисли че ако искате функционалността на **TextArea** навсякъде може просто да използвате едноредов **TextArea** на местата, където иначе бихте използвали **TextField**. В Java 1.0 също получавате и скрол-инструменти **TextArea** даже ако не са уместни; тоест получавате и хоризонтална и вертикална скролна лента даже и за едноредов **TextArea**. В Java 1.1 това е преодоляно с допълнителен конструктор, който позволява да се избере кои скролбарове (ако въобще има) ще са представени. Следният пример показва само поведението при Java 1.0 където винаги има скролбарове. По-късно в главата ще видите пример който демонстрира как е при Java 1.1 **TextArea**тата.

```
//: c13:TextArea1.java
// Using the text area control
// <applet code=TextArea1 width=350 height=200>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class TextArea1 extends JApplet {
    JButton
    b1 = new JButton("Text Area 1"),
    b2 = new JButton("Text Area 2"),
    b3 = new JButton("Replace Text"),
    b4 = new JButton("Insert Text");
    JTextArea
    t1 = new JTextArea("t1", 1, 30),
    t2 = new JTextArea("t2", 4, 30);
    ActionListener a1 = new ActionListener() {
        public void actionPerformed(ActionEvent e){
            getAppletContext().showStatus(t1.getText());
        }
    };
    ActionListener a2 = new ActionListener() {
        public void actionPerformed(ActionEvent e){
            t2.setText("Inserted by Button 2");
            t2.append(": " + t1.getText());
            getAppletContext().showStatus(t2.getText());
        }
    };
    ActionListener a3 = new ActionListener() {
        public void actionPerformed(ActionEvent e){
            String s = " Replacement ";
            t2.replaceRange(s, 3, 3 + s.length());
        }
    };
}
```

```

};

ActionListener a4 = new ActionListener() {
    public void actionPerformed(ActionEvent e){
        t2.insert(" Inserted ", 10);
    }
};

public void init() {
    b1.addActionListener(a1);
    b2.addActionListener(a2);
    b3.addActionListener(a3);
    b4.addActionListener(a4);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(b1);
    cp.add(t1);
    cp.add(b2);
    cp.add(t2);
    cp.add(b3);
    cp.add(b4);
}
} //:~

```

Има НЯКОЛКО конструктора на **TextArea**, но показаният тук дава началния стринг и броя на редовете и колоните. Различните бутони показват вземането, заместването, добавянето и вмъкването на текст.

ЕТИКЕТИ

JLabel прави точно това, което показва името: слага етикет на формата. Това е особено полезно за текстови полета които нямат собствени етикети, а също може да бъде полезно ако просто искате да сложите текстова информация във формата. Можете, както е показано в първия пример от тази глава, да използвате **drawString()** вътре в **paint()** за да сложите текста на точното място. Когато използвате **JLabel** можете (приблизително) да асоциирате текста с някой друг компонент чрез управителя на формата (който ще се дискутира по-късно в тази глава).

С конструктора може да създадете празен етикет или с първоначален текст в него (последното ще е по-типично). По-малко използваните възможности включват поставяне на подравняването на етикетите. Може също да сменяте текста на етикета с **setText()** и да четете стойността с **getText()**. Този пример показва общоупотребяваните неща които може да се правят:

```

//: c13:Label1.java
// Using labels
// <applet code=Label1 width=200 height=100>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class Label1 extends JApplet {
    JTextField t1 = new JTextField("t1", 10);
    JLabel
    labl1 = new JLabel("TextField t1"),
    labl2 = new JLabel("      ");
    JButton

```

```

b1 = new JButton("Test 1");
b2 = new JButton("Test 2");
ActionListener a1 = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        t1.setText("Button 1");
        lbl2.setText("Text set into Label");
    }
};
ActionListener a2 = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        t1.setText("Button 2");
        lbl1.setText("Hello");
    }
};
public void init() {
    b1.addActionListener(a1);
    b2.addActionListener(a2);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(lbl1);
    cp.add(t1);
    cp.add(b1);
    cp.add(lbl2);
    cp.add(b2);
}
} //:~

```

Първият начин на използване е най-типичният: етикетиране на **TextField** или **TextArea**. Във втората част на примера се резервират много празни места и когато натиснете "Test 1" бутона **setText()** се използва за вмъкване на текст в полето.

ОТМЕТКИ

Отметката дава възможност да се направи единичен избор включено-изключено; състои се от мъничка кутийка и етикет. Кутийката типично съдържа малко 'x' (или друг индикатор че е "вкллючено") или е пътна в зависимост кое от двете състояния е налице.

Нормално ще създавате **Checkbox** чрез конструктор който взема етикет като аргумент. Може да четете и поставяте състоянието, а също и етикета ако искате да го променяте след като **Checkbox** е бил вече създаден. Забележете че капитализацията на **Checkbox** е несъстоятелна заедно с другите контроли, което може да ви изненада ако очаквахте да е "CheckBox."

Щом **Checkbox** е създаден или поставен възниква събитие, което може да се хване по същия начин както за бутона. Следния пример използва **TextArea** за пронумероване на всички отметки които са включени:

```

//: c13:CheckBox1.java
// Using check boxes
// <applet code=CheckBox1 width=200 height=200>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class CheckBox1 extends JApplet {

```

```

JTextArea t = new JTextArea(6, 15);
JCheckBox
cb1 = new JCheckBox("Check Box 1"),
cb2 = new JCheckBox("Check Box 2"),
cb3 = new JCheckBox("Check Box 3");
public void init() {
    cb1.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
            trace("1", cb1);
        }
    });
    cb2.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
            trace("2", cb2);
        }
    });
    cb3.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
            trace("3", cb3);
        }
    });
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(new JScrollPane(t));
    cp.add(cb1);
    cp.add(cb2);
    cp.add(cb3);
}
void trace(String b, JCheckBox cb) {
    if(cb.isSelected())
        t.append("Box " + b + " Set\n");
    else
        t.append("Box " + b + " Cleared\n");
}
} //:~

```

Забележете че в този пример анонимният вътрешен клас който реализира **ActionListener** се създава инлайн, така че не се използва междинна променлива.

Методът **trace()** изпраща името на избрания **Checkbox** и текущото му състояние на **TextArea** чрез **appendText()** така че ще видите сумарен списък на отметките и тяхното състояние.

Радиобуто ни

Концепцията за радиобуто ните в програмирането на GUI идва от преди-електронната ера на автомобилните радиа с механични буто ни: като натиснете един, всеки друг ако е бил натиснат изскуча. Така може да се направи един избор между много възможни.

Swing няма отделен клас за представяне на радиобуто на; вместо това се използва повторно кода за **Checkbox**. За да се сложи обаче **Checkbox** в радиобуто на група (и за да се промени видът му така че наглед да е различен от **Checkbox**) трябва да използвате специален конструктор който взема **CheckboxGroup** обект като аргумент. (Може също да извикате **setCheckboxGroup()** след като **Checkbox** е бил създаден.)

CheckboxGroup няма аргумент на конструктора; единствената причина за съществуването му е да се съберат няколко **Checkbox** в група от радиобуто ни. Един от **Checkbox** обектите

трябва да има поставено състояние **true** преди да се опитате да изобразите групата радиобутони; иначе ще се получи изключение по време на изпълнение. Ако сложите повече от един бутон да бъде **true** само последният ще остане **true**.

Ето прост пример за използването на радиобутони. Забележете че събитията с радиобутоните се хващат като всички други:

```
//: c13:RadioButton1.java
// Using radio buttons
// <applet code=RadioButton1
// width=200 height=75> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class RadioButton1 extends JApplet {
    JTextField t =
        new JTextField("Radio button 2", 15);
    ButtonGroup g = new ButtonGroup();
    JRadioButton
        rb1 = new JRadioButton("one", false),
        rb2 = new JRadioButton("two", true),
        rb3 = new JRadioButton("three", false);
    ActionListener al = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            t.setText("Radio button " +
                ((JRadioButton)e.getSource()).getText());
        }
    };
    public void init() {
        rb1.addActionListener(al);
        rb2.addActionListener(al);
        rb3.addActionListener(al);
        g.add(rb1); g.add(rb2); g.add(rb3);
        t.setEditable(false);
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(t);
        cp.add(rb1);
        cp.add(rb2);
        cp.add(rb3);
    }
} ///:~
```

Текстово поле се използва за изобразяване на състоянието. Полето е сложено да бъде нередактируемо понеже се използва само за изобразяване, не за събиране на данни. Това е показано като алтернатива на използването на **Label**. Забележете че текстът на полето е инициализиран като "Radio button two" понеже това е първоначално избрания радиобутон.

Може да има произволно много **CheckboxGroup**и в една форма.

Падащи списъци

Като група от радиобутони и падащия списък е начин да се накара потребителят да избере една и само една възможност от няколко. Това обаче е много по-сбит начин да се свърши

работата и е по-лесно да се сменят възможностите без изненади за потребителя. (Може да променяте радиобутоните динамично, но това има тенденция да дразни визуално).

Choice боксът на Java не е като combo box в Windows, който позволява да се избере от списък или да се въведе ваш собствен избор. С **Choice** кутията избирате един и само един елемент от списъка. В следния пример **Choice** боксът започва с някакъв брой елементи и после се добавяват нови когато се натисне бутон. Това позволява да се наблюдават някои интересни страни от поведението на **Choice** боксовете:

```
//: c13:Choice1.java
// Using drop-down lists
// <applet code=Choice1
// width=200 height=100> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class Choice1 extends JApplet {
    String[] description = { "Ebullient", "Obtuse",
        "Recalcitrant", "Brilliant", "Somnescient",
        "Timorous", "Florid", "Putrescent" };
    JTextField t = new JTextField(15);
    JComboBox c = new JComboBox();
    JButton b = new JButton("Add items");
    int count = 0;
    public void init() {
        for(int i = 0; i < 4; i++)
            c.addItem(description[count++]);
        t.setEditable(false);
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e){
                if(count < description.length)
                    c.addItem(description[count++]);
            }
        });
        c.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e){
                t.setText("index: " + c.getSelectedIndex()
                    + " " + ((JComboBox)e.getSource())
                    .getSelectedItem());
            }
        });
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(t);
        cp.add(c);
        cp.add(b);
    }
} ///:~
```

TextField изобразява “избрания индекс,” който е номерът на избрания елемент, както и **String** представянето на втория аргумент на **action()**, който в този случай е избраният стринг.

Когато пуснете този аплет, обърнете внимание на определянето на дължината на **Choice** бокса: в Windows дължината е фиксирана от първото падане на списъка. Това значи че ако “паднете” списъка и после добавите елементи, те ще са там но списъкът никак няма да се

удълже⁵ (може да се движките насам-нанатък по елементите). Ако обаче добавите всичко преди първото падане на списъка, той ще бъде с коректната дължина. Разбира се, потребителят очаква да види целия списък когато е паднал, така че това поведение слага значителни ограничения на добавянето на елементи в **Choice** боксовете.

Списъчни кутии

Те различават значително от **Choice** боксовете, и то не само по изглед. Докато **Choice** боксът пада при активирането си, **List** заема фиксиран брой редове на екрана през цялото време. Освен това **List** позволява многократна селекция: ако кликнете върху повече от един елемент всеки остава осветен и може да изберете колкото искате. Ако искате да видите елементите на списъка, просто викате **getSelectedItems()**, което дава масив от **String** от избрани елементи. За да махнете елемент от грипа трябва да го кликнете пак.

Един проблем с **List** е че подразбиращата се акция е двойно кликване, не единичното. Единичното кликване добавя и премахва елементи от избраната група, а двойното вика **action()**. Един начин да се заобиколи това е да се преобразова вашият потребител, което предположение е направено в следната програма:

```
//: c13:List1.java
// Using lists
// <applet code=List1>
// width=200 height=350> </applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;

public class List1 extends JApplet {
    String[] flavors = { "Chocolate", "Strawberry",
        "Vanilla Fudge Swirl", "Mint Chip",
        "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie" };
    JList list = new JList(flavors);
    JTextArea t =
        new JTextArea(flavors.length + 1, 15);
    public void init() {
        t.setEditable(false);
        list.addListSelectionListener(
            new ListSelectionListener() {
                public void
                valueChanged(ListSelectionEvent e) {
                    t.setText(""); // Erase the text area
                    Object[] items= list.getSelectedValues();
                    for(int i = 0; i < items.length; i++)
                        t.append(items[i] + "\n");
                }
            });
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(t);
        cp.add(list);
    }
} ///:~
```

⁵ This behavior is apparently a bug and will be fixed in a later version of Java.

Когато натиснете бутона той даваля елементи отгоре на списъка (поради втория аргумент 0 на **addItem()**). Добавянето на елементи към **List** е по-подходящо отколкото към **Choice** бокс понеже потребителят очаква да се движки нагоре-надолу по списъчната кутия (най-малкото, тя има вградена лента за това) но не очаква да трябва да намира как да го прави с падащия списък, както в предишния пример.

Обаче единствения начин да се извика **action()** е чрез двойно кликане. Ако трябва да следите други активности на потребителя над вашия **List** (в частност единичните кликвания) трябва да приемете алтернативен подход.

Пана с ушички

JTabbedPane позволява да се създаде диалог с "ушенца," излизящи от един ръб, та като кликнете ушенце излиза съответния диалог.

```
//: c13:TabbedPane1.java
// Demonstrates the Tabbed Pane
// <applet code=TabbedPane1
// width=350 height=200> </applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;

public class TabbedPane1 extends JApplet {
    String[] flavors = { "Chocolate", "Strawberry",
        "Vanilla Fudge Swirl", "Mint Chip",
        "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie" };
    JTabbedPane tabs = new JTabbedPane();
    public void init() {
        for(int i = 0; i < flavors.length; i++)
            tabs.addTab(flavors[i],
                new JButton("Tabbed pane " + i));
        tabs.addChangeListener(new ChangeListener(){
            public void stateChanged(ChangeEvent e) {
                getAppletContext().showStatus(
                    "Tab selected: " +
                    tabs.getSelectedIndex());
            }
        });
        Container cp = getContentPane();
        cp.setLayout(new BorderLayout());
        cp.add(tabs);
    }
} //:~
```

Аплетът е промене да използва **BorderLayout** вместо подразбиращия се **FlowLayout** (вярно ли е това????).

В Java използването на някакъв вид "tabbed panel" механизъм е доста важно понеже (както ще видите по-късно) в програмирането на аплети използването на изскачащи диалози силно се обезкуражава.

Кутии за съобщения

Средите с прозорци често съдържат стандартно множество от кутии за съобщения които позволяват бързо да се изпрати информация на потребителя или да се вземе информация от потребителя. В Swing тези кутии се съдържат в **JOptionPane**. Има много различни възможности (някои много усложнени), но най-често използваните ще са вероятно диалогът със съобщения, диалогът за потвърждаване, викан чрез **static JOptionPane.showMessageDialog()** и **JOptionPane.showConfirmDialog()**.

```
//: c13:MessageBoxes.java
// Demonstrates JOptionPane
// <applet code=MessageBoxes
// width=200 height=100> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class MessageBoxes extends JApplet {
    JButton b() = { new JButton("alert"),
        new JButton("yes/no"), new JButton("Color"),
        new JButton("input"), new JButton("3 vals")
    };
    ActionListener al = new ActionListener() {
        public void actionPerformed(ActionEvent e){
            String id =
                ((JButton)e.getSource()).getText();
            if(id.equals("alert"))
                JOptionPane.showMessageDialog(null,
                    "There's a bug on you!", "Hey!",
                    JOptionPane.ERROR_MESSAGE);
            else if(id.equals("yes/no"))
                JOptionPane.showConfirmDialog(null,
                    "or no", "choose yes",
                    JOptionPane.YES_NO_OPTION);
            else if(id.equals("Color")){
                Object[] options = { "Red", "Green" };
                int sel = JOptionPane.showOptionDialog(
                    null, "Choose a Color!", "Warning",
                    JOptionPane.DEFAULT_OPTION,
                    JOptionPane.WARNING_MESSAGE, null,
                    options, options(0));
                if(sel != JOptionPane.CLOSED_OPTION)
                    getAppletContext().showStatus(
                        "Color Selected: " + options(sel));
            } else if(id.equals("input")){
                String val = JOptionPane.showInputDialog(
                    "How many fingers do you see?");
                getAppletContext().showStatus(val);
            } else if(id.equals("3 vals")){
                Object[] selections = {
                    "First", "Second", "Third" };
                Object val = JOptionPane.showInputDialog(
                    null, "Choose one", "Input",
                    JOptionPane.INFORMATION_MESSAGE,
                    null, selections, selections(0));
                if(val != null)

```

```

        getAppletContext().showStatus(
            val.toString());
    }
}
};

public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < b.length; i++) {
        b(i).addActionListener(al);
        cp.add(b(i));
    }
}
} ///:~

```

Менюта

```

//: c13:SimpleMenus.java
// <applet code=SimpleMenus
// width=200 height=75> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class SimpleMenus extends JApplet {
    JTextField t =
        new JTextField(15);
    ActionListener al = new ActionListener() {
        public void actionPerformed(ActionEvent e){
            t.setText(
                ((JMenuItem)e.getSource()).getText());
        }
    };
    JMenu menus() = { new JMenu("Winken"),
        new JMenu("Blinken"), new JMenu("Nod") };
    JMenuItem items() = {
        new JMenuItem("Fee"), new JMenuItem("Fi"),
        new JMenuItem("Fo"), new JMenuItem("Zip"),
        new JMenuItem("Zap"), new JMenuItem("Zot"),
        new JMenuItem("Olly"), new JMenuItem("Oxen"),
        new JMenuItem("Free") };
    public void init() {
        JMenuBar mb = new JMenuBar();
        setJMenuBar(mb);
        for(int i = 0; i < items.length; i++) {
            items(i).addActionListener(al);
            menus(i%3).add(items(i));
        }
        for(int i = 0; i < menus.length; i++)
            mb.add(menus(i));
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(t);
    }
}

```

```
| } ///:~
```

Диалогови кутии

```
//: c13:DialogDemo.java
// Creating and using Dialog Boxes
// <applet code=DialogDemo width=125 height=50>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class MyDialog extends JDialog {
    public MyDialog(JFrame parent) {
        super(parent, "My dialog", true);
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(new JLabel("Here is my dialog"));
        JButton ok = new JButton("OK");
        ok.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e){
                setVisible(false); // Closes the dialog
            }
        });
        cp.add(ok);
        setSize(150,125);
    }
}

public class DialogDemo extends JApplet {
    JButton b1 = new JButton("Dialog Box");
    MyDialog dlg = new MyDialog(null);
    public void init() {
        b1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e){
                dlg.show();
            }
        });
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(b1);
    }
} ///:~
```

Ако пуснете тези програми, ще видите че всичко което изскача от аплета, включително диалоговите кутии, е "неползващо се с доверие." Тоест получавате съобщение в прозореца, който е изскочил. Това е защото натеория може да се изльже обикновения потребител че работи с обикновено приложение и да се накара да напише номера на кредитната си карта, който после поема през Мрежата. Аплетът винаги е свързан с Web страница и визим с вашия браузър, докато диалогът не е и споменатото може да стане на теория. Като резултат не се среща често използването на диалози в аплет.

Управление на разположението

Начинът по който се разполагат компонентите в една форма в Java вероятно е различен от всички други GUI системи които сте използвали. Първо, всичко е код; няма "ресурси" които управляват разполагането на компонентите. Второ, разполагането се управлява от "управител по разполагането" който решава как ще легнат компонентите на основата на това как ги `add()`-вате. Дължината, вида и разполагането на компонентите ще бъдат забележително различни за различните управители. Освен това този управител приспособява размерите на вашия аплет или прозорец, така че ако те се променят (например в HTML спецификацията на аплета в страницата) дължината, вида и разполагането на компонентите може да се промени.

И **Applet** и **Frame** класовете са извлечени от **Container**, чиято задача е да съдържа и изобразява **Componenti**. (**Container** е **Component** така че също може да реагира на събития.) В **Container** има метод наречен **setLayout()** който позволява да се избере различен управител на подреждането.

В тази секция ще изследваме различни управители на разположението чрез слагане на бутони (понеже това е най-простото нещо). Няма да има никаква обработка на събитията от бутоните понеже целта е само да се покаже как лягат компонентите.

FlowLayout

До тук всички аплети които създадохме изглежда да разполагаха компонентите си по някаква вътрешна логика. Така е защото те използваха схема на разполагане по подразбиране: **FlowLayout**. Това просто "излива" компонентите във формата, от ляво надясно докато се запълни горната част, после слиза един ред надолу и продължава.

Ето пример който явно (излишно) слага управителя да бъде **FlowLayout** после слага бутони във формата. Ще видите че с **FlowLayout** компонентите вземат "натуналната" си дължина. Един **Button**, например, ще бъде с дължината на текста си.

```
//: c13:FlowLayout1.java
// Demonstrates the FlowLayout
// <applet code=FlowLayout1
// width=300 height=250> </applet>
import javax.swing.*;
import java.awt.*;

public class FlowLayout1 extends JApplet {
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        for(int i = 0; i < 20; i++)
            cp.add(new JButton("Button " + i));
    }
} //:~
```

Всички компоненти се довеждат до минималните им размери при **FlowLayout**, така че може да се получи и малко изненадващо поведение. Например един етикет ще има дължината на текста си, така че подравняването му в дясно не променя изгледа.

BorderLayout

Този поддържа концепцията за четири странични полета и централно такова. Когато добавяте нещо към панел използващ **BorderLayout** трябва да използвате **add()** който взема **String** обект за първи аргумент, който посочва (с правилна капитализация) "North" (горе), "South" (долу), "East" (дясно), "West" (ляво), или "Center." Ако ги сгрешите, евентуално и капитализацията, няма да стане грешка по време на компилацията, но аплетът няма да прави каквото очаквате. За щастие, както след малко ще видите, има много подобрен начин в Java 1.1.

Ето прост пример:

```
//: c13:BorderLayout1.java
// Demonstrates the BorderLayout
// <applet code=BorderLayout1
// width=300 height=250> </applet>
import javax.swing.*;
import java.awt.*;

public class BorderLayout1 extends JApplet {
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new BorderLayout());
        cp.add(BorderLayout.NORTH,
               new JButton("North"));
        cp.add(BorderLayout.SOUTH,
               new JButton("South"));
        cp.add(BorderLayout.EAST,
               new JButton("East"));
        cp.add(BorderLayout.WEST,
               new JButton("West"));
        cp.add(BorderLayout.CENTER,
               new JButton("Center"));
    }
} //:~
```

За всяко разположение освен "Center" елементът който слагате се намалява до минималния размер по едното измерение и се увеличава до максималния по другото. "Center," обаче, разширява и двата така, че да се заеме средата.

BorderLayout е подразбиращ се за приложенията и диалозите.

GridLayout

GridLayout подволява постройката на таблица от компоненти и като ги добавяте те се разполагат отляво-надясно и отгоре-надолу на мрежата. В конструктора определяте броя редове и колони които искате и те се разполагат равномерно.

```
//: c13:GridLayout1.java
// Demonstrates the GridLayout
// <applet code=GridLayout1
// width=300 height=250> </applet>
import javax.swing.*;
import java.awt.*;

public class GridLayout1 extends JApplet {
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new GridLayout(7,3));
```

```

    for(int i = 0; i < 20; i++)
        cp.add(new JButton("Button " + i));
}
} //:~

```

В този случай има 21 слота (места - б.пр.) но само 20 бутона. Последният слот е оставен празен; не се "балансира" при **GridLayout**.

GridLayout

Преди време се е вярвало, че планетите, слънцето и луната се въртят около земята. Интуитивно така изглежда при наблюдение. После астрономите станали по-изхитрени и започнали да следят движението на отделни обекти, някои от които изглеждало от време на време че се връщат назад по пътя си. Доколкото било известно че всичко се върти около земята, тези астрономи изразходвали много време да създават уравнения и теории обясняващи движението на звездните обекти.

Когато се опитвате да работите с **GridLayout** може да се чувствате като аналог на въпросните астрономи. Основната насока (поставена, доста интересно, от проектантите в "Sun") че всичко ще се прави в код. Коперниковата революция (отново мазно (буквално преведено - б.пр.) от ирония, откритието че планетите от слънчевата система се въртят около слънцието) е използването на ресурси за определяне на разположението и правене на програмистката работа лесна. Докато това бе добавено в Java, бяхте ограничени (за да продължат метафората) в Испанската Инквизиция на **GridLayout** и **GridBagConstraints**.

Препоръката ми е да се избягва **GridLayout**. Вместо това използвайте други управители и особено техниката за комбиниране на няколко в една програма. Вашите аплети няма да изглеждат толкова различни; във всеки случай не толкова че да оправдаят трудностите от **GridLayout**. Колкото до мене, твърде болезнено е да се направи пример за това (и аз не мога да окуражавам този начин на правене на библиотеки). Вместо това ще ви насоча към *Core Java 2* от Horstmann & Cornell (Prentice-Hall, 1999) за да започнете.

BoxLayout

Понеже толкова много хора имаха толкова много неприятности да се оправят с **GridLayout**, JavaSoft въведе **BoxLayout** в Java 2, което дава много от ползите от **GridLayout** без неприятностите. **BoxLayout** позволява да управлявате разположението или вертикално или хоризонтално и да управлявате мястото между тях викайки нещо като "Перчи се и лепи." Първо да погледнем използването на **BoxLayout** направо, както се използват всички други управители:

```

//: c13:BoxLayout1.java
// Vertical and horizontal BoxLayouts
// <applet code=BoxLayout1
// width=450 height=200> </applet>
import javax.swing.*;
import java.awt.*;

public class BoxLayout1 extends JApplet {
    public void init() {
        JPanel jpv = new JPanel();
        jpv.setLayout(
            new BoxLayout(jpv, BoxLayout.Y_AXIS));
        for(int i = 0; i < 5; i++)
            jpv.add(new JButton("" + i));
        JPanel jph = new JPanel();
        jph.setLayout(

```

```

        new BoxLayout(jph, BoxLayout.X_AXIS));
    for(int i = 0; i < 5; i++)
        jph.add(new JButton("'" + i));
    Container cp = getContentPane();
    cp.add(BorderLayout.EAST, jpv);
    cp.add(BorderLayout.SOUTH, jph);
}
} //://:~
```

Конструкторът на **BoxLayout** е малко различен от другите управители – давате **Container** който да се управлява от **BoxLayout** като първи аргумент и направление на управлението като втори.

За да се опростят нещата има специален контейнер наречен **Box** който използва **BoxLayout** като свой естествен управител. Следният пример разполага компонентите хоризонтално и вертикално чрез **Box**, който има два статични метода за създаване на кутии с вертикално и хоризонтално подравняване:

```

//: c13:Box1.java
// Vertical and horizontal BoxLayouts
// <applet code=Box1
// width=450 height=200> </applet>
import javax.swing.*;
import java.awt.*;

public class Box1 extends JApplet {
    public void init() {
        Box bv = Box.createVerticalBox();
        for(int i = 0; i < 5; i++)
            bv.add(new JButton("'" + i));
        Box bh = Box.createHorizontalBox();
        for(int i = 0; i < 5; i++)
            bh.add(new JButton("'" + i));
        Container cp = getContentPane();
        cp.add(BorderLayout.EAST, bv);
        cp.add(BorderLayout.SOUTH, bh);
    }
} //://:~
```

```

//: c13:Box2.java
// Adding struts
// <applet code=Box2
// width=450 height=300> </applet>
import javax.swing.*;
import java.awt.*;

public class Box2 extends JApplet {
    public void init() {
        Box bv = Box.createVerticalBox();
        for(int i = 0; i < 5; i++) {
            bv.add(new JButton("'" + i));
            bv.add(Box.createVerticalStrut(i*10));
        }
        Box bh = Box.createHorizontalBox();
        for(int i = 0; i < 5; i++) {
            bh.add(new JButton("'" + i));
            bh.add(Box.createHorizontalStrut(i*10));
        }
    }
}
```

```
    }
    Container cp = getContentPane();
    cp.add(BorderLayout.EAST, bv);
    cp.add(BorderLayout.SOUTH, bh);
}
} //:/~
```

```
//: c13:Box3.java
// Using Glue
// <applet code=Box3
// width=450 height=300> </applet>
import javax.swing.*;
import java.awt.*;

public class Box3 extends JApplet {
    public void init() {
        Box bv = Box.createVerticalBox();
        bv.add(new JLabel("Hello"));
        bv.add(Box.createVerticalGlue());
        bv.add(new JLabel("Applet"));
        bv.add(Box.createVerticalGlue());
        bv.add(new JLabel("World"));
        Box bh = Box.createHorizontalBox();
        bh.add(new JLabel("Hello"));
        bh.add(Box.createHorizontalGlue());
        bh.add(new JLabel("Applet"));
        bh.add(Box.createHorizontalGlue());
        bh.add(new JLabel("World"));
        bv.add(Box.createVerticalGlue());
        bv.add(bh);
        bv.add(Box.createVerticalGlue());
        getContentPane().add(bv);
    }
} //:/~
```

```
//: c13:Box4.java
// Rigid Areas are like pairs of struts
// <applet code=Box4
// width=450 height=300> </applet>
import javax.swing.*;
import java.awt.*;

public class Box4 extends JApplet {
    public void init() {
        Box bv = Box.createVerticalBox();
        bv.add(new JButton("Top"));
        bv.add(Box.createRigidArea(
            new Dimension(120, 90)));
        bv.add(new JButton("Bottom"));
        Box bh = Box.createHorizontalBox();
        bh.add(new JButton("Left"));
        bh.add(Box.createRigidArea(
            new Dimension(160, 80)));
        bh.add(new JButton("Right"));
    }
} //:/~
```

```

        bv.add(bh);
        getContentPane().add(bv);
    }
} //:~

```

Трябва да сте предупредени, че установените неща са противоречиви. Понеже се използват абсолютни стойности, някои хора считат че това създава повече неприятности отколкото изобщо се печели.

Алтернативи на `action`

Както бе отбелязано, `action()` не е единственият метод викан автоматично от `handleEvent()` щом сортира всичко за вас. Има три други множества методи които биват викани и ако искате да хванете някои типове събития (клавиатурни, мишкови и фокусни събития) всичко което трябва да направите е да подтиснете наличния метод. Тези методи са определени в базовия клас **Component**, така че са практически достъпни за всяка контроли, които може да сложите. Обаче трябва да сте предупредени, че този подход се счита остатъл в Java 1.1, та макар и да може да срещнете наследен код, трябва да използвате подхода на Java 1.1 (описан по-късно в главата) вместо въпросния.

Метод на компонент	Кога се вика
action (Event evt, Object what)	Когато стане "тиично" събитие за този компонент (например когато се натисне бутон или се избере елемент от падащ списък)
keyDown (Event evt, int key)	Натиснат е ключ когато този компонент е във фокус. Вторият аргумент е натиснатия клавиш и е излишно копиран в <code>evt.key</code> .
keyUp(Event evt, int key)	Отпуснат е клавиш когато този компонент има фокуса.
lostFocus(Event evt, Object what)	Фокусът се е отместил от целта. Нормално, <code>what</code> е излишно копиран от <code>evt.arg</code> .
gotFocus(Event evt, Object what)	Фокусът се е преместил в целта.
mouseDown(Event evt, int x, int y)	"Мишката долу" в координати <code>x, y</code> .
mouseUp(Event evt, int x, int y)	"Мишката горе" върху компонента
mouseMove(Event evt, int x, int y)	Мишката е мръднала докато е била върху компонента.
mouseDrag(Event evt, int x, int y)	Мишката се тегли след като се е случило <code>mouseDown</code> върху компонента. Всички събития на влечение се докладват на компонента, над който е станало <code>mouseDown</code> докато има <code>mouseUp</code> .
mouseEnter(Event evt, int x, int y)	Мишката е дошла върху

Метод на компонент	Кога се вика
	компонента (преди не е била там).
mouseExit(Event evt, int x, int y)	Мишката е заминала от компонента (преди е била върху него).

Може да се види, че всеки обект приема **Event** обект заедно с някаква информация от която типично се нуждаеме когато обработвате конкретната ситуация – за събитие с мишката, например, е вероятно да ви трябват координатите където се е случило събитието. Интересно е да се отбележи че когато **handleEvent()** на **Component** вика някой от тези методи (типичният случай), допълнителните аргументи винаги са излишни, понеже се съдържат в **Event** обекта. Ако погледнете сорса на **Component.handleEvent()** ще видите, че той експлицитно скубе допълнителните аргументи от **Event** обекта. (Това може да се счете неефикасно в някои програмни езици, но да напомним че фокусът в Java е на сигурността, не непременно на скоростта.)

За да видите сами, че тези събития са извиквани и като интересен експеримент си струва да напишете аплет който подтиска всичките методи от по-горе (освен за **action()**, който е подтиснат на много други места в тази глава) и който аплет извежда данни за всяко събитие като се случи.

Този пример също показва как да си създадете собствен бутон, понеже такъв се използва като цел на всички интересуващи ни събития. Бихте могли първо (естествено) да считате, че за да създадете собствен бутон трябва да наследите от **Button**. Това обаче не работи. Вместо него наследявате от **Canvas** (много по-родов компонент) и боядисвате бутона си чрез **paint()** метода. Както ще видите, твърде лошо е, че не може да се наследи **Button** (това не работи), понеже има част от кода която боядисва бутона. (Ако не ми вярвате, опитайте се да смените **Canvas** с **Button** в този пример и не забравяйте да извикате **super(label)** от базовия клас. Ще видите че бутона не се оцветява и събитията не се обработват.)

Класът **myButton** е специфичен: работи само с “бащиния прозорец” на **TrackEvent** (не базов клас, а прозорец в който този бутон е създаден и живее). С това знание, **myButton** може да стигне бащиния прозорец и да манипулира текстовите му полета, което е достатъчно да се пише информация в статус-линията на бащиния прозорец. Разбира се това е много по-ограничено решение, понеже **myButton** може да се използва само заедно с **TrackEvent**. Този вид код понякога се нарича “много свързан.” Обаче за да се направи **myButton** по-общ трябва много повече усилие което не е оправдано в този пример (и възможно за много от аплетите, които ще пишете). Отново, помнете че следният код използва APIта които са останали в Java 1.1.

```
//: c13:TrackEvent.java
// Show events as they happen
// <applet code=TrackEvent
// width=700 height=500></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

class MyButton extends JButton {
    HashMap h;
    MyButton(TrackEvent parent,
    Color color, String label) {
        super(label);
        h = parent.h;
        setBackground(color);
```

```

        addFocusListener(fl);
        addKeyListener(kl);
        addMouseListener(ml);
        addMouseMotionListener(mml);
    }
    void report(String field, String msg) {
        ((JTextField)h.get(field)).setText(msg);
    }
    FocusListener fl = new FocusListener() {
        public void focusGained(FocusEvent e) {
            report("focusGained", e paramString());
        }
        public void focusLost(FocusEvent e) {
            report("focusLost", e paramString());
        }
    };
    KeyListener kl = new KeyListener() {
        public void keyPressed(KeyEvent e) {
            report("keyPressed", e paramString());
        }
        public void keyReleased(KeyEvent e) {
            report("keyReleased", e paramString());
        }
        public void keyTyped(KeyEvent e) {
            report("keyTyped", e paramString());
        }
    };
    MouseListener ml = new MouseListener() {
        public void mouseClicked(MouseEvent e) {
            report("mouseClicked", e paramString());
        }
        public void mouseEntered(MouseEvent e) {
            report("mouseEntered", e paramString());
        }
        public void mouseExited(MouseEvent e) {
            report("mouseExited", e paramString());
        }
        public void mousePressed(MouseEvent e) {
            report("mousePressed", e paramString());
        }
        public void mouseReleased(MouseEvent e) {
            report("mouseReleased", e paramString());
        }
    };
    MouseMotionListener mml =
        new MouseMotionListener() {
        public void mouseDragged(MouseEvent e) {
            report("mouseDragged", e paramString());
        }
        public void mouseMoved(MouseEvent e) {
            report("mouseMoved", e paramString());
        }
    };
}

public class TrackEvent extends JApplet {
    HashMap h = new HashMap();
}

```

```

String[] event = {
    "focusGained", "focusLost", "keyPressed",
    "keyReleased", "keyTyped", "mouseClicked",
    "mouseEntered", "mouseExited", "mousePressed",
    "mouseReleased", "mouseDragged", "mouseMoved"
};

MyButton
b1 = new MyButton(this, Color.blue, "test1"),
b2 = new MyButton(this, Color.red, "test2");
public void init() {
    Container c = getContentPane();
    c.setLayout(new GridLayout(event.length+1,2));
    for(int i = 0; i < event.length; i++) {
        JTextField t = new JTextField();
        t.setEditable(false);
        c.add(new JLabel(event(i), JLabel.RIGHT));
        c.add(t);
        h.put(event(i), t);
    }
    c.add(b1);
    c.add(b2);
}
public static void main(String[] args) {
    JApplet applet = new TrackEvent();
    JFrame frame = new JFrame("TrackEvent");
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    frame.getContentPane().add(applet);
    frame.setSize(700, 500);
    applet.init();
    applet.start();
    frame.setVisible(true);
}
} //:~

```

Може да се види техниката използвана в конструктора да е едно и също името на аргумента и на това към което се приравнява и различаването им чрез **this**:

```
| this.label = label;
```

Методът **paint()** започва просто: запълва "окръглен правоъгълник" с цвета на бутона, после изчертава черна линия около него. Забележете използването на **size()** за да се определи дължината и ширината на компонента (в пиксели, разбира се). Нататък **paint()** изглежда доста сложен поради много изчисления провеждани за да се определи как да се центрира текстът върху бутона като се имат предвид "метриките на шрифта." Може да получите доста добра представа какво става, като гледате викането на метода и излиза, че това е доста типичен код така че може просто да го копирате навсякъде, където трябва да се центрира текст върху компонент.

Не може да разберете точно как **keyDown()**, **keyUp()** и т.н. методи работят докато не погледнете надолу към класа **TrackEvent**. Там се съдържа **HashMap** за държане на стринговете представлящи типа на събитието и **TextField** където се съдържа информацията за събитието. Разбира се, те биха могли да се създадат статично вместо да се пъхат в **HashMap**, но мисля ще се съгласите че така е много по-лесно за писане и промени. В

частност ако искате да прибавите или мащете нов тип събитие към **TrackEvent**, просто прибавяте или мащете стринг от масива **event** – всичко друго става автоматично.

Мястото където се разглеждат стринговете е в **keyDown()**, **keyUp()** и т.н. методите в **MyButton**. Всеки от тези методи използва **parent** манипулятор да достигне обратно в родителския прозорец. Тъй като този родител е **TrackEvent** той съдържа **HashMap h**, а **get()** методът, когато е снабден с подходящ **String**, ще дава манипулятор към **Object** което ние знаем че е **TextField** – така че се прави каст към таъв тип. После **Event** се превръща в неговото **String** представяне, което се изобразява в **TextField**.

Оказва се, че работата с този пример е весела понеже се вижда непосредствено какво става със събитията във вашата програма.

Ограничения при аплетите

Заради безопасността аплетите са твърде ограничени и има много неща които не можете да правите. Може да се отговори на въпроса какво изобщо могат да правят аплетите като се погледне какво се очаква да правят: да разширят функционалността на Web страница в браузър. Понеже като Web сърфист никога не може да знаете дали страницата е от приятелско място или не, иска се всеки код който ще работи да бъде безопасен. Така че най-големите ограничения които ще отбележите са вероятно:

1) Аплет не може да пипа локалния диск. Това значи писане и четене, понеже не бихте искали аплетът да чете и изпрати важна информация за вас по Web-а. Писането не се допуска, разбира се, защото е покана за вируси. Тези ограничения може да се отслабят когато цифровото подписване се реализира напълно.

Много ограничения са отслабени за тъй наречените доверени аплети (такива подписани от доверен източник) в по-новите браузери.

Има и други моменти когато се мисли за писането на аплети:

- ◆ Аплетите се разтоварват по-бавно, понеже винаги трябва да се разтовари цялото нещо, включително отделен достъп до сървъра за всеки отделен клас. Вашият браузър може да кешира аплета, но няма гаранции. Едно подобрение в Java 1.1 е JAR (Java ARchive) файла който позволява пакетирането на всичките компоненти на аплета (включително други .class файлове както и образи и звуци) заедно в един компресиран файл който може да бъде свален с един достъп до сървъра. “Цифровото подписване” (възможността да се осигури авторството на класа) е налична за всяко отделно нещо в JAR файла.
- ◆ Поради изискванията за безопасност трябва да се работи повече за някои неща като достъп до бази данни или изпращане на електронна поща. Освен това изискванията за сигурност правят достъпа до няколко източника труден, понеже всичко трябва да мина през Web сървър, който по този начин става тясно място и точка, която може да провали целия процес.
- ◆ Аплетът в браузър няма тази възможност за управление което нормалното приложение има. Например не може да имате модален диалогов прозорец в аплет, понеже потребителят винаги може да превключи на друго. Когато потребителят прескача от Web страница и даже прекратява работата на браузера, резултатите могат да бъдат катастрофални за вашия аплет – няма начин да се запази състоянието и ако сте на средата на транзакция може да се загуби информация. Освен това различните браузъри правят различни неща с вашия аплет когато излизате от Web страница така че резултатите са основно неопределени.

Предимства на аплетите

Ако ограниченията са приемливи, аплетите определено имат преимущества, особено при създаването на client/server или други мрежови приложения:

- ◆ Няма го въпросът с инсталацирането. Аплетите имат истинска платформена независимост (включително възможността за лесно възпроизвеждане на звукови файлове и т.н.) така че няма да правите никакви промени в кода си за различните платформи нито пък ще карате някого да се разправя с инсталации. Фактически инсталацията е автоматична всеки път когато се товари Web страницата заедно с аплетите, така че осъвременяването става тихо и автоматично. В традиционните клиент/сървър системи построяването и инсталацирането на нова версия е може да бъде кошмар.
- ◆ Поради вградените в самия език Java мерки за сигурност и структурата на аплетите няма нужда да се притеснявате за лош код който може да повреди нещия система. Това, заедно с предишната точка, прави Java (както и алтернативните средства за програмиране на клиентската страна при Web програмиране като JavaScript и VBScript) популярни за така наречените *Intranet* клиент/сървър приложения които живеят само в рамките на компания и не излизат навън по Internet.
- ◆ Понеже аплетите са автоматично интегрирани с HTML, имате платформено независима, вградена система за документация за аплета. Това е интересен номер, понеже сме свикнали да имаме документационната част от програма, а не обратното.

Приложения с прозорци

Възможно е да се види че поради изискванията за сигурност има ограничения при използването на аплети. В реален смисъл, аплетът е разширение на Web браузера така че неговата функционалност трябва да е ограничена според знанията и управлението на последния. Понякога, обаче, когато искате да направите програма за друго освен да седи на Web страница, може би като трябва да се направят неща като в "обикновено" приложение и все пак да е налице прехвалената преносимост на Java. В предишните глави на книгата сме правили приложения пусканни от команден ред, но в някои операционни среди (Macintosh, например) няма команден ред. Така че по много причини бихте могли да искате да направите прозоречна, но не-аплет програма използвайки Java. Това е обосновано желание.

Прозоречно приложение на Java може да има менюта и диалогови кутии нещо невъзможно или трудно в рамките на аплет (което е изгледа и чувството естествени за конкретната платформа. JFC/Swing библиотеката позволява да направите приложение което запазва изгледа и пр. На подлежащата операционна среда. Ако искате за правите прозоречни приложения, има смисъл да го правите само ако можете да използвате най-скорошната версия на Java и съответните инструменти така че да правите приложения които не биха обърквали потребителите. Ако по някаква причина се налага да използвате по-стара версия на Java, добре си помислете преди да решите да направите голямо прозоречно приложение.

Ето пример за прозоречно приложение:

```
//: c13:ButtonApp.java
// Creating an application
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class ButtonApp extends JFrame {
    JButton
```

```

b1 = new JButton("Hello"),
b2 = new JButton("Howdy");
JTextField t = new JTextField(15);
ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e){
        String name =
            ((JButton)e.getSource()).getText();
        t.setText(name);
    }
};
public ButtonApp(String name) {
    super(name);
    b1.addActionListener(al);
    b2.addActionListener(al);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(b1);
    cp.add(b2);
    cp.add(t);
}
public static void main(String[] args) {
    JFrame frame = new ButtonApp("ButtonApp");
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    frame.setSize(400,100);
    frame.setVisible(true);
}
} //:~

```

Комбинировано приближение/аплет

```

//: c13:ButtonAppApplet.java
// Creating an applet-application
// <applet code=ButtonAppApplet
// width=375 height=50> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class ButtonAppApplet extends JApplet {
    JButton
        b1 = new JButton("Hello"),
        b2 = new JButton("Howdy");
    JTextField t = new JTextField(15);
    ActionListener al = new ActionListener() {
        public void actionPerformed(ActionEvent e){
            String name =
                ((JButton)e.getSource()).getText();
            t.setText(name);
        }
    };
}

```

```

public void init() {
    b1.addActionListener(al);
    b2.addActionListener(al);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(b1);
    cp.add(b2);
    cp.add(t);
}
public static void main(String[] args) {
    JApplet applet = new ButtonAppApplet();
    JFrame frame = new JFrame("ButtonAppApplet");
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    frame.getContentPane().add(applet);
    frame.setSize(400,100);
    applet.init();
    applet.start();
    frame.setVisible(true);
}
} ///:~

```

Менюта

Има четири различни типа наследени от абстрактния клас **MenuComponent**: **MenuBar** (може да имате само един **MenuBar** на конкретен **Frame**), **Menu** за поддържане на едно падащо меню и подменюта, **MenuItem** за представяне на един елемент от меню и **CheckboxMenuItem**, който е извлечен от **MenuItem** и дава марка която показва дали даден елемент на менюто е избран.

За разлика от система която използва ресурси, с Java и Swing трябва да направите всички менюта в сорс код. Ето миризмите на сладоледа пак, използвани за създаване на меню:

```

//: c13:Menu1.java
// Shows submenus, checkbox menu items
// and swapping menus
// <applet code=Menu1
// width=300 height=100> </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event;

public class Menu1 extends JApplet {
    String[] flavors = { "Chocolate", "Strawberry",
        "Vanilla Fudge Swirl", "Mint Chip",
        "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie"
    };
    JTextField t = new JTextField("No flavor", 30);

```

```

JMenuBar mb1 = new JMenuBar();
JMenu
f = new JMenu("File"),
m = new JMenu("Flavors"),
s = new JMenu("Safety");
// Alternative approach:
JCheckBoxMenuItem() safety = {
    new JCheckBoxMenuItem("Guard"),
    new JCheckBoxMenuItem("Hide")
};
JMenuItem() file = {
    new JMenuItem("Open"),
    new JMenuItem("Exit")
};
// A second menu bar to swap to:
JMenuBar mb2 = new JMenuBar();
JMenu fooBar = new JMenu("fooBar");
JMenuItem() other = {
    new JMenuItem("Foo"),
    new JMenuItem("Bar"),
    new JMenuItem("Baz"),
};
JButton b = new JButton("Swap Menus");
public void init() {
    for(int i = 0; i < flavors.length; i++) {
        JMenuItem mi = new JMenuItem(flavors(i));
        mi.addActionListener(al);
        m.add(mi);
        // Add separators at intervals:
        if((i+1) % 3 == 0)
            m.addSeparator();
    }
    for(int i = 0; i < safety.length; i++) {
        safety(i).addActionListener(al);
        s.add(safety(i));
    }
    f.add(s);
    for(int i = 0; i < file.length; i++) {
        file(i).addActionListener(al);
        f.add(file(i));
    }
    mb1.add(f);
    mb1.add(m);
    t.setEditable(false);
    Container cp = getContentPane();
    cp.add(t, BorderLayout.CENTER);
    // Set up the system for swapping menus:
    b.addActionListener(al);
    cp.add(b, BorderLayout.NORTH);
    for(int i = 0; i < other.length; i++) {
        other(i).addActionListener(al);
        fooBar.add(other(i));
    }
    mb2.add(fooBar);
    setJMenuBar(mb1);
}
ActionListener al = new ActionListener() {

```

```

public void actionPerformed(ActionEvent e) {
    String arg = e.getActionCommand();
    Object source = e.getSource();
    if(source.equals(b)) {
        JMenuBar m = getJMenuBar();
        setJMenuBar(m == mb1 ? mb2 : mb1);
        validate(); // Refresh the frame
    } else if(source instanceof JMenuItem) {
        if(arg.equals("Open")) {
            String s = t.getText();
            boolean chosen = false;
            for(int i = 0; i < flavors.length; i++)
                if(s.equals(flavors(i)))
                    chosen = true;
            if(!chosen)
                t.setText("Choose a flavor first!");
            else
                t.setText("Opening "+s+". Mmm, mm!");
        } else if(source.equals(file(1)))
            System.exit(0);
        // CheckboxMenuItem cannot use String
        // matching; you must match getSource():
        else if(source.equals(safety(0)))
            t.setText("Guard the Ice Cream! " +
                      "Guarding is "+safety(0).getState());
        else if(source.equals(safety(1)))
            t.setText("Hide the Ice Cream! " +
                      "Is it cold? "+safety(1).getState());
        else
            t.setText(arg.toString());
    }
}
};

public static void main(String[] args) {
    JApplet applet = new Menu1();
    JFrame frame = new JFrame("Menu1");
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    frame.getContentPane().add(applet);
    frame.setSize(300, 100);
    applet.init();
    applet.start();
    frame.setVisible(true);
}
} ///:~

```

В тази програма избегнах типичните дълги списъци от викания на **add()** за всяко меню понеже те ми изглеждаха като много ненужно писане. Вместо това сложих елементите на менютата в масив и после просто се придвижих по всеки масив викайки **add()** във **for** цикъл. Това прави добавянето и мащабирането на елементи на менютата по-малко досадно.

Като алтернативен подход (който намерих за по-малко желан понеже изисква повече писане), **CheckboxMenuItem**ите са създадени в масив наречен **safety**; това е така за масивите **file** и **other** също.

Тази програма създава не един, а два **MenuBar** за да демонстрира че те могат да бъдат активно превключвани по време на изпълнение. Може да видите как **MenuBar** е направен от **MenuItem** и всяко **Menu** е направено от **MenuItem**, **CheckboxMenuItem**, даже от други **MenuItem** (което дава подменюта). Когато се съставя **MenuBar** той може да бъде поставен в текущата програма чрез **setMenuBar()** метода. Забележете че когато е натиснат клавиш, той гледа кое меню е текущо сложено чрез **getMenuBar()**, после слага другата лента с меню на мястото.

Когато се проверява за "Open," забележете че правилното писане и капитализацията са критични, но Java не сигнализира за грешка ако няма съвпадение за "Open." Този начин на сравняване на стрингове е чист източник на програмни грешки.

Отбелязването на елементите на менютата става автоматично, но оправяне с **CheckboxMenuItem**s може да бъзе малко с изненади, понеже по някаква причина не са позволили съвпадането на стрингове. (Въпреки че съвпадането на стрингове не е добър подход, това изглежда безсмислено.) Така че може да съвпаднате само обекта-цел и не неговия етикет. Както е показано, **getState()** може да бъде използван за да се види състоянието. Може също да сменяте състоянието на **CheckboxMenuItem** със **setState()**.

Може да помислите че едно меню може да стои на няколко ленти с менюта. Това изглежда смислено понеже всичко което се подава на **MenuBar add()** метода е манипулятор. Обаче ако опитате това, поведението ще е странно и не такова, каквото го очаквате. (Трудно е да се каже дали това е бъг или е направено нарочно така.)

Този пример също показва какво трябва да се направи за да се създаде приложение заместо аплет. (Отново, понеже приложението може да има меню а аплетът на може да поддържа менюта.) Наместо да се наследи от **Applet**, наследявате от **Frame**. Вместо **init()** да стартира нещата, правите конструктор за вашия клас. Накрая, създавате **main()** в съято създавате обект от новия тип, преоразмерявате го, а после викате **show()**. Това е различно от аплетите само малко на няколко места, но вече имате самостоятелно приложение с прозорци и менюта.

Диалогови кутии

Диалоговата кутия е прозорец, който изскуча от друг прозорец. Предназначението му е да се реши конкретен въпрос без да се занимава оригиналния прозорец с него. Диалоговите кутии много се използват в програмните среди с прозорци, но както се спомена по-рано, рядко се използват с аплети.

За да създавате диалогова кутия наследявате от **Dialog**, който е просто още един вид **Window**, както **Frame**. За разлика от **Frame**, **Dialog** не може да има лента с менюта или да промени курсора, но во останалото те са доста подобни. Диалогът има управител на разполагането (по подразбиране **BorderLayout**) и подтискате **action()** и т.н., или **handleEvent()** за работа със събитията. Една значителна разлика виждаме в **handleEvent()**: когато се случи събитието **WINDOW_DESTROY**, не искате да затваряте приложението! Вместо това освобождавате ресурсите на диалога чрез **dispose()** след като го приключите.

В следния пример диалоговата кутия е направена с мрежа (чрез **GridLayout**) от специален вид бутон който е дефиниран тук като **ToeButton**. Този бутон си рисува линия около себе си и, в зависимост от състоянието си, празно, "x" или "o" в средата. Започва с празно, после в зависимост чий ред е, се променя на "x" или "o." Обаче също ще се превключва напред-назад между "x" и "o" когато кликнете бутона. (Това прави концепцията на играта само малко по-досадна отколкото тя си е.) Освен това диалоговия прозорец може да бъде направен с

произволен брой редове и колони чрез задаване на числа в главния прозорец на приложението.

```
//: c13:TicTacToe.java
// Demonstration of dialog boxes
// and creating your own components
import javax.swing.*;
import java.awt.*;
import java.awt.event;

class ToeButton extends JPanel {
    int state = ToeDialog.BLANK;
    ToeDialog p;
    MouseListener ml = new MouseAdapter() {
        public void mousePressed(MouseEvent e) {
            if(state == ToeDialog.BLANK) {
                state = p.turn;
                p.turn = (p.turn == ToeDialog.XX ?
                    ToeDialog.OO : ToeDialog.XX);
            } else
                state = (state == ToeDialog.XX ?
                    ToeDialog.OO : ToeDialog.XX);
            repaint();
        }
    };
    ToeButton(ToeDialog parent) {
        p = parent;
        addMouseListener(ml);
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int x1 = 0;
        int y1 = 0;
        int x2 = getWidth() - 1;
        int y2 = getHeight() - 1;
        g.drawRect(x1, y1, x2, y2);
        x1 = x2/4;
        y1 = y2/4;
        int wide = x2/2;
        int high = y2/2;
        if(state == ToeDialog.XX) {
            g.drawLine(x1, y1, x1 + wide, y1 + high);
            g.drawLine(x1, y1 + high, x1 + wide, y1);
        }
        if(state == ToeDialog.OO) {
            g.drawOval(x1, y1, x1+wide/2, y1+high/2);
        }
    }
}

class ToeDialog extends JDialog {
    static final int BLANK = 0, XX = 1, OO = 2;
    int turn = XX; // Start with x's turn
    // w = number of cells wide
    // h = number of cells high
    public ToeDialog(JFrame p, int w, int h) {
        super(p, "The game itself", false);
```

```

addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        dispose();
    }
});
Container cp = getContentPane();
cp.setLayout(new GridLayout(w, h));
for(int i = 0; i < w * h; i++)
    cp.add(new ToeButton(this));
setSize(w * 50, h * 50);
}
}

public class TicTacToe extends JFrame {
JTextField
rows = new JTextField("3"),
cols = new JTextField("3");
 JButton go = new JButton("go");
public TicTacToe() {
setTitle("Toe Test");
JPanel p = new JPanel();
p.setLayout(new GridLayout(2,2));
p.add(new JLabel("Rows", JLabel.CENTER));
p.add(rows);
p.add(new JLabel("Columns", JLabel.CENTER));
p.add(cols);
Container cp = getContentPane();
cp.add(p, BorderLayout.NORTH);
cp.add(go, BorderLayout.SOUTH);
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e){
        System.exit(0);
    }
});
go.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e){
        JDialog d = new ToeDialog(TicTacToe.this,
            Integer.parseInt(rows.getText()),
            Integer.parseInt(cols.getText()));
        d.setVisible(true);
    }
});
}
public static void main(String[] args) {
    JFrame frame = new TicTacToe();
    frame.setSize(200, 100);
    frame.setVisible(true);
}
} //:~

```

Класът **ToeButton** пази манипулятор към родителя си, който трябва да бъде от тип **ToeDialog**. Както по-рано, това създава тясно свързване, понеже **ToeButton** може да се използва само с **ToeDialog**, но решава множество проблеми и наистина не изглежда толкова лошо решение понеже няма друг диалог който да следи чий е редът. Разбира се бихте могли да приемете друг подход, който е да се направи **ToeDialog.turn** да бъде **static** член на **ToeButton**. Това премахва свързването, но и не позволява да имате повече от един **ToeDialog** едновременно. (Или поне повече от един който работи правилно, във всеки случай.)

Методът **paint()** е за графиката: чертане на квадрат около бутона и "х" или "о." Пълен е с досадни изчисления, но е съвсем праволинеен.

Кликването с мишката се улавя от подтиснатия **mouseDown()** метод, който първо гледа дали нещо е написано върху бутона. Ако не е, запитва се бащиния прозорец чий ред е и се изобразява съответно на бутона. Забележете че бутона по-случајно отива в бащиния прозорец и сменя реда. Ако бутона вече показва "х" или "о" се превключва да показва другото. Може да видите в тези изчисления троичното if-else описано в глава 3. След като състоянието на бутона се промени той се прерисува.

Конструкторът на **ToeDialog** е доста прост: той добавя в **GridLayout** толкова бутона, колкото искате, после го преоразмерява на 50 пиксела на страна за всеки бутон. (Ако не преоразмерите **Window**, той няма да се покаже!) Забележете че **handleEvent()** просто вика **dispose()** за **WINDOW_DESTROY** така че не заминава цялото приложение.

TicTacToe установява цялото приложение чрез запълването на **TextField**ове (за въвеждането на стълбовете и колоните на мрежата) и "go" бутона. Ще видите в **action()** че тази програма използва по-малко желаната "string match" техника за детекция на натискането на бутон (осигурете че сте ги написали верно, с капитализацията включително!). Когато се натисне бутон данните в **TextFields** трябва да се извлекат и, понеже са във форма на **String** да се превърнат в **int** чрез **static Integer.parseInt()** метода. Щом е създаден **Dialog**ът, методът **show()** трябва да се извика за да го изобрази и активира.

Ще забележите че обекта **ToeDialog** е присвоен на **Dialog** манипулатора **d**. Това е пример на ъпкастинг, макар че тук да няма особена разлика понеже всичко което става е да се извика метода **show()**. Обаче ако искахте да викате метод който съществува само в **ToeDialog** щяхте да пожелаете да присвоите на **ToeDialog** манипулатор и да не губите информация при ъпкаста.

Файлови диалози

Някои операционни системи имат множество вградени диалози за избор на такива неща като фонтове, цветове, принтери и други подобни. Практически всички графични операционни системи поддържат отварянето и пр. Работа с файлове, обаче, та **JFileChooser** на Java ги капсулира за лесна употреба.

Следното приложение изпитва две форми на **JFileChooser** диалози, едната за отваряне и другата за запазване. Повечето код би трябвало да е ясен, а интересните неща стават в слушателите на натискането на бутоните:

```
//: c13:FileChooserTest.java
// Demonstration of File dialog boxes
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class FileChooserTest extends JFrame {
    JTextField
    filename = new JTextField(),
    dir = new JTextField();
    JButton
    open = new JButton("Open"),
    save = new JButton("Save");
    public FileChooserTest() {
        setTitle("File Dialog Test");
        JPanel p = new JPanel();
        open.addActionListener(new OpenL());
        p.add(open);
```

```

save.addActionListener(new SaveL());
p.add(save);
Container cp = getContentPane();
cp.add(p, BorderLayout.SOUTH);
dir.setEditable(false);
filename.setEditable(false);
p = new JPanel();
p.setLayout(new GridLayout(2,1));
p.add(filename);
p.add(dir);
cp.add(p, BorderLayout.NORTH);
}
class OpenL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JFileChooser c = new JFileChooser();
        // Demonstrate "Open" dialog:
        int rVal =
            c.showOpenDialog(FileChooserTest.this);
        if(rVal == JFileChooser.APPROVE_OPTION) {
            filename.setText(
                c.getSelectedFile().getName());
            dir.setText(
                c.getCurrentDirectory().toString());
        }
        if(rVal == JFileChooser.CANCEL_OPTION) {
            filename.setText("You pressed cancel");
            dir.setText("");
        }
    }
}
class SaveL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JFileChooser c = new JFileChooser();
        // Demonstrate "Save" dialog:
        int rVal =
            c.showSaveDialog(FileChooserTest.this);
        if(rVal == JFileChooser.APPROVE_OPTION) {
            filename.setText(
                c.getSelectedFile().getName());
            dir.setText(
                c.getCurrentDirectory().toString());
        }
        if(rVal == JFileChooser.CANCEL_OPTION) {
            filename.setText("You pressed cancel");
            dir.setText("");
        }
    }
}
public static void main(String[] args) {
    JFrame frame = new FileChooserTest();
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    frame.setSize(250,110);
}

```

```

    frame.setVisible(true);
}
} //:~
```

Забележете, че много вариации са възможни с **JFileChooser**, включително филтри за стесняване обхвата на имената които може да се използват.

За "open file" диалог използвате конструктор който има два аргумента; първият е манипулатора на бащиния прозорец а вторият е текста за заглавната лента на диалога **FileDialog**. Методът **setFile()** дава начално файлово име – предполагамо ОС поддържа метасимволи, така че в този пример всичките **.java** файлове първоначално ще се изобразяват. Методът **setDirectory()** избира директорията в която селекцията на файловете ще започне. (Изобщо, ОС позволява на потребителя да сменя директориите.)

Командата **show()** не завършва докато диалогът не приключи работата си. Обектът **FileDialog** все още съществува, така че може да четете данни от него. Ако извикате **getFile()** и той връне **null** това значи че потребителят е канцелирал диалога. И името на файла и резултата от **getDirectory()** са изобразени в **TextField**ове.

Бутона за запазване работи по същия начин, освен че използва различен конструктор на **FileDialog**. Този конструктор взима три аргумента и третият трябва да бъде или **FileDialog.SAVE** или **FileDialog.OPEN**.

Моделът на събитията

В новия модел на събитията компонент може да предизвика ("fire") събитие. Всеки тип събитие е представен от различен клас. Когато събитието стане, то се приема от един или повече "слушатели," които се задействат при това. Така мястото където събитието се предизвиква и това където се обработва може да не е едно и също.

Всеки слушател на събитие е обект от конкретен клас който прилага слушателски **interface**. Така като програмист всичко което трябва да направите е да създадете слушателя и да го регистрирате с обекта, който задейства събитието. Тази регистрация се изпълнява чрез извикване на **addXXXListener()** метода на задействащия събитието обект, където **XXX** типа за който е слушателят. Лесно може да видите кои типове събития може да се поддържат като видите имената на **addListener** методите и ако се опитате да слушате за неправилно събитие ще получите грешка по време на компилация. Java Beans също използва имената на **addListener** за определяне какво Bean може да прави.

Цялата логика за събитието, значи, ще бъде в слушателския клас. Когато създавате клас-слушател, единственото ограничение е че трябва да прилага съответния интерфейс. Може да създадете глобален клас-слушател, но това е ситуация в която вътрешните класове имат тенденция да бъдат твърде полезни, не само защото дават логическо групиране на слушателите в UI или класовете на работната логика които обслужват, но и (както ще видите по-късно) фактът че вътрешният клас съдържа манипулятор към родителя си дава прекрасен начин за вikanе през границите на класовете и подсистемата.

Прост пример ще изясни това. Да вземем примера **Button2.java** от по-рано в тази глава.

```

//: c13:Button2New.java
// Capturing button presses
// <applet code=Button2New
// width=200 height=50> </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.; // Must add this
```

```

public class Button2New extends JApplet {
    JButton
    b1 = new JButton("Button 1"),
    b2 = new JButton("Button 2");
    public void init() {
        b1.addActionListener(new B1());
        b2.addActionListener(new B2());
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(b1);
        cp.add(b2);
    }
    class B1 implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            getAppletContext().showStatus("Button 1");
        }
    }
    class B2 implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            getAppletContext().showStatus("Button 2");
        }
    }
} //:~

```

Така може да сравнете двата подхода, старият е оставен под коментар. В **init()** единствената промяна е добавката на два реда:

```

b1.addActionListener(new B1());
b2.addActionListener(new B2());

```

addActionListener() казва на бутона кой обект да активира когато е натиснат бутон. Класовете **B1** и **B2** са вътрешни класове които прилагат **interface ActionListener**. Този интерфейс съдържа единствен метод **actionPerformed()** (значещ "Това е действието което ще се изпълни когато стане събитието"). Забележете че **actionPerformed()** не взема родово събитие, а конкретен тип такова, **ActionEvent**. Така че не се занимавате с проверки и даункастинг за да намерите специфична **ActionEvent** информация.

Едно от най-хубавите неща на **actionPerformed()** е простотата му. Това е просто метод който се вика. Сравнете го със стария **action()** метод, в който трябваше да установите какво е станало и да реагирате съответно, а също да се притеснявате за версията от базовия клас на **action()** и да връщате стойност индицираща дали е била направена обработката. С новия модел на събитията всичко се прави от логиката и няма нужда да се грижите вие; просто казвате какво става и сте готови. Ако още не предпочитате този подход пред стария, скоро ще започнете.

ТИПОВЕ СЪБИТИЯ И СЛУШАТЕЛИ

Всичките компоненти на Swing са променени да включват **addXXXListener()** и **removeXXXListener()** така че подходящи типове слушатели да могат да бъдат добавяни към и махани от всеки компонент. Ще забележите че "XXX" във всеки клас също представя и аргумент на метода, например **addFooListener(FooListener f)**. Следващата таблица дава асоциираните събития, слушатели, методи и компоненти които поддържат конкретния вид събитие чрез **addXXXListener()** и **removeXXXListener()** методи.

Събитие, слушателски интерфейс и методи за добавяне и мащане	Компоненти поддържащи това събитие
ActionEvent ActionListener addActionListener() removeActionListener()	Button , List , TextField , MenuItem , и деривати вкл. CheckboxMenuItem , Menu и PopupMenu

Събитие, слушателски интерфейс и методи за добавяне и мащане	Компоненти поддържащи това събитие
AdjustmentEvent AdjustmentListener addAdjustmentListener() removeAdjustmentListener()	Scrollbar Всичко прилагащо Adjustable интерфейс
ComponentEvent ComponentListener addComponentListener() removeComponentListener()	Component и производните му, вкл. Button , Canvas , Checkbox , Choice , Container , Panel , Applet , ScrollPane , Window , Dialog , FileDialog , Frame , Label , List , Scrollbar , TextArea и TextField
ContainerEvent ContainerListener addContainerListener() removeContainerListener()	Container и деривати, вкл. Panel , Applet , ScrollPane , Window , Dialog , FileDialog и Frame
FocusEvent FocusListener addFocusListener() removeFocusListener()	Component и деривати, вкл. Button , Canvas , Checkbox , Choice , Container , Panel , Applet , ScrollPane , Window , Dialog , FileDialog , Frame , Label , List , Scrollbar , TextArea и TextField
KeyEvent KeyListener addKeyListener() removeKeyListener()	Component и деривати, вкл. Button , Canvas , Checkbox , Choice , Container , Panel , Applet , ScrollPane , Window , Dialog , FileDialog , Frame , Label , List , Scrollbar , TextArea и TextField
MouseEvent (за кликове и движение) MouseListener addMouseListener() removeMouseListener()	Component и деривати, вкл. Button , Canvas , Checkbox , Choice , Container , Panel , Applet , ScrollPane , Window , Dialog , FileDialog , Frame , Label , List , Scrollbar , TextArea и TextField
MouseEvent ⁶ (за кликове и движение) MouseMotionListener addMouseMotionListener() removeMouseMotionListener()	Component и деривати, вкл. Button , Canvas , Checkbox , Choice , Container , Panel , Applet , ScrollPane , Window , Dialog , FileDialog , Frame , Label , List , Scrollbar , TextArea и TextField
WindowEvent WindowListener addWindowListener() removeWindowListener()	Window и деривати, вкл. Dialog , FileDialog и Frame
ItemEvent ItemListener addItemListener() removeItemListener()	Checkbox , CheckboxMenuItem , Choice , List и всичко, което прилага ItemSelectable интерфейс
TextEvent TextListener	Всичко извлечено от TextComponent , вкл. TextArea и

⁶ There is no **MouseMotionEvent**, even though it seems like there ought to be. Clicking and motion is combined into **MouseEvent**, so this second appearance of **MouseEvent** in the table is not an error.

Събитие, слушателски интерфейс и методи за добавяне и мащане	Компоненти поддържащи това събитие
<code>addTextListener()</code> <code>removeTextListener()</code>	<code>TextField</code>

Може да се види, че всеки тип компонент поддържа само някои видове събития. Полезно е да се видят събитията поддържани от всеки компонент, както в следната таблица:

Тип на компонент	Събития поддържани от този компонент
Adjustable	AdjustmentEvent
Applet	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Button	ActionEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Canvas	FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Checkbox	ItemEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
CheckboxMenuItem	ActionEvent, ItemEvent
Choice	ItemEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Component	FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Container	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Dialog	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
FileDialog	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Frame	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Label	FocusEvent, KeyEvent, MouseEvent, ComponentEvent
List	ActionEvent, FocusEvent, KeyEvent, MouseEvent, ItemEvent, ComponentEvent
Menu	ActionEvent
MenuItem	ActionEvent
Panel	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
PopupMenu	ActionEvent
Scrollbar	AdjustmentEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
ScrollPane	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
TextArea	TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent

Тип на компонент	Събития поддържани от този компонент
TextComponent	TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
TextField	ActionEvent, TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Window	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent

Щом знаете кое събитие поддържа конкретен компонент, няма нужда да гледате нищо допълнително за да стигнете до събитието. Просто:

1. Вземате името на събитийния клас и махате **“Event.”** Добавяте думата **“Listener”** към това, което остава. Това е слушателския интерфейс който трябва да приложите във вашия клас.
2. Реализирайте наследяването по-горе и добавете методи за събитията които искате да хванете. Например може да внимавате за придвижвания на мишката, така че напишете код за **mouseMoved()** метода от **MouseMotionListener** интерфейса. (Трябва да реализирате и другите методи, разбира се, но има кратък път, както скоро ще видите.)
3. Създайте обект от слушателския клас от стъпка 2. Регистрирайте го във вашия компонент с метод започващ с представка **“add”** към името на слушателя. Например, **addMouseMotionListener()**.

За да завършив с каквото трябва да се знае, ето слушателските интерфейси:

Слушателски интерфейс с/адаптер	Методи в интерфейса
ActionListener	actionPerformed(ActionEvent)
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
ComponentListener ComponentAdapter	componentHidden(ComponentEvent) componentShown(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent)
ContainerListener ContainerAdapter	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
FocusListener FocusAdapter	focusGained(FocusEvent) focusLost(FocusEvent)
KeyListener KeyAdapter	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
MouseListener MouseAdapter	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)
MouseMotionListener MouseMotionAdapter	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
WindowListener WindowAdapter	windowOpened(WindowEvent) windowClosing(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent)

Слушателски интерфейс с/адаптер	Методи в интерфейса
	windowDeactivated(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent)
ItemListener	itemStateChanged(ItemEvent)
TextListener	textValueChanged(TextEvent)

Използване на слушателски адаптери за простота

В горната таблица може да видите, че някои слушателски интерфейси имат само един метод. Те са лесни за реализация, понеже ги реализирате само когато ви трябва точно този метод. Обаче слушателските интерфейси които имат много методи може да не са толкова приятни за използване. Например нещо което винаги трябва да правите приложение е да доставите **WindowListener** на **Frame** така че когато получите **windowClosing()** събитие да може да извикате **System.exit(0)** за завършване на програмата. Но понеже **WindowListener** е **interface**, трябва да напишете всичките методи даже и да не правите нищо. Това може да дразни.

За да се реши проблема, всички такива интерфейси са снабдени с **адаптери**, имената на които може да видите в таблицата по-горе. Всеки адаптер дава методи по подразбиране за всеки от интерфейсните методи. (Уви, **WindowAdapter** няма **windowClosing()** по подразбиране който вика **System.exit(0)**.) При това положение трябва само да се наследи от адаптера и да се подтиснат методите които ви трябват. Например типичният **WindowListener** който ще използвате изглежда примерно така:

```
class MyWindowListener extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
}
```

Цялата работа с адаптерите е да се направи създаването на слушателски класове лесно.

Адаптерите имат и обратна страна, обаче, във формата на капан. Да предположим че пишете **WindowAdapter** като горния:

```
class MyWindowListener extends WindowAdapter {
    public void WindowClosing(WindowEvent e) {
        System.exit(0);
    }
}
```

Това не работи, но може да ви подлуди докато търсите защо, понеже всичко се компилира както трябва – освен че затварянето на прозореца не приключва и програмата. Можете ли да видите проблема? Той е в името на метода: **WindowClosing()** вместо **windowClosing()**. Простишък пропуск в капитализацията води до появата на нов метод. Обаче това не е метода, който се вика когато прозорецът се затваря, така че не получавате желаните резултати.

Правене на прозорци и аплети

Често ще искате да напишете клас така, че да може да бъде викан както прозорец, така и като аплет. За да се постигне това просто добавяте **main()** към вашия аплет който построява интерфейс на аплета вътре във **Frame**. Като приста пример нека погледнем **Button2New.java** променен да работи както като прозорец,, така и като аплет:

```
//: c13:Button2NewB.java
```

```

// An application and an applet
// <applet code=Button2NewB width=250
// height=75></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event;

public class Button2NewB extends JApplet {
    JButton
    b1 = new JButton("Button 1"),
    b2 = new JButton("Button 2");
    JTextField t = new JTextField(20);
    public void init() {
        b1.addActionListener(new B1());
        b2.addActionListener(new B2());
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(b1);
        cp.add(b2);
        cp.add(t);
    }
    class B1 implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            t.setText("Button 1");
        }
    }
    class B2 implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            t.setText("Button 2");
        }
    }
    // To close the application:
    static class WL extends WindowAdapter {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    }
    // A main() for the application:
    public static void main(String[] args) {
        JApplet applet = new Button2NewB();
        JFrame frame = new JFrame("Button2NewB");
        frame.addWindowListener(new WL());
        frame.getContentPane().add(applet);
        frame.setSize(300,100);
        applet.init();
        applet.start();
        frame.setVisible(true);
    }
} ///:~

```

Вътрешният клас **WL** и **main()** са единствените два елемента добавени към аплета, а останалата част от него е непроменена. Фактически можете често да копирате и вмъквате класа **WL** и **main()** във вашите аплети с малка промяна. Класът **WL** е **static** така че може лесно да бъде създаден в **main()**. (Помните че вътрешният клъс нормално иска манипулятор на външен клас при създаването си. Като се направи **static** тази необходимост отпада.) Може да видите че в **main()** аплетът е явно инициализиран и стартиран понеже в този случай браузерът го няма да свърши тази работа. Разбира се, това не дава пълната функционалност на

браузъра, който също вика **stop()** и **destroy()**, но в помечето случаи е приемливо. Ако има проблеми, можете:

1. Да направите манипулатора **applet** да бъде **static** член на класа (вместо локална променлива в **main()**), а после:
2. Да извикате **applet.stop()** и **applet.destroy()** в **WindowAdapter.windowClosing()** преди да извикате **System.exit()**.

Забележете последния ред:

```
| frame.setVisible(true);
```

Това е една от промените в AWT на Java 1.1. Методът **show()** е остатъл и **setVisible(true)** го замества. Този сорт изглеждащи капризни замени ще добият смисъла си за вас като се запознаете с Java Beans по-късно в тази глава.

Този пример е също промене да използва **TextField** вместо да извежда на конзолата или статус-линията на браузъра. Едно ограничение при правенето на програми, които могат да бъдат и аплети, и самостоятелни е, че трябва да изберете такава форма на вход-идхода, че да работи и в двата случая.

Има и друга малка черта в AWT Java 1.1 показана тук. Вече не е необходимо да използвате потенциално пълния с възможности за грешки подход със специфициране на **BorderLayout** местата използвайки **String**. Когато добавяте елемент към **BorderLayout** в Java 1.1 може да напишете:

```
| frame.add(applet, BorderLayout.CENTER);
```

Наричате мястото с една от **BorderLayout** константите, която после ще се провери по време на компилация (наместо тихичко да се правят бели, както с предишния начин). Това определено е подобрение и ще се използва в тази книга.

Правене на анонимен клас слушател на прозорец

Всеки от слушателските класове би могъл да се реализира като анонимен, но винаги има шанс да поискате да използвате тяхната функционалност и другаде. Обаче тук слушателят се използва само за да затвори прозореца, така че може безопасно да се използва анонимен клас. Тогава редът в **main()**:

```
| frame.addWindowListener(new WL());
```

ще стане:

```
frame.addWindowListener(  
    new WindowAdapter() {  
        public void windowClosing(WindowEvent e) {  
            System.exit(0);  
        }  
    });
```

Това има предимството че не изисква още едно име на клас. Трябва сами за себе си да решите дали това прави кода по-лесен или по-труден за разбиране. Обаче в останалата част на книгата често ще бъде използван анонимен клас за слушател на прозорец.

Пакетиране на аплет в JAR файл

Важно използване на JAR е да се оптимизира товаренето на аплетите. В Java 1.0 имаше тенденция потребителите да слагат всички си код в един **Applet** клас така че да трябва един достъп до сървъра за товаренето му. Не само че това резултираше в объркан, труден за четене (и поддръжка) код, но **.class** файлът си беше некомпресиран и товаренето му не бе така бързо както можеше въобще да бъде.

JAR файловете промениха всичко това така, че да може да се компресират всички **.class** файлове в един, който да бъде свален от браузъра. Сега няма нужда да правите дизайн който да минимизира броя на файловете и потребителят ще получи много по-бързо сваляне на необходимото.

Да видим горния пример. Той изглежда като **Button2NewB** в единствен клас, но фактически съдържа три вътрешни класа, така че общо са четири. Веднъж компилирали програмата, пакетирайте я в JAR файл с реда:

```
| jar cf Button2NewB.jar *.class
```

Това предполага че само **.class** файловете в текущата директория са тези на **Button2NewB.java** (иначе ще имате излишен багаж).

Сега създавате HTML страница с нов **archive** таг за индикация на името на JAR файла, подобно на тази:

```
<head><title>Button2NewB Example Applet
</title></head>
<body>
<applet code=Button2NewB.class
        archive=Button2NewB.jar
        width=200 height=150>
</applet>
</body>
```

Всичко останало за таговете за аплети в HTML файлове остава същото.

Преразглеждане на по-раншните примери

За да можете да видите множество примери с новия модел на събитията, а също така да видите пътя по който става конверсията на код от стария модел към новия, много от нещата разгледани в предишни примери се разглеждат тук повторно с използване на новия модел. В добавка, всяка програма сега е аплет и приложение, така че можете да се пуска и с, и без браузър.

Демонстриране методите на рамката

Интересно е да се видят някои от тези методи в действие. (Този пример ще разгледа само **init()**, **start()** и **stop()** понеже **paint()** и **destroy()** са очевидни и не така лесно трасираме.) Следният аплет следи колко пъти са използвани методите и извежда това чрез **paint()**:

```
//: c13:Applet3.java
// Shows init(), start() and stop() activities
// <applet code=Applet3 width=150 height=50>
// </applet>
```

```

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

public class Applet3 extends JApplet {
    String s;
    int inits = 0;
    int starts = 0;
    int stops = 0;
    public void init() { inits++; }
    public void start() { starts++; }
    public void stop() { stops++; }
    public void paint(Graphics g) {
        s = "inits: " + inits +
            ", starts: " + starts +
            ", stops: " + stops;
        g.drawString(s, 10, 10);
    }
} //:~

```

Когато подписвате метод обикновено искате да знаете дали е необходимо да викате версията от базовия клас, в случай че тя прави нещо важно. Например с **init()** може да трябва да викате **super.init()**. Обаче документацията на **Applet** специално твърди че **init()**, **start()** и **stop()** методите в **Applet** нищо не правят, така че не е необходимо да се викат тук.

Когато експериментирате с този аплет може да откриете, че когато минимизирате прозореца на Web браузъра или го покриете с друг прозорец може да не се извикат **stop()** и **start()** (когато прозорецът отново стане видим - б.пр.). (Това поведение изглежда варира с приложенията; може да поискате да си изострите усещането за Web браузърите с това за средствата за показване на аплети.) Единственият път когато ще се извикат споменатите методи е когато отидете на друга страница и после се върнете на тази, която съдържа аплета.

Текстови полета

Това е подобно на **TextField1.java**, но добавя значително повече възможности:

```

//: c13:TextNew.java
// Text fields and Java events
// <applet code=TextNew width=375
// height=125></applet>
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;

public class TextNew extends JApplet {
    JButton
        b1 = new JButton("Get Text"),
        b2 = new JButton("Set Text");
    JTextField
        t1 = new JTextField(30),
        t2 = new JTextField(30),
        t3 = new JTextField(30);
    String s = new String();
    UpperCaseDocument
        ucd = new UpperCaseDocument();

```

```

public void init() {
    t1.setDocument(ucd);
    ucd.addDocumentListener(new T1());
    b1.addActionListener(new B1());
    b2.addActionListener(new B2());
    DocumentListener dl = new T1();
    t1.addActionListener(dl);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(b1);
    cp.add(b2);
    cp.add(t1);
    cp.add(t2);
    cp.add(t3);
}
class T1 implements DocumentListener {
    public void changedUpdate(DocumentEvent e){}
    public void insertUpdate(DocumentEvent e){
        t2.setText(t1.getText());
        System.out.println("Text: " + t1.getText());
    }
    public void removeUpdate(DocumentEvent e){
        t2.setText(t1.getText());
    }
}
class T1A implements ActionListener {
    private int count = 0;
    public void actionPerformed(ActionEvent e) {
        t3.setText("t1 Action Event " + count++);
    }
}
class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        if(t1.getSelectedText() == null)
            s = t1.getText();
        else
            s = t1.getSelectedText();
        t1.setEditable(true);
    }
}
class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        ucd.setUpperCase(false);
        t1.setText("Inserted by Button 2: " + s);
        ucd.setUpperCase(true);
        t1.setEditable(false);
    }
}
public static void main(String[] args) {
    JApplet applet = new TextNew();
    JFrame frame = new JFrame("TextNew");
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
}

```

```

frame.getContentPane().add(applet);
frame.setSize(400,200);
applet.init();
applet.start();
frame.setVisible(true);
}
}

class UpperCaseDocument extends PlainDocument {
boolean upperCase = true;
public void setUpperCase(boolean flag) {
    upperCase = flag;
}
public void insertString(int offset,
    String string, AttributeSet attributeSet)
throws BadLocationException {
    if(upperCase)
        string = string.toUpperCase();
    super.insertString(offset,
        string, attributeSet);
}
} ///:~

```

TextField t3 е включено като място за докладване когато се задейства слушателят на действието на **TextField t1**. Ще видите че слушателят на действието на **TextField** се задейства когато натиснете клавиша “enter”.

TextField t1 има няколко присъединени слушателя. Слушателят **T1** копира всичкия текст от **t1** в **t2** и слушателят **T1K** прави всички знаци голяма буква. Ще забележите че двата работят заедно и ако добавите **T1K** слушател след като добавите слушателя **T1**, няма значение: всички знаци ще отидат в горен регистър и в двете текстови полета. Би изглеждало като чели всичките събития от клавиатурата се задействат преди **TextComponent** събитията и ако искате знаците в **t2** да си запазят регистра в който са въведени, ще трябва да свършите допълнителна работа.

T1K има и други интересни активности. Трябва да се детектира клавиша за изтриване на последния въведен символ (понеже управлявате всичко сега) и да се изпълни изтриването. Курсорът трябва явно да се сложи в края на полето; иначе няма да стане каквото се очаква. Накрая, за да се предотврати обработката на последния знак от механизма по подразбиране, събитието трябва да бъде “консумирано” чрез метода **consume()** който съществува за обектите-събития. Този метод казва на системата да не активира останалите обработчици за конкретното събитие.

Този пример също тихо демонстрира една от ползите от вътрешните класове. Забележете че във вътрешния клас:

```

class T1 implements TextListener {
    public void textValueChanged(TextEvent e) {
        t2.setText(t1.getText());
    }
}

```

t1 и **t2** не са членове на **T1**, но все пак са достъпни без специални мерки. Това е защото вътрешният клас автоматично получава манипулятор към създалия го (външен) клас, така че полетата на последния могат да се третират като свои. Както виждате, това е доста удобно.⁷

⁷ [And it also](#) solves the problem of “callbacks” without adding any awkward “method pointer” feature to Java.

ТЕКСТОВИ ПОЛЕТА

Най-значителната промяна на текстовите полета в Java 1.1 засяга скрол лентите. С конструктора на **TextArea** сега може да се задава дали **TextArea** ще има такива: вертикална, хоризонтална, двете или пък никаква. Този пример променя по-ранния Java 1.0 **TextArea1.java** за да покаже конструкторите за скрол ленти на Java 1.1:

```
//: c13:TextAreaNew.java
// Controlling scrollbars with JTextArea
// <applet code=TextAreaNew width=300 height=725>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;

public class TextAreaNew extends JApplet {
    JButton
        b1 = new JButton("Text Area 1"),
        b2 = new JButton("Text Area 2"),
        b3 = new JButton("Replace Text"),
        b4 = new JButton("Insert Text");
    JTextArea
        t1 = new JTextArea("t1", 1, 20),
        t2 = new JTextArea("t2", 4, 20),
        t3 = new JTextArea("t3", 1, 20),
        t4 = new JTextArea("t4", 10, 10),
        t5 = new JTextArea("t5", 4, 20),
        t6 = new JTextArea("t6", 10, 10);
    JScrollPane
        sp3 = new JScrollPane(t3,
            JScrollPane.VERTICAL_SCROLLBAR_NEVER,
            JScrollPane.HORIZONTAL_SCROLLBAR_NEVER),
        sp4 = new JScrollPane(t4,
            JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
            JScrollPane.HORIZONTAL_SCROLLBAR_NEVER),
        sp5 = new JScrollPane(t5,
            JScrollPane.VERTICAL_SCROLLBAR_NEVER,
            JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS),
        sp6 = new JScrollPane(t6,
            JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
            JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
    class B1L implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            t5.append(t1.getText() + "\n");
        }
    }
    class B2L implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            t2.setText("Inserted by Button 2");
            t2.append(": " + t1.getText());
            t5.append(t2.getText() + "\n");
        }
    }
    class B3L implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String s = "Replacement ";

```

```

        t2.replaceRange(s, 3, 3 + s.length());
    }
}

class B4L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t2.insert(" Inserted ", 10);
    }
}

public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    // Create Borders for components:
    Border brd = BorderFactory.createMatteBorder(
        1, 1, 1, 1, Color.black);
    t1.setBorder(brd);
    t2.setBorder(brd);
    sp3.setBorder(brd);
    sp4.setBorder(brd);
    sp5.setBorder(brd);
    sp6.setBorder(brd);
    // Initialize listeners and add components:
    b1.addActionListener(new B1L());
    cp.add(b1);
    cp.add(t1);
    b2.addActionListener(new B2L());
    cp.add(b2);
    cp.add(t2);
    b3.addActionListener(new B3L());
    cp.add(b3);
    b4.addActionListener(new B4L());
    cp.add(b4);
    cp.add(sp3);
    cp.add(sp4);
    cp.add(sp5);
    cp.add(sp6);
}
public static void main(String[] args) {
    JApplet applet = new TextAreaNew();
    JFrame frame = new JFrame("TextAreaNew");
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    frame.getContentPane().add(applet);
    frame.setSize(300,725);
    applet.init();
    applet.start();
    frame.setVisible(true);
}
} ///:~

```

Ще забележите че на скрол лентите може да се влияе само по време на конструирането на **TextArea**. Също, даже ако **TextArea** няма скрол лента, може да се сложи курсора така, че да се предизвика наличието на такава. (Може да видите такова поведение като си играете с примера.)

Отметки и радиобутони

Както беше споменато, отметките и радиобутоните са създадени от един клас, **Checkbox**, но радиобутоните са **Checkbox**ове сложени в **CheckboxGroup**. Във всеки от случаите интересно събитие е **ItemEvent**, за което създавате **ItemListener**.

Когато се разправяте с група отметки или радиобутони имате избор. Може или да създадете вътрешен клас който да обработва събитията за всеки отделен **Checkbox** или да създадете един вътрешен клас който да определя кой **Checkbox** е бил кликнат и да регистрирате обект от въпросния клас за всеки **Checkbox** обект. Следния пример показва двата подхода:

```
//: c13:RadioCheckNew.java
// Radio buttons and Check Boxes
// <applet code=RadioCheckNew
// width=325 height=100></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class RadioCheckNew extends JApplet {
    JTextField t = new JTextField(20);
    JCheckBox[] cb = {
        new JCheckBox("Check Box 1"),
        new JCheckBox("Check Box 2"),
        new JCheckBox("Check Box 3") };
    ButtonGroup group = new ButtonGroup();
    JRadioButton
        cb4 = new JRadioButton("four"),
        cb5 = new JRadioButton("five"),
        cb6 = new JRadioButton("six");
    // Checking the source:
    class ILCheck implements ItemListener {
        public void itemStateChanged(ItemEvent e) {
            for(int i = 0; i < cb.length; i++) {
                if(e.getSource().equals(cb[i])) {
                    t.setText("Check box " + (i + 1));
                    return;
                }
            }
        }
    }
    // vs. an individual class for each item:
    class IL4 implements ItemListener {
        public void itemStateChanged(ItemEvent e) {
            t.setText("Radio button four");
        }
    }
    class IL5 implements ItemListener {
        public void itemStateChanged(ItemEvent e) {
            t.setText("Radio button five");
        }
    }
    class IL6 implements ItemListener {
        public void itemStateChanged(ItemEvent e) {
            t.setText("Radio button six");
        }
    }
}
```

```

public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    t.setEditable(false);
    cp.add(t);
    ILCheck il = new ILCheck();
    for(int i = 0; i < cb.length; i++) {
        cb[i].addItemListener(il);
        cp.add(cb[i]);
    }
    group.add(cb4);
    group.add(cb5);
    group.add(cb6);
    cb4.addItemListener(new IL4());
    cb5.addItemListener(new IL5());
    cb6.addItemListener(new IL6());
    cp.add(cb4); cp.add(cb5); cp.add(cb6);
}
public static void main(String[] args) {
    JApplet applet = new RadioCheckNew();
    Frame frame = new Frame("RadioCheckNew");
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    frame.add(applet);
    frame.setSize(325, 150);
    applet.init();
    applet.start();
    frame.setVisible(true);
}
} //:~

```

ILCheck има предимството че автоматично работи когато махате или добавяте **checkbox**ове. Разбира се, това може да се направи и с радиобутони. То ще се използва, обаче, когато логиката ви е достатъчно обща за да се приложи този подход. Иначе ще завършите с каскади **if** оператори, сигурен признак че трябва да се ориентирате към независими слушателски класове.

Падащи списъци

Падащите списъци (**Choice**) в Java 1.1 също използват **ItemListeners** за да ви уведомят когато е променен изборът:

```

//: c13:ChoiceNew.java
// Drop-down lists (combo boxes)
// <applet code=ChoiceNew
// width=450 height=175></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ChoiceNew extends JApplet {
    String[] descriptions1 = { "Ebullient",
        "Obtuse", "Recalcitrant", "Brilliant" };

```

```

String[] descriptions2 = { "Somnescient",
    "Timorous", "Florid", "Putrescent" };
JTextArea t = new JTextArea(7, 40);
JComboBox c = new JComboBox(descriptions1);
JButton b = new JButton("Add items");
int count = 0;
public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    t.setLineWrap(true);
    t.setEditable(false);
    cp.add(t);
    cp.add(c);
    cp.add(b);
    c.addItemListener(new CL());
    b.addActionListener(new BL());
}
class CL implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
        t.setText("index: " + c.getSelectedIndex()
            + " " + e.toString());
    }
}
class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        if(count < descriptions2.length)
            c.addItem(descriptions2(count++));
        if(count >=descriptions2.length)
            b.setEnabled(false);
    }
}
public static void main(String[] args) {
    JApplet applet = new ChoiceNew();
    Frame frame = new Frame("ChoiceNew");
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    frame.add(applet);
    frame.setSize(450,200);
    applet.init();
    applet.start();
    frame.setVisible(true);
}
} //:~

```

Нищо друго тук не е особено ново (освен че Java 1.1 има значително по-малко бъгове в UI класовете).

СПИСЪЦИ

Припомняме си че един от проблемите с дизайна в Java 1.0 на **List** че изискваше допълнителна работа за да се направи да работи както трябва: да реагира на единично кликване върху един елемент на списъка. Java 1.1 е решил този проблем:

```

//: c13>ListNew.java
// <applet code=ListNew width=250
// height=250 > </applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;

public class ListNew extends JApplet {
    String[] flavors = { "Chocolate", "Strawberry",
        "Vanilla Fudge Swirl", "Mint Chip",
        "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie" };
    DefaultListModel lItems=new DefaultListModel();
    JList lst = new JList(lItems);
    JTextArea t = new JTextArea(flavors.length,20);
    JButton b = new JButton("Add Item");
    ActionListener bl = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            if(count < flavors.length) {
                lItems.add(0, flavors(count++));
            } else {
                // Disable, since there are no more
                // flavors left to be added to the List
                b.setEnabled(false);
            }
        }
    };
    ListSelectionListener ll =
    new ListSelectionListener() {
        public void valueChanged(
            ListSelectionEvent e) {
            t.setText("");
            Object[] items=lst.getSelectedValues();
            for(int i = 0; i < items.length; i++)
                t.append(items(i) + "\n");
        }
    };
    int count = 0;
    public void init() {
        Container cp = getContentPane();
        t.setEditable(false);
        cp.setLayout(new FlowLayout());
        // Create Borders for components:
        Border brd = BorderFactory.createMatteBorder(
            1, 1, 2, 2, Color.black);
        lst.setBorder(brd);
        t.setBorder(brd);
        // Add the first four items to the List
        for(int i = 0; i < 4; i++)
            lItems.addElement(flavors(count++));
        // Add items to the Content Pane for Display
        cp.add(t);
        cp.add(lst);
        cp.add(b);
        // Register event listeners
    }
}

```

```

    lst.addListSelectionListener(l);
    b.addActionListener(bl);
}
public static void main(String[] args) {
    JApplet applet = new ListNew();
    JFrame frame = new JFrame("ListNew");
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    frame.getContentPane().add(applet);
    frame.setSize(250, 350);
    applet.init();
    applet.start();
    frame.setVisible(true);
}
} //:~

```

Може да се види, че не е необходима допълнителна логика за поддържането на единично кликване върху елемент на списъка. Просто присъединявате слушател както за всяко друго нещо.

Менюта

Обработката на събития в менюта изглежда се възползва от събитийния модел в Java 1.1, но подходът на Java към менютата е все още объркан и изисква много ръчно кодиране. Точната следа за менютата като чели са ресурси, а не много код. Помните че инструментите за правене на програми обикновено ще правят менютата заради вас, така че напрежението би се понамалило при такъв подход (доколкото те биха обслужвали и поддръжката!).

Освен това ще намерите, че събитията за менюта са несъстоятелни и могат да доведат до конфузии: **MenuItem**ите използват **ActionListener**и, но **CheckboxMenuItem**ите използват **ItemListener**и. **Menu** обектите могат също да поддържат **ActionListener**и, но обикновено това не е полезно. Изобщо, ще вържете слушатели към всеки **MenuItem** или **CheckboxMenuItem**, но следния пример (преработен от по-раншна версия) също показва начин за хващане на много компоненти на меню в един слушателски клас. Както ще видите, може би не си заслужава суматохата за да се направи така.

```

//: c13:MenuNew.java
// Menu shortcuts and action commands
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MenuNew extends JFrame {
    String[] flavors = { "Chocolate", "Strawberry",
        "Vanilla Fudge Swirl", "Mint Chip",
        "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie" };
    JTextField t = new JTextField("No flavor", 30);
    JMenuBar mb1 = new JMenuBar();
    JMenu
    f = new JMenu("File"),
    m = new JMenu("Flavors"),
    s = new JMenu("Safety");

```

```

// Alternative approach:
JCheckBoxMenuItem() safety = {
    new JCheckBoxMenuItem("Guard"),
    new JCheckBoxMenuItem("Hide")
};
JMenuItem() file = {
    // No menu shortcut:
    new JMenuItem("Open"),
    // Adding a menu shortcut is very simple:
    new JMenuItem("Exit", KeyEvent.VK_E)
};
// A second menu bar to swap to:
JMenuBar mb2 = new JMenuBar();
JMenu fooBar = new JMenu("fooBar");
JMenuItem() other = {
    new JMenuItem("Foo"),
    new JMenuItem("Bar"),
    new JMenuItem("Baz"),
};
JButton b = new JButton("Swap Menus");
public MenuNew() {
    super("MenuNew");
    ML ml = new ML();
    CMIL cmil = new CMIL();
    safety(0).setActionCommand("Guard");
    safety(0).addItemListener(cmil);
    safety(1).setActionCommand("Hide");
    safety(1).addItemListener(cmil);
    file(0).setActionCommand("Open");
    file(0).addActionListener(ml);
    file(1).setActionCommand("Exit");
    file(1).addActionListener(ml);
    other(0).addActionListener(new FooL());
    other(1).addActionListener(new BarL());
    other(2).addActionListener(new BazL());
    FL fl = new FL();
    for(int i = 0; i < flavors.length; i++) {
        JMenuItem mi = new JMenuItem(flavors(i));
        mi.addActionListener(fl);
        m.add(mi);
        // Add separators at intervals:
        if((i+1) % 3 == 0)
            m.addSeparator();
    }
    for(int i = 0; i < safety.length; i++)
        s.add(safety(i));
    f.add(s);
    for(int i = 0; i < file.length; i++)
        f.add(file(i));
    mb1.add(f);
    mb1.add(m);
    setJMenuBar(mb1);
    t.setEditable(false);
    Container cp = getContentPane();
    cp.add(t, BorderLayout.CENTER);
    // Set up the system for swapping menus:
    b.addActionListener(new BL());
}

```

```

cp.add(b, BorderLayout.NORTH);
for(int i = 0; i < other.length; i++)
    fooBar.add(other(i));
mb2.add(fooBar);
}
class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JMenuBar m = getJMenuBar();
        setJMenuBar(m == mb1 ? mb2 : mb1);
        validate(); // Refresh the frame ////////// Necessary???
    }
}
class ML implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JMenuItem target = (MenuItem)e.getSource();
        String actionCommand =
            target.getActionCommand();
        if(actionCommand.equals("Open")) {
            String s = t.getText();
            boolean chosen = false;
            for(int i = 0; i < flavors.length; i++)
                if(s.equals(flavors(i))) chosen = true;
            if(!chosen)
                t.setText("Choose a flavor first!");
            else
                t.setText("Opening "+s+". Mmm, mm!");
        } else if(actionCommand.equals("Exit")) {
            // This trick won't work with applets
            // because MenuNew.this doesn't produce
            // a Window if this is a JApplet:
            dispatchEvent(
                new WindowEvent(MenuNew.this,
                    WindowEvent.WINDOW_CLOSING));
        }
    }
}
class FL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JMenuItem target = (MenuItem)e.getSource();
        t.setText(target.getText());
    }
}
// Alternatively, you can create a different
// class for each different MenuItem. Then you
// Don't have to figure out which one it is:
class FooL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t.setText("Foo selected");
    }
}
class BarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t.setText("Bar selected");
    }
}
class BazL implements ActionListener {
    public void actionPerformed(ActionEvent e) {

```

```

        t.setText("Baz selected");
    }
}

class CMIL implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
        JCheckBoxMenuItem target =
            (JCheckBoxMenuItem)e.getSource();
        String actionCommand =
            target.getActionCommand();
        if(actionCommand.equals("Guard"))
            t.setText("Guard the Ice Cream! " +
                "Guarding is " + target.getState());
        else if(actionCommand.equals("Hide"))
            t.setText("Hide the Ice Cream! " +
                "Is it cold? " + target.getState());
    }
}

public static void main(String[] args) {
    JFrame frame = new MenuNew();
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    frame.setSize(300, 100);
    frame.setVisible(true);
}
} //:~
```

Този код е подобен на кода за предишната (на Java 1.0) версия докато се стигне до **init()** метода. Тука може да видите **ItemListener** и **ActionListener** присъединени към различни компоненти на менюта.

Java 1.1 поддържа "menu shortcuts," така че може да изберете елемент на меню чрез клавиатурата освен с мишката. Тия са доста прости; просто използвате претоварен конструктор на **MenuItem** който взема като втори аргумент обект **MenuShortcut**. Конструкторът за **MenuShortcut** взема клавищът който е интересния, който магически се появява на елемента когато менюто се разгъне. Горният пример добавя Control-E към елемента "Exit" на менюто.

Може също да видите използването на **setActionCommand()**. То изглежда малко странно защото във всеки отделен случай "командата за действие" е точно същата както етикета на компонента от менюто. Защо да не се използва просто етикета наместо този друг стринг? Проблемът е в интернационализацията. Ако искате да направите тази програма за друг език, ще искате да смените само етикета в менюто, а не целия код, което несъмнено би донесло грешки. Така че за улеснение на кода който проверява, командалата трябва да остава неизменна при промяна на етикета. Кодът работи само с командите, така че не зависи от етикетите. Забележете че в тази програма не всички компоненти имат команди, така че не всички се проверяват за такива.

Повечето конструктор е непроменен, с изключение на двете извиквания за добавяне на слушатели. Повечето работа е в слушателите. В **BL** става смяна на **MenuBar**ове както в предишния пример. В **ML** е приет подхода "изясни кой позвъни" за намиране съдържанието на **ActionEvent** и кастинг към **MenuItem**, после вземане на командния стринг и подаването му през каскадирани **if** оператори. Повечето е същото като преди, но забележете че ако е избран "Exit" нов **WindowEvent** се създава, подавайки манипулятор към обграждащия обект (**MenuNew.this**) и създавайки събитие **WINDOW_CLOSING**. Това се дава на метода

dispatchEvent() на обекта от обгръщащия клас, който завършва с **windowClosing()** вътре в слушателя на прозореца във **Frame** (този слушател е създаден като анонимен вътрешен клас, в **main()**), точно както ако съобщението бе генерирано по "нормален" начин. Чрез този механизъм може да се диспечира всякакво съобщение при всякакви условия, така че той е твърде мощен.

Слушателят **FL** е прости чак макар и да обработва всичките вкусове в менюто на вкусовете. Този подход е полезен ако логиката ви е проста, но изобщо ще трябва да използвате подхода с **FooL**, **BarL** и **BazL**, където всеки е присъединен към един компонент на меню така че не е необходима допълнителна логика и се знае кой е извикал слушателя. Макар и да се генерира изобилие от класове при този начин, кодът има тенденция да е по-малък.

ДИАЛОГОВИ КУТИИ

Това е преписано направо предишното **TicTacToe.java**. В тази версия обаче всичко е сложено във вътрешен клас. Макар и това напълно да елиминира нуждата да се следи кой е породил даден обект, както беше в няя версия на **TicTacToe.java**, може би сега концепцията за вътрешните класове отива твърде далеч. Вътрешните класове са вместени на четири нива! Това е случай на проектиране, когато трябва да решите дали ползите от вътрешните класове си заслужават повишена сложност. Освен това когато създавате **не-static** вътрешен клас го връзвате с външния клас. Понякога един самостоятелен клас би могъл по-лесно да бъде използван повторно.

```
//: c13:TicTacToelnner.java
// TicTacToe.java with heavy use of inner classes
import javax.swing.*;
import java.awt.*;
import java.awt.event;

public class TicTacToelnner extends JFrame {
    JTextField
    rows = new JTextField("3"),
    cols = new JTextField("3");
    public TicTacToelnner() {
        setTitle("Toe Test");
        JPanel p = new JPanel();
        p.setLayout(new GridLayout(2,2));
        p.add(new JLabel("Rows", JLabel.CENTER));
        p.add(rows);
        p.add(new JLabel("Columns", JLabel.CENTER));
        p.add(cols);
        Container cp = getContentPane();
        cp.add(p, BorderLayout.NORTH);
        JButton b = new JButton("go");
        b.addActionListener(new BL());
        cp.add(b, BorderLayout.SOUTH);
    }
    static final int BLANK = 0, XX = 1, OO = 2;
    class ToeDialog extends JDialog {
        int turn = XX; // Start with x's turn
        // w = number of cells wide
        // h = number of cells high
        public ToeDialog(int w, int h) {
            super(TicTacToelnner.this,
                  "The game itself", false);
            Container cp = getContentPane();
```

```

cp.setLayout(new GridLayout(w, h));
for(int i = 0; i < w * h; i++)
    cp.add(new ToeButton());
setSize(w * 50, h * 50);
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e){
        dispose();
    }
});
}
class ToeButton extends JPanel {
    int state = BLANK;
    ToeButton() {
        addMouseListener(new ML());
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int x1 = 0;
        int y1 = 0;
        int x2 = getSize().width - 1;
        int y2 = getSize().height - 1;
        g.drawRect(x1, y1, x2, y2);
        x1 = x2/4;
        y1 = y2/4;
        int wide = x2/2;
        int high = y2/2;
        if(state == XX) {
            g.drawLine(x1, y1,
                       x1 + wide, y1 + high);
            g.drawLine(x1, y1 + high,
                       x1 + wide, y1);
        }
        if(state == OO) {
            g.drawOval(x1, y1,
                       x1 + wide/2, y1 + high/2);
        }
    }
    class ML extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
            if(state == BLANK) {
                state = turn;
                turn = (turn == XX ? OO : XX);
            }
            else
                state = (state == XX ? OO : XX);
            repaint();
        }
    }
}
class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JDialog d = new ToeDialog(
            Integer.parseInt(rows.getText()),
            Integer.parseInt(cols.getText()));
        d.setVisible(true);
    }
}

```

```

}
public static void main(String[] args) {
    JFrame frame = new TicTacToeInner();
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    frame.setSize(200,100);
    frame.setVisible(true);
}
} //:~

```

Понеже **static**ите могат да бъдат само във външното ниво клас, вътрешните класове не могат да имат **static** данни или **static** вътрешни класове.

Избиране на изглед и усещане

Един от много интересните аспекти на Swing е "Прикачвани изглед и усещане." Това позволява вашата програма да имитира изгледа и усещането за различни операционни среди. Може даже да правите всички видове играви неща като например промяна на вида и усещането в процеса на изпълнение на програмата. Обикновено обаче искате да правите едното от двете неща: или да се избере "междуплатформен" изглед и усещане (което е "металик" при Swing) или пък да изберете изгледа и усещането за операционната система с която се работи в момента, така че вашите Java програми да изглеждат създадени специално за нея операционна система. Кодът за избиране на тези неща е доста прост, но трябва да се изпълни преди да се създаде визуален компонент, понеже това ще стане на базата на избрания изглед и няма да бъде променено просто защото сте сменили изгледа по време на изпълнение на програмата (такъв процес е по-необичаен и малко разпространен, та ще се насочите към книги специално за Swing за него).

Фактически ако искате да използвате многоплатформенния ("металик") вариант който е характерен за Swing програми не трябва да правите нищо – той е по подразбиране. Но ако искате да имате изгледа на операционната среда в която се работи, просто вмъквате следния код, някъде в началото на **main()** преди да се създадат компоненти:

```

try {
    UIManager.setLookAndFeel(UIManager.
        getSystemLookAndFeelClassName());
} catch (Exception ex) { }

```

Нищо няма в **catch** клаузата понеже **UIManager** ще използва междуплатформения изглед ако опитите да се избере друг се провалят. По време на дебъгинг обаче изключението може да бъде доста пълзно.

Ето програма която приема аргумент на командния ред за избиране на изглед и показва как изглежда избирането на компоненти:

```

//: c13:LookAndFeel.java
// Selecting different looks & feels
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class LookAndFeel extends JFrame {

```

```

String[] choices = {
    "eeny", "meeny", "minie", "moe", "toe", "you"
};
Component[] samples = {
    new JButton("JButton"),
    new JTextField("JTextField"),
    new JLabel("JLabel"),
    new JCheckBox("JCheckBox"),
    new JRadioButton("Radio"),
    new JComboBox(choices),
    new JList(choices),
};
public LookAndFeel() {
    super("Look And Feel");
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < samples.length; i++)
        cp.add(samples[i]);
}
private static void usageError() {
    System.out.println(
        "Usage:LookAndFeel (cross | system | motif)");
    System.exit(1);
}
public static void main(String[] args) {
    if(args.length == 0) usageError();
    if(args[0].equals("cross")) {
        try {
            UIManager.setLookAndFeel(UIManager.
                getCrossPlatformLookAndFeelClassName());
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    } else if(args[0].equals("system")) {
        try {
            UIManager.setLookAndFeel(UIManager.
                getSystemLookAndFeelClassName());
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    } else if(args[0].equals("motif")) {
        try {
            UIManager.setLookAndFeel("com.sun.java.+" +
                "swing.plaf.motif.MotifLookAndFeel");
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    } else usageError();
    // Note the look & feel must be set before
    // any components are created.
    JFrame frame = new LookAndFeel();
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
}

```

```
    frame.setSize(300, 200);
    frame.setVisible(true);
}
} //:/~
```

Вижда се че едната възможност е да се зададе явно стринг който да показва изгледа, както се вижда с **MotifLookAndFeel**. Обаче то и "металик" -а са единствените, които могат да се използват на всяка платформа; Макар че има стрингове за Windows и Macintosh изгледи, те могат да се използват само със съответните платформи (правят се когато извикате **getSystemLookAndFeelClassName()** и сте на съответната машина).

Възможно е също да се създаде специален пакет за изглед и усещане, например ако работите за компания която си има специфични изисквания за вида на нещата. Това е голяма задача и е далеч извън обхвата на тази книга (фактически ще откриете, че е извън обхвата на много книги, посветени на Swing!).

Свързване на събитията динамично

Една от ползите от новия AWT модел на събитията е гъвкавостта. При стария модел бяхте принудени да вградите твърдо логиката на вашата програма, но при новия може да вмъквате или махате обработчици с едно извикване на метод. Следния пример демонстрира това:

```
//: c13:DynamicEvents.java
// You can change event behavior dynamically.
// Also shows multiple actions for an event.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class DynamicEvents extends JFrame{
    ArrayList v = new ArrayList();
    int i = 0;
    JButton
        b1 = new JButton("Button1"),
        b2 = new JButton("Button2");
    public DynamicEvents() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        b1.addActionListener(new B());
        b1.addActionListener(new B1());
        b2.addActionListener(new B());
        b2.addActionListener(new B2());
        cp.add(b1);
        cp.add(b2);
    }
    class B implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            System.out.println("A button was pressed");
        }
    }
    class CountListener implements ActionListener {
        int index;
        public CountListener(int i) { index = i; }
```

```

public void actionPerformed(ActionEvent e) {
    System.out.println(
        "Counted Listener " + index);
}
}

class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button 1 pressed");
        ActionListener a = new CountListener(i++);
        v.add(a);
        b2.addActionListener(a);
    }
}

class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button2 pressed");
        int end = v.size() - 1;
        if(end >= 0) {
            b2.removeActionListener(
                (ActionListener)v.get(end));
            v.remove(end);
        }
    }
}

public static void main(String[] args) {
    JFrame frame = new DynamicEvents();
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    frame.setSize(300, 100);
    frame.setVisible(true);
}
} ///:~

```

Новите неща в този пример са:

1. Има повече от един слушател закачен за всеки **Button**. Обикновено компонентите обслужват събитията като *multicast*, което значи че регистрирате много слушатели за едно събитие. В специалните компоненти в които събитието се обслужва като *unicast* ще получите **TooManyListenersException**.
2. По време на изпълнението на програмата слушатели динамично се добавят и махат от **Button b2**. Добавянето става по начина който сте виждали преди, но всеки компонент също има **removeXXXListener()** метод за махране на всеки тип слушател.

Този вид гъвкавост дава много повече мощ на вашето програмиране.

Трябва да знаете, че слушателите на събития не се викат непременно в реда, в който са добавени (макар и много реализации да правят именно така).

Отделяне на основната логика от UI логиката

Изобщо ще искате да направите така своите класове, че всеки да прави "само едно нещо." Това е особено важно когато е замесен потребителския интерфейс, понеже е лесно да загърнете "каквото правите" с "как го идобразявате." Този начин на свързване предотвратява повторното използване на кода. Много по-желателно е да разделите кода на "основната логика" от GUI. По този начин не само можете да използвате основната логика по-лесно, същото се отнася и за GUI.

Друг момент са многоредовите системи, където "основните обекти" са резидентни на съвсем отделна машина. Това централно място на бизнеслогиката прави възможно всички промени да са моментално ефективни за всички следващи транзакции, а с това са примамлив начин за оформяне на системата. Обаче тези бизнес обекти може да се използват с най-различни приложения и не трябва да са обвързани с някой конкретен режим или дисплей. Те само трябва да изпълняват основните операции и нищо повече.

Следния пример показва колко е лесно да се разделит бизнес логиката от GUI кода:

```
//: c13:Separation.java
// Separating GUI logic and business objects
// <applet code=Separation
// width=250 height=100> </applet>
import javax.swing.*;
import java.awt.*;
import javax.swing.event.*;
import java.awt.event.*;
import java.applet.*;

class BusinessLogic {
    private int modifier;
    BusinessLogic(int mod) {
        modifier = mod;
    }
    public void setModifier(int mod) {
        modifier = mod;
    }
    public int getModifier() {
        return modifier;
    }
    // Some business operations:
    public int calculation1(int arg) {
        return arg * modifier;
    }
    public int calculation2(int arg) {
        return arg + modifier;
    }
}

public class Separation extends JApplet {
    JTextField t = new JTextField(15),
    mod = new JTextField(15);
    BusinessLogic bl = new BusinessLogic(2);
    JButton
```

```

calc1 = new JButton("Calculation 1"),
calc2 = new JButton("Calculation 2");
public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    calc1.addActionListener(new Calc1L());
    calc2.addActionListener(new Calc2L());
    JPanel p1 = new JPanel();
    p1.add(calc1);
    p1.add(calc2);
    cp.add(p1);
    mod.getDocument();
    addDocumentListener(new ModL());
    JPanel p2 = new JPanel();
    p2.add(new JLabel("Modifier:"));
    p2.add(mod);
    cp.add(p2);
}
static int getValue(JTextField tf) {
    try {
        return Integer.parseInt(tf.getText());
    } catch(NumberFormatException e) {
        return 0;
    }
}
class Calc1L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t.setText(Integer.toString(
            bl.calculation1(getValue(t))));
    }
}
class Calc2L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t.setText(Integer.toString(
            bl.calculation2(getValue(t))));
    }
}
// If you want something to happen whenever
// a JTextField changes, add this listener:
class ModL implements DocumentListener {
    public void changedUpdate(DocumentEvent e) {}
    public void insertUpdate(DocumentEvent e) {
        bl.setModifier(getValue(mod));
    }
    public void removeUpdate(DocumentEvent e) {
        bl.setModifier(getValue(mod));
    }
}
public static void main(String[] args) {
    JApplet applet = new Separation();
    JFrame frame = new JFrame("Separation");
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        }
    )
}

```

```

    });
    frame.getContentPane().add(applet);
    frame.setSize(300, 150);
    applet.init();
    applet.start();
    frame.setVisible(true);
}
} //:~

```

Може да видите че **BusinessLogic** е праволинеен клас който изпълнява задачите си без никакъв намек че може да бъде използван в GUI обкръжение. Просто си върши работата.

Separation следи за всички UI детайли, говори на **BusinessLogic** само чрез неговия **public** интерфейс. Всички операции са концентрирани около предаването на информация към и от UI и обекта **BusinessLogic** съответно. Така **Separation**, на свой ред, просто си върши работата. Тъй като **Separation** знае само че говори на **BusinessLogic** обект (тоест, не са много свързани), може да се направи да говори на други видове обекти без много мъки.

В термините на отделяне на UI също се стига до облекчаването на живота и когато имате да приспособявате наследе код към Java.

Препоръчвани подходи при кодирането

Новият модел на събитията, заедно с използването на стария и черти от стария в някои библиотеки, води до нов момент на смущение. Сега има даже повече начини да се напише лош код. За нещастие лош код се показва в книги и статии, даже в документация и примери разпространявани от Sun! В тази секция ще разгледаме някои недоразбрани неща относно това какво да се прави и да не се прави с AWT, ще завършим като видим че с изключение на смекчаващи обстоятелства винаги може да се използват слушателски класове (написани като вътрешни класове) за да се решат нуждите за обработка на събития. Тъй като това е същевременно и най-лесния и ясен подход, облекчение за вас ще бъде да го научите.

Преди да погледнем каквото и да било, трябва да знаете че макар и Java 1.1 да е съвместим назад с Java 1.0 (тоест, може да компилирате и пускате 1.0 програми с 1.1), не може да смесвате моделите на събитията в една и съща програма. Тоест не може да използвате стария **action()** метод в програма, в която използвате слушатели. Това може да е проблем в голяма програма, където трябва да се интегрира стар с нов код, понеже трябва да решите дали ще използвате стария, труден за поддръжка подход или да осъвремените старата програма. Няма да е трудно да решите, защото новият метод твърде много превъзхожда стария.

Чертата: добрият начин да го правите

За да имате база за сравнение, ето пример показващ препоръчвания подход. Сега той трябва да е достатъчно ясен и разбираем:

```

//: c13:GoodIdea.java
// The best way to design classes using the
// Java event model: use an inner class for
// each different event. This maximizes
// flexibility and modularity.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

```

```

import java.util.*;

public class GoodIdea extends JFrame {
    JButton
    b1 = new JButton("Button 1"),
    b2 = new JButton("Button 2");
    public GoodIdea() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        b1.addActionListener(new B1L());
        b2.addActionListener(new B2L());
        cp.add(b1);
        cp.add(b2);
    }
    public class B1L implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            System.out.println("Button 1 pressed");
        }
    }
    public class B2L implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            System.out.println("Button 2 pressed");
        }
    }
    public static void main(String[] args) {
        JFrame frame = new GoodIdea();
        frame.addWindowListener(
            new WindowAdapter() {
                public void windowClosing(WindowEvent e){
                    System.out.println("Window Closing");
                    System.exit(0);
                }
            });
        frame.setSize(300,200);
        frame.setVisible(true);
    }
} //:~

```

Достатъчно тривиално е: всеки бутоң има собствен слушател който извежда нещо на конзолата. Но забележете че няма никакъв **if** оператор в цялата програма, или пък такъв който казва, "Чудя се кой задейства това събитие." Всяко парче код е в зависимост от правенето, не от проверяването на тип. Това е най-добрия начин да напишете кода си; не само е по-лесно за концепция, но също много по-лесно за четене и поддръжка. Копирането в спомагателна памет и прехвърлянето после в нова програма е също по-лесно.

Реализиране на главния клас като слушател

Първата лоша идея е общоприет и препоръчван подход. Прави се главния клас (типовично **Applet** или **Frame**, но може да бъде всеки клас) да реализира различни слушатели. Ето пример:

```

//: c13:BadIdea1.java
// Some literature recommends this approach, but
// it's missing the point of the Java event model
import javax.swing.*;

```

```

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

public class BadIdea1 extends JFrame
    implements ActionListener, WindowListener {
    JButton
        b1 = new JButton("Button 1"),
        b2 = new JButton("Button 2");
    public BadIdea1() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        addWindowListener(this);
        b1.addActionListener(this);
        b2.addActionListener(this);
        cp.add(b1);
        cp.add(b2);
    }
    public void actionPerformed(ActionEvent e) {
        Object source = e.getSource();
        if(source == b1)
            System.out.println("Button 1 pressed");
        else if(source == b2)
            System.out.println("Button 2 pressed");
        else
            System.out.println("Something else");
    }
    public void windowClosing(WindowEvent e) {
        System.out.println("Window Closing");
        System.exit(0);
    }
    public void windowClosed(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
    public void windowOpened(WindowEvent e) {}

    public static void main(String[] args) {
        JFrame frame = new BadIdea1();
        frame.setSize(300,200);
        frame.setVisible(true);
    }
} //:~

```

Използването се показва с три реда:

```

addWindowListener(this);
b1.addActionListener(this);
b2.addActionListener(this);

```

Понеже **BadIdea1** реализира **ActionListener** и **WindowListener**, тези линии несъмнено са приемливи, а още сте привърженик на модата да се правят по-малко класове за да се намалят достъпите до сървъра, изглежда и добра идея. Обаче:

1. Java 1.1 поддържа JAR файлове така че всичките ви файлове могат да бъдат сложени в един компресиран JAR архив който изисква един достъп до сървъра. Вече не е необходимо да намалявате броя на класовете за ефективност в Internet.

2. Кодът по-горе е много по-малко модулен, така че не е лесно да се грабне и сложи на друго място. Забележете че не само е необходимо да приложите различни интерфейси във вашия главен клас, но и в **actionPerformed()** трябва да разпознаете каква акция е предприета чрез каскадирани **if** оператори. Не само че това води назад, отдалечава от модела със слушатели, но и не може лесно да използвате повторно метода **actionPerformed()** понеже е специфичен за конкретното приложение. Вижте контраста с **GoodIdea.java**, където може да грабнете който и да е отделен клас и да го сложите някъде без много връява. Плюс че може да регистрирате няколко слушателя с едно събитие, което дава по-голяма модулност отколкото единичния слушателски клас.

Смесване на подходите

Втората лоша идея е да се смесят двета подхода: да се използват вътрешни класове, но също и да се сложат един или повече слушатели в главен клас. Този подход се появя в документацията без всякакви обяснения и аз само мога да предполагам, че авторите са смятали, че трябва да използват различни подходи за различните цели. Но вие не вие ще използвате изключително вътрешни класове.

```
//: c13:BadIdea2.java
// An improvement over BadIdea1.java, since it
// uses the WindowAdapter as an inner class
// instead of implementing all the methods of
// WindowListener, but still misses the
// valuable modularity of inner classes
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class BadIdea2 extends JFrame
    implements ActionListener {
    JButton
        b1 = new JButton("Button 1"),
        b2 = new JButton("Button 2");
    public BadIdea2() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        addWindowListener(new WL());
        b1.addActionListener(this);
        b2.addActionListener(this);
        cp.add(b1);
        cp.add(b2);
    }
    public void actionPerformed(ActionEvent e) {
        Object source = e.getSource();
        if(source == b1)
            System.out.println("Button 1 pressed");
        else if(source == b2)
            System.out.println("Button 2 pressed");
        else
            System.out.println("Something else");
    }
    class WL extends WindowAdapter {
        public void windowClosing(WindowEvent e) {
            System.out.println("Window Closing");
            System.exit(0);
        }
    }
}
```

```

        }
    }

public static void main(String[] args) {
    JFrame frame = new BadIdea2();
    frame.setSize(300,200);
    frame.setVisible(true);
}

} //:~
```

Тъй като **actionPerformed()** е все още силно свързан към главния клас, трудно е да се използва този код. Също е по-трудно за четене отколкото при другия подход.

Няма причини да използвате старото мислене за събитията в Java 1.1 – така че защо да го правите?

Наследяване на компонент

Друго място където често ще виждате вариации на стария начин на правене на нещата е когато се създава нов тип компонент. Ето пример който показва че тук, пак, работи и новия начин:

```

//: c13:GoodTechnique.java
// Your first choice when overriding components
// should be to install listeners. The code is
// much safer, more modular and maintainable.
import java.awt.*;
import java.awt.event.*;

class Display {
    public static final int
        EVENT = 0, COMPONENT = 1,
        MOUSE = 2, MOUSE_MOVE = 3,
        FOCUS = 4, KEY = 5, ACTION = 6,
        LAST = 7;
    public String[] evnt;
    Display() {
        evnt = new String[LAST];
        for(int i = 0; i < LAST; i++)
            evnt[i] = new String();
    }
    public void show(Graphics g) {
        for(int i = 0; i < LAST; i++)
            g.drawString(evnt[i], 0, 10 * i + 10);
    }
}

class EnabledPanel extends Panel {
    Color c;
    int id;
    Display display = new Display();
    public EnabledPanel(int i, Color mc) {
        id = i;
        c = mc;
        setLayout(new BorderLayout());
        add(new MyButton(), BorderLayout.SOUTH);
        addComponentListener(new CL());
        addFocusListener(new FL());
```

```

    addKeyListener(new KL());
    addMouseListener(new MLO());
    addMouseMotionListener(new MML());
}
// To eliminate flicker:
public void update(Graphics g) {
    paint(g);
}
public void paint(Graphics g) {
    g.setColor(c);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
    g.setColor(Color.black);
    display.show(g);
}
// Don't need to enable anything for this:
public void processEvent(AWTEvent e) {
    display.evnt(Display.EVENT)= e.toString();
    repaint();
    super.processEvent(e);
}
class CL implements ComponentListener {
    public void componentMoved(ComponentEvent e){
        display.evnt(Display.COMPONENT) =
            "Component moved";
        repaint();
    }
    public void
        componentResized(ComponentEvent e) {
            display.evnt(Display.COMPONENT) =
                "Component resized";
            repaint();
    }
    public void
        componentHidden(ComponentEvent e) {
            display.evnt(Display.COMPONENT) =
                "Component hidden";
            repaint();
    }
    public void componentShown(ComponentEvent e){
        display.evnt(Display.COMPONENT) =
            "Component shown";
        repaint();
    }
}
class FL implements FocusListener {
    public void focusGained(FocusEvent e) {
        display.evnt(Display.FOCUS) =
            "FOCUS gained";
        repaint();
    }
    public void focusLost(FocusEvent e) {
        display.evnt(Display.FOCUS) =
            "FOCUS lost";
        repaint();
    }
}

```

```

class KL implements KeyListener {
    public void keyPressed(KeyEvent e) {
        display.evnt(Display.KEY) =
            "KEY pressed: ";
        showCode(e);
    }
    public void keyReleased(KeyEvent e) {
        display.evnt(Display.KEY) =
            "KEY released: ";
        showCode(e);
    }
    public void keyTyped(KeyEvent e) {
        display.evnt(Display.KEY) =
            "KEY typed: ";
        showCode(e);
    }
    void showCode(KeyEvent e) {
        int code = e.getKeyCode();
        display.evnt(Display.KEY) +=
            KeyEvent.getKeyText(code);
        repaint();
    }
}
class ML implements MouseListener {
    public void mouseClicked(MouseEvent e) {
        requestFocus(); // Get FOCUS on click
        display.evnt(Display.MOUSE) =
            "MOUSE clicked";
        showMouse(e);
    }
    public void mousePressed(MouseEvent e) {
        display.evnt(Display.MOUSE) =
            "MOUSE pressed";
        showMouse(e);
    }
    public void mouseReleased(MouseEvent e) {
        display.evnt(Display.MOUSE) =
            "MOUSE released";
        showMouse(e);
    }
    public void mouseEntered(MouseEvent e) {
        display.evnt(Display.MOUSE) =
            "MOUSE entered";
        showMouse(e);
    }
    public void mouseExited(MouseEvent e) {
        display.evnt(Display.MOUSE) =
            "MOUSE exited";
        showMouse(e);
    }
    void showMouse(MouseEvent e) {
        display.evnt(Display.MOUSE) +=
            ", x = " + e.getX() +
            ", y = " + e.getY();
        repaint();
    }
}

```

```

class MML implements MouseMotionListener {
    public void mouseDragged(MouseEvent e) {
        display.evnt(Display.MOUSE_MOVE) =
            "MOUSE dragged";
        showMouse(e);
    }
    public void mouseMoved(MouseEvent e) {
        display.evnt(Display.MOUSE_MOVE) =
            "MOUSE moved";
        showMouse(e);
    }
    void showMouse(MouseEvent e) {
        display.evnt(Display.MOUSE_MOVE) +=
            ", x = " + e.getX() +
            ", y = " + e.getY();
        repaint();
    }
}
}

class MyButton extends Button {
    int clickCounter;
    String label = "";
    public MyButton() {
        addActionListener(new AL());
    }
    public void paint(Graphics g) {
        g.setColor(Color.green);
        Dimension s = getSize();
        g.fillRect(0, 0, s.width, s.height);
        g.setColor(Color.black);
        g.drawRect(0, 0, s.width - 1, s.height - 1);
        drawLabel(g);
    }
    private void drawLabel(Graphics g) {
        FontMetrics fm = g.getFontMetrics();
        int width = fm.stringWidth(label);
        int height = fm.getHeight();
        int ascent = fm.getAscent();
        int leading = fm.getLeading();
        int horizMargin =
            (getSize().width - width)/2;
        int verMargin =
            (getSize().height - height)/2;
        g.setColor(Color.red);
        g.drawString(label, horizMargin,
            verMargin + ascent + leading);
    }
    class AL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            clickCounter++;
            label = "click #" + clickCounter +
                " " + e.toString();
            repaint();
        }
    }
}

```

```

public class GoodTechnique extends Frame {
    GoodTechnique() {
        setLayout(new GridLayout(2,2));
        add(new EnabledPanel(1, Color.cyan));
        add(new EnabledPanel(2, Color.lightGray));
        add(new EnabledPanel(3, Color.yellow));
    }
    public static void main(String[] args) {
        Frame f = new GoodTechnique();
        f.setTitle("Good Technique");
        f.addWindowListener(
            new WindowAdapter() {
                public void windowClosing(WindowEvent e){
                    System.out.println(e);
                    System.out.println("Window Closing");
                    System.exit(0);
                }
            });
        f.setSize(700,700);
        f.setVisible(true);
    }
} //:~

```

Този пример също демонстрира различните възникващи събития и извежда информация за тях. Класът **Display** е начин за централизиране на извеждането на информацията. Има масив от **String**ове да държи информация за всеки тип събитие и методът **show()** взема манипулятор към какъвто **Graphic**чен обект имате и пише директно на съответната повърхност. Схемата е направена с намерение да бъде повторно използваема по някакъв начин.

EnabledPanel представя новия тип компонент. Това е оцветен панел с бутона в долния край и той хваща всички събития произлезли над него чрез използване на вътрешни слушателски класове за всяко отделно събитие освен онези в които **EnabledPanel** поддържа **processEvent()** в стария стил (забележете че трябва също да извика **super.processEvent()**). Единствената причина да се използва този метод е че хваща всяко станало събитие, така че може да видите всичко, което се случва. **processEvent()** не прави нищо повече от показването на стринговото представяне на всяко събитие, иначе щеше да има нужда от каскадни **if** оператори за да се установи типа на събитието. От друга страна, вътрешните слушателски класове вече знаят точно какво се е случило. (Предполагайки че ги регистрирате с компонент където не е необходима никакава управляваща логика, което и трябва да е целта ви.) Така те не трябва нищо да проверяват; просто си вършат тяхното.

Всеки слушател променя **Display** стринга асоцииран с конкретното събитие и вика **repaint()** да се изобразят стринговете. Може също да видите един трик който обикновено премахва трепкането:

```

public void update(Graphics g) {
    paint(g);
}

```

Не е необходимо винаги да поддържате **update()**, но ако напишете нещо което трябва, опитайте. Версията по подразбиране чисти фона и после вика **paint()** да прерисува гряфиките. Това чистене обикновено предизвиква трепкането но не непременно, понеже **paint()** прерисува цялата площ.

Може да видите че има много слушатели – обаче проверка на типа се провежда за слушателите, та не може да слушате за нещо, което компонентът не поддържа (за разлика от **BadTechnique.java**, което моментално ще видите).

Експериментирането с тази програма е много поучително понеже се вижда как се работи със събитията в Java. Първо, показва протичането на проектирането в повечето системи с прозорци: твърде трудно е да се натисне и отпусне бутон на мишката без тя да се мръдне, та системата ще мисли че влажите наместо това което искате. Решението е да използвате **mousePressed()** и **mouseReleased()** вместо **mouseClicked()**, а после да решите дали да викате ваш собствен "mouseReallyClicked()" метод основан на време и около 4 пиксела хистерезис на мишката.

Грозен интерфейс на компонент

Алтернативата, която ще видите в много публикации, е да се извика **enableEvents()** и да се даде маска която показва кои събития искате да обработите. Това довежда до даването на тези събития на методи по стария стил (макар че са нови в Java 1.1) с имена като **processFocusEvent()**. Трябва също да помните да викате версията на базовия клас. Ето как изглежда това:

```
//: c13:BadTechnique.java
// It's possible to override components this way,
// but the listener approach is much better, so
// why would you?
import java.awt.*;
import java.awt.event.*;

class Display {
    public static final int
        EVENT = 0, COMPONENT = 1,
        MOUSE = 2, MOUSE_MOVE = 3,
        FOCUS = 4, KEY = 5, ACTION = 6,
        LAST = 7;
    public String[] evnt;
    Display() {
        evnt = new String[LAST];
        for(int i = 0; i < LAST; i++)
            evnt[i] = new String();
    }
    public void show(Graphics g) {
        for(int i = 0; i < LAST; i++)
            g.drawString(evnt[i], 0, 10 * i + 10);
    }
}

class EnabledPanel extends Panel {
    Color c;
    int id;
    Display display = new Display();
    public EnabledPanel(int i, Color mc) {
        id = i;
        c = mc;
        setLayout(new BorderLayout());
        add(new MyButton(), BorderLayout.SOUTH);
        // Type checking is lost. You can enable and
        // process events that the component doesn't
        // capture:
        enableEvents(
            // Panel doesn't handle these:
            AWTEvent.ACTION_EVENT_MASK |
```

```

AWTEvent.ADJUSTMENT_EVENT_MASK |
AWTEvent.ITEM_EVENT_MASK |
AWTEvent.TEXT_EVENT_MASK |
AWTEvent.WINDOW_EVENT_MASK |
// Panel can handle these:
AWTEvent.COMPONENT_EVENT_MASK |
AWTEvent.FOCUS_EVENT_MASK |
AWTEvent.KEY_EVENT_MASK |
AWTEvent.MOUSE_EVENT_MASK |
AWTEvent.MOUSE_MOTION_EVENT_MASK |
AWTEvent.CONTAINER_EVENT_MASK);
// You can enable an event without
// overriding its process method.
}
// To eliminate flicker:
public void update(Graphics g) {
    paint(g);
}
public void paint(Graphics g) {
    g.setColor(c);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
    g.setColor(Color.black);
    display.show(g);
}
public void processEvent(AWTEvent e) {
    display.evnt(Display.EVENT)= e.toString();
    repaint();
    super.processEvent(e);
}
public void
processComponentEvent(ComponentEvent e) {
    switch(e.getID()) {
        case ComponentEvent.COMPONENT_MOVED:
            display.evnt(Display.COMPONENT) =
                "Component moved";
            break;
        case ComponentEvent.COMPONENT_RESIZED:
            display.evnt(Display.COMPONENT) =
                "Component resized";
            break;
        case ComponentEvent.COMPONENT_HIDDEN:
            display.evnt(Display.COMPONENT) =
                "Component hidden";
            break;
        case ComponentEvent.COMPONENT_SHOWN:
            display.evnt(Display.COMPONENT) =
                "Component shown";
            break;
        default:
    }
    repaint();
    // Must always remember to call the "super"
    // version of whatever you override:
    super.processComponentEvent(e);
}
public void processFocusEvent(FocusEvent e) {

```

```

switch(e.getID()) {
    case FocusEvent.FOCUS_GAINED:
        display.evnt(Display.FOCUS) =
            "FOCUS gained";
        break;
    case FocusEvent.FOCUS_LOST:
        display.evnt(Display.FOCUS) =
            "FOCUS lost";
        break;
    default:
}
repaint();
super.processFocusEvent(e);
}

public void processKeyEvent(KeyEvent e) {
    switch(e.getID()) {
        case KeyEvent.KEY_PRESSED:
            display.evnt(Display.KEY) =
                "KEY pressed: ";
            break;
        case KeyEvent.KEY_RELEASED:
            display.evnt(Display.KEY) =
                "KEY released: ";
            break;
        case KeyEvent.KEY_TYPED:
            display.evnt(Display.KEY) =
                "KEY typed: ";
            break;
        default:
    }
    int code = e.getKeyCode();
    display.evnt(Display.KEY) +=
        KeyEvent.getKeyText(code);
    repaint();
    super.processKeyEvent(e);
}

public void processMouseEvent(MouseEvent e) {
    switch(e.getID()) {
        case MouseEvent.MOUSE_CLICKED:
            requestFocus(); // Get FOCUS on click
            display.evnt(Display.MOUSE) =
                "MOUSE clicked";
            break;
        case MouseEvent.MOUSE_PRESSED:
            display.evnt(Display.MOUSE) =
                "MOUSE pressed";
            break;
        case MouseEvent.MOUSE_RELEASED:
            display.evnt(Display.MOUSE) =
                "MOUSE released";
            break;
        case MouseEvent.MOUSE_ENTERED:
            display.evnt(Display.MOUSE) =
                "MOUSE entered";
            break;
        case MouseEvent.MOUSE_EXITED:
            display.evnt(Display.MOUSE) =

```

```

        "MOUSE exited";
        break;
    default:
    }
    display.evnt(Display.MOUSE) +=
        ", x = " + e.getX() +
        ", y = " + e.getY();
    repaint();
    super.processMouseEvent(e);
}
public void
processMouseMotionEvent(MouseEvent e) {
    switch(e.getID()) {
        case MouseEvent.MOUSE_DRAGGED:
            display.evnt(Display.MOUSE_MOVE) =
                "MOUSE dragged";
            break;
        case MouseEvent.MOUSE_MOVED:
            display.evnt(Display.MOUSE_MOVE) =
                "MOUSE moved";
            break;
        default:
    }
    display.evnt(Display.MOUSE_MOVE) +=
        ", x = " + e.getX() +
        ", y = " + e.getY();
    repaint();
    super.processMouseMotionEvent(e);
}
}

class MyButton extends Button {
    int clickCounter;
    String label = "";
    public MyButton() {
        enableEvents(AWTEvent.ACTION_EVENT_MASK);
    }
    public void paint(Graphics g) {
        g.setColor(Color.green);
        Dimension s = getSize();
        g.fillRect(0, 0, s.width, s.height);
        g.setColor(Color.black);
        g.drawRect(0, 0, s.width - 1, s.height - 1);
        drawLabel(g);
    }
    private void drawLabel(Graphics g) {
        FontMetrics fm = g.getFontMetrics();
        int width = fm.stringWidth(label);
        int height = fm.getHeight();
        int ascent = fm.getAscent();
        int leading = fm.getLeading();
        int horizMargin =
            (getSize().width - width)/2;
        int verMargin =
            (getSize().height - height)/2;
        g.setColor(Color.red);
        g.drawString(label, horizMargin,

```

```

        verMargin + ascent + leading);
    }

    public void processActionEvent(ActionEvent e) {
        clickCounter++;
        label = "click #" + clickCounter +
            " " + e.toString();
        repaint();
        super.processActionEvent(e);
    }
}

public class BadTechnique extends Frame {
    BadTechnique() {
        setLayout(new GridLayout(2,2));
        add(new EnabledPanel(1, Color.cyan));
        add(new EnabledPanel(2, Color.lightGray));
        add(new EnabledPanel(3, Color.yellow));
        // You can also do it for Windows:
        enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    }

    public void processWindowEvent(WindowEvent e) {
        System.out.println(e);
        if(e.getID() == WindowEvent.WINDOW_CLOSING) {
            System.out.println("Window Closing");
            System.exit(0);
        }
    }

    public static void main(String[] args) {
        Frame f = new BadTechnique();
        f.setTitle("Bad Technique");
        f.setSize(700,700);
        f.setVisible(true);
    }
} //:~

```

За работата работи. Но е грозновато, трудно за писане, четене, дебъгинг, поддръжка и повторно използване. Така че защо да се тормозим когато можем да използваме вътрешни класове за слушатели?

JFC APIта

JFC включват важна функционалност, включително за фокуса, достъпа до цветовете, печатане "вътре в кутията," началото на поддръжка на клипборд и влечене-пускане.

Работата с фокуса е доста лесна, понеже тя е представена прозарбно за програмиста в компонентите от Swing библиотеката и не е необходимо да правите каквото и да било, за да работи. Ако правите собствени компоненти и искате да работят с фокуса, подтискате **isFocusTraversable()** да връща **true**. Ако искате да хванете фокуса на клавиатурата с кликване на мишката, хващате събитието на мишката и викате **requestFocus()**.

Цветовете

Дадена е възможност да се разбере какви от разнообразните възможности с цветовете са избрани. По този начин може да използвате точно тези цветове, ако искате. Цветовете автоматично се инициализират и слагат в **static** членове на **SystemColor**, така че всичко което трябва да направите е да прочетете интересуващия ви член. Имената са международно

самоговорещи: `desktop`, `activeCaption`, `activeCaptionText`, `activeCaptionBorder`, `inactiveCaption`, `inactiveCaptionText`, `inactiveCaptionBorder`, `window`, `windowBorder`, `windowText`, `menu`, `menuText`, `text`, `textText`, `textHighlight`, `textHighlightText`, `textInactiveText`, `control`, `controlText`, `controlHighlight`, `controlLtHighlight`, `controlShadow`, `controlDkShadow`, `scrollbar`, `info` (за помощ) и `infoText` (за текста - помощ).

Печатане

За нещастие няма много автоматизация за печатането. Вместо това трябва да извървите механични, не-ОО стъпки за да може да печатате. Печатането на компонент графично може да бъде малко по-автоматизирано: по подразбиране методат `print()` вика метода `paint()` за да си свърши работата. Има случаи когато това е удовлетворително, но ако искате да правите нещо по-специализирано трябва да знаете какво и как ще се печата в частност за да може да намерите размерите на хартията.

Следният пример демонстрира печатане на текст и графика и различните подходи които може да използвате за печатането на графика. Освен това се тества поддръжката за печатането:

```
//: c13:PrintDemo.java
// Printing with the JFC
// Note: This example compiles, but doesn't run
// correctly.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class PrintDemo extends JFrame {
    JButton
        text = new JButton("Print Text"),
        graphics = new JButton("Print Graphics");
    JTextField ringNum = new JTextField(3);
    JComboBox faces = new JComboBox();
    Graphics g = null;
    Plot plot = new Plot3D(); //Try different plots
    Toolkit tk = Toolkit.getDefaultToolkit();
    public PrintDemo() {
        ringNum.setText("3");
        ringNum.addActionListener(new RingL());
        JPanel p = new JPanel();
        p.setLayout(new FlowLayout());
        text.addActionListener(new TBL());
        p.add(text);
        p.add(new Label("Font:"));
        p.add(faces);
        graphics.addActionListener(new GBL());
        p.add(graphics);
        p.add(new Label("Rings:"));
        p.add(ringNum);
        Container cp = getContentPane();
        cp.setLayout(new BorderLayout());
        cp.add(p, BorderLayout.NORTH);
        cp.add(plot, BorderLayout.CENTER);
        // Toolkit.getFontList() is deprecated
        GraphicsEnvironment ge=GraphicsEnvironment.
            getLocalGraphicsEnvironment();
```

```

String[] fontList = ge.
    getAvailableFontFamilyNames();
for(int i = 0; i < fontList.length; i++)
    faces.addItem(fontList(i));
faces.setSelectedItem("Serif");
}
class PrintData {
    public PrintJob pj;
    public int pageWidth, pageHeight;
    PrintData(String jobName) {
        pj = getToolkit().getPrintJob(
            PrintDemo.this, jobName, null);
        if(pj != null) {
            pageWidth=pj.getPageDimension().width;
            pageHeight=pj.getPageDimension().height;
            g = pj.getGraphics();
        }
    }
    void end() { pj.end(); }
}
class ChangeFont {
    private int stringHeight;
    ChangeFont(String face, int styl, int pnt){
        if (g != null) {
            g.setFont(new Font(face, styl, pnt));
            stringHeight =
                g.getFontMetrics().getHeight();
        }
    }
    int stringWidth(String s) {
        return g.getFontMetrics().stringWidth(s);
    }
    int stringHeight() { return stringHeight; }
}
class TBL implements ActionListener {
    public void actionPerformed(ActionEvent e){
        PrintData pd =
            new PrintData("Print Text Test");
        // Null means print job canceled
        if(pd == null) return;
        String s = "PrintDemo";
        ChangeFont cf = new ChangeFont(
            faces.getSelectedItem().toString(),
            Font.ITALIC,72);
        g.drawString(s,
            (pd.pageWidth - cf.stringWidth(s))/2,
            (pd.pageHeight - cf.stringHeight())/3);

        s = "A smaller print size";
        cf = new ChangeFont(
            faces.getSelectedItem().toString(),
            Font.BOLD,48);
        g.drawString(s,
            (pd.pageWidth - cf.stringWidth(s))/2,
            (int)((pd.pageHeight -
                cf.stringHeight())/1.5));
        g.dispose();
    }
}

```

```

        pd.end();
    }
}

class GBL implements ActionListener {
    public void actionPerformed(ActionEvent e){
        PrintData pd =
            new PrintData("Print Graphics Test");
        if(pd == null) return;
        plot.print(g);
        g.dispose();
        pd.end();
    }
}

class RingL implements CaretListener {
    public void caretUpdate(CaretEvent e) {
        int i = 1;
        try {
            i = Integer.parseInt(ringNum.getText());
        } catch(NumberFormatException ex) {
            i = 1;
        }
        plot.rings = i;
        plot.repaint();
    }
}

public static void main(String[] args) {
    JFrame pdemo = new PrintDemo();
    pdemo.setTitle("Print Demo");
    pdemo.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    pdemo.setSize(600,600);
    pdemo.setVisible(true);
}
}

class Plot extends JPanel {
    public int rings = 3;
}

class Plot1 extends Plot {
    // Default print() calls paint();
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int w = getSize().width;
        int h = getSize().height;
        int xc = w/2;
        int yc = w/2;
        int x = 0,y = 0;
        for (int i = 0; i < rings; i++) {
            if (x<xc && y<yc) {
                g.drawOval(x, y, w, h);
                x += 10; y += 10;
                w -= 20; h -= 20;
            }
        }
    }
}

```

```

        }
    }
}

class Plot2 extends Plot {
    // To fit the picture to the page, you must
    // know whether you're printing or painting:
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int w, h;
        if (g instanceof PrintGraphics) {
            PrintJob pj =
                ((PrintGraphics)g).getPrintJob();
            w = pj.getPageDimension().width;
            h = pj.getPageDimension().height;
        }
        else {
            w = getSize().width;
            h = getSize().height;
        }
        int xc = w / 2;
        int yc = w / 2;
        int x = 0, y = 0;
        for (int i = 0; i < rings; i++) {
            if (x < xc && y < yc) {
                g.drawOval(x,y,w,h);
                x += 10; y += 10;
                w -= 20; h -= 20;
            }
        }
    }
}

```

```

class Plot3 extends Plot {
    // Somewhat better. Separate
    // printing from painting:
    public void print(Graphics g) {
        // Assume it's a PrintGraphics object:
        PrintJob pj=((PrintGraphics)g).getPrintJob();
        int w = pj.getPageDimension().width;
        int h = pj.getPageDimension().height;
        doGraphics(g, w, h);
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int w = getSize().width;
        int h = getSize().height;
        doGraphics(g, w, h);
    }
    private void
    doGraphics(Graphics g, int w,int h) {
        int xc = w/2;
        int yc = w/2;
        int x = 0,y = 0;
        for (int i = 0; i < rings; i++) {
            if (x < xc && y < yc) {

```

```

        g.drawOval(x, y, w, h);
        x += 10; y += 10;
        w -= 20; h -= 20;
    }
}
}
} //:~

```

Програмата позволява да се избират шрифтове от **Choice** списък (и ще видите че броят на шрифтовете достъпни в Java 1.1 е още крайно ограничен, а нищо не може да направите с евентуални допълнителни шрифтове инсталиирани на вашата машина). Използва ги за печатане на текст с надебеляване на буквите, курсив и различни големини. Освен това се създава нов тип компонент наречен **Plot** за демонстрация на графиките. **Plot** има пръстенчека които ще изведе на дисплея и ще отпечата на хартия и три извлечени класа **Plot1**, **Plot2** и **Plot3** правят това по различни начини да видите алтернативите когато печатате графики. Също може да промените броя на пръстенчетата на чертеж – това е интересно понеже показва деликатността на печатането в Java 1.1. На моята система принтерът даваше съобщения за грешки и не печаташе когато колелцата станеха “твърде много” (каквото и да значи това), но работеше чудесно когато броят беше “достатъчно малък.” Ще забележите също че размерите на полето при печатане не съвпадат с размерите зададени за листа хартия. Това може би ще се оправи в бъдеща версия на Java и може да използвате тази програма за проверка на това.

Тази програма капсулира функционалността във вътрешни класове навсякъде където е възможно, заради повторното използване на кода. Например винаги когато искате да започнете печатане (без значение графика или текст), трябва да създадете обект **PrintJob**, който има собствен **Graphics** обект заедно с ширина и дължина на листа. Създаването на **PrintJob** и извличането на размерите на листа са капсулирани в класа **PrintData**.

Печатане на текст

Концептуално печатането на текст е праволинейно действие: избирате шрифт и големина на буквите, решавате каде стрингът ще се появи на страницата и го рисувате с **Graphics.drawString()**. Това значи, обаче, че трябва вие да изпълните изчисленията къде точно трабва да отиде всеки ред за да осигурите че няма да излезе от страницата и няма да попадне върху други редове. Ако искате да правите програма за обработка на текстове, имате достатъчно работа.

ChangeFont капсулира малка част от процеса на преминаване от един шрифт към друг чрез автоматично създаване на нов **Font** обект с желания шрифт, стил (**Font.BOLD** или **Font.ITALIC** – няма поддръжка за подчертаване, зачертаване и т.н.), и големина на точката. Също опростява изчислението на ширината и височината на стринга.

Когато натиснете бутона “Print text”, активира се слушателят **TBL**. Може да видите че той преминава през две итерации на създаване на **ChangeFont** и викане на **drawString()** за изпечатване на стринга в изчисленото място, центриран, една трета и две трети надолу от началото на страницата, респективно. Забележете дали тези операции дават желаните резултати. (Те не ги даваха при използване на първа версия.)

Печатане на графика

Когато натиснете бутона “Print graphics” се активира слушателят **GBL**. Създаване на обект **PrintData** инициализира **g**, а после може просто да се вика **print()** за компонента който искате да печатате. За да се причини печатане трябва да се извика **dispose()** за **Graphics** обект и **end()** за обекта **PrintData** (който се завърта и вика **end()** за **PrintJob**).

Работата продължава вътре в **Plot** обекта. Може да се види че базовия клас **Plot** е простичик – той разширява **Canvas** и съдържа **int** наречен **rings** за индикация колко концентрични кръга да

изрисува на тази конкретна **Canvas**. Трите извлечени класа показват три начина за достигане на целта: чертане на екрана и печатане на хартия.

Plot1 използва най-простиия подход: игнорира факта че има разлики между оцветяването (на екран-б.пр.) и печатането, просто се подтиска **paint()**. Причината това да работи е че **print()** методът просто се оглежда и вика **paint()**. Обаче ще видите че дължината на идхода е зависима от размерите на текстурата на екрана, така че има смисъл **width** и **height** да се определят чрез викане на **Canvas.getSize()**. Друга ситуация където това е приемливо е когато образът винаги е с едни и същи размери.

Когато дължината на повърхността върху която се чертае е важна се налага да се открият размерите. За нещастие това излиза тромава работа, както се вижда с **Plot2**. Поради някаква може би добра причина която аз не знам не може просто да се пита **Graphics** обекта за размерите на графичната му повърхност. Това щеше да направи процеса доста елегантен. Вместо това да видите дали печатате, а не рисувате, трябва да детектирате **PrintGraphics** чрез ключовата дума на RTTI **instanceof** (описана в глава 11), после даункаст и викате единствения **PrintGraphics** метод: **getPrintJob()**. Сега имате манипулятор към **PrintJob** и можете да намерите ширината и дължината на хартията. Това е груб подход, но сигурно има сериозна причина за него. (От друга страна вече сте виждали примери на библиотеки които създават впечатление че дизайнърите само са кашляли (или крали идеи, код и пр. - игра на думи, бел.пр.) наоколо...)

Може да се види че **paint()** в **Plot2** минава през двете възможности за рисуване и печатане. Но тъй като **print()** методът трябва да се вика когато се печата, защо да не се използва? Този подход се използва в **Plot3**, като елиминира нуждата да се използва **instanceof** понеже вътре в **print()** може да се приеме, че може да се каства към **PrintGraphics** обект. Така е малко по-добре. Ситуацията е подобрена чрез слагане на всички рисуващ код (щом веднъж размерите са намерени) вътре в отделен метод **doGraphics()**.

Стартиране на Frameове в аплети

Ами ако искате да печатате от аплет? За да се печата каквото и да е трябва да се вземе **PrintJob** обект чрез **getPrintJob()** метода на **Toolkit** обект, който взима само **Frame** обект, не и **Applet**. Така изглежда възможно да се печата от приложение, но невъзможно от аплет. Оказва се обаче че може да създадете **Frame** в аплет (което е обратното на това което правех за аплет/приложение примерите до тук, което беше да се направи аплет и да се сложи вътре във **Frame**). Това е полезна техника понеже позволява да се използват много приложения в аплети (доколкото не нарушават сигурността на аплетите). Когато прозорецът на приложението има аплет, обаче, ще забележите че Web броузърът мърмори, нещо като "Warning: Applet Window."

Може да видите че е доста праволинейно да сложите **Frame** в аплет. Единственото нещо което трябва да добавите е код за **dispose()** на **Frame** за когато потребителят го затваря (вместо викане на **System.exit()**):

```
//: c13:PrintDemoApplet.java
// Creating a Frame from within an Applet
// <applet code=PrintDemoApplet
// width=125 height=50> </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class PrintDemoApplet extends JApplet {
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
```

```

JButton b = new JButton("Run PrintDemo");
b.addActionListener(new PDL());
cp.add(b);
}
class PDL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        final PrintDemo pd = new PrintDemo();
        pd.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                pd.dispose();
            }
        });
        pd.setSize(600,600);
        pd.setVisible(true);
    }
}
} //:~

```

Поддръжката на печатането в Java 1.1 води до известни конфузии. Някои публикации каточели твърдят, че може да се печата от аплет. Обаче системата за сигурност на Java има тази черта, че ще предотврати правенето от аплета на собствена принтерска задача, изисквайки това да бъде направено чрез Web браузъра или това, което показва (работи с) аплета. По време на написването на тези редове това изглежда оставаше нерешен въпрос. Когато пуснах тази програма от Web браузър, **PrintDemo** прозорецът си излезе много добре, но не можа да печата от браузъра.

Джобът

JFC поддържа ограничена функционалност със системната памет (където може временно да се слагат данни, обекти и т.н - б.пр.). (в пакета **java.awt.datatransfer**). Може да се копират **String** обекти в "джоба" като текст, може и да се вмъква текст от джоба в **String** обекти. Разбира се, джобът е проектиран да работи с всякакъв вид данни, но как данните са представени е работа на програмата, чрез която става обменът. Макар и в момента да поддържа само стрингови данни, API на Java за джоба дава разширяемост чрез концепцията за "вкус, дъх." Когато данните напускат джоба, с тях се асоциира множество от вкусове, каквито те биха могли да бъдат (например един граф би могъл да се представи с колона числа или с образа си) и би могло да се провери дали там, където ще ходят данните въпросният вкус се поддържа.

Следната програма е проста илюстрация на слагане в джоба, копиране в джоба и изваждане от джоба на **String** данни в **TextArea**. Едното нещо което ще забележите е, че комбинациите от клавиши, които обикновено използвате за споменатите операции работят. Но ако погледнете към кой да е **TextField** или **TextArea** в коя да е друга програма ще видите, че и там се поддържат. Тази програма просто добавя програмно еправление на джоба, може да използвате тези техники за хващане на не-**TextComponent** в джоба.

```

//: c13:CutAndPaste.java
// Using the clipboard
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.datatransfer.*;

public class CutAndPaste extends JFrame {
    JMenuBar mb = new JMenuBar();
    JMenu edit = new JMenu("Edit");
    JMenuItem

```

```

cut = new JMenuItem("Cut"),
copy = new JMenuItem("Copy"),
paste = new JMenuItem("Paste");
JTextArea text = new JTextArea(20, 20);
Clipboard clipbd =
getToolkit().getSystemClipboard();
public CutAndPaste() {
cut.addActionListener(new CutL());
copy.addActionListener(new CopyL());
paste.addActionListener(new PasteL());
edit.add(cut);
edit.add(copy);
edit.add(paste);
mb.add(edit);
setJMenuBar(mb);
getContentPane().add(text);
}
class CopyL implements ActionListener {
public void actionPerformed(ActionEvent e) {
String selection = text.getSelectedText();
if (selection == null)
return;
StringSelection clipString =
new StringSelection(selection);
clipbd.setContents(clipString, clipString);
}
}
class CutL implements ActionListener {
public void actionPerformed(ActionEvent e) {
String selection = text.getSelectedText();
if (selection == null)
return;
StringSelection clipString =
new StringSelection(selection);
clipbd.setContents(clipString, clipString);
text.replaceRange("", text.getSelectionStart(),
text.getSelectionEnd());
}
}
class PasteL implements ActionListener {
public void actionPerformed(ActionEvent e) {
Transferable clipData =
clipbd.getContents(CutAndPaste.this);
try {
String clipString =
(String)clipData.
getTransferData(
DataFlavor.stringFlavor);
text.replaceRange(clipString,
text.getSelectionStart(),
text.getSelectionEnd());
} catch(Exception evt) {
System.out.println("not String flavor");
}
}
}

```

```

public static void main(String[] args) {
    JFrame frame = new CutAndPaste();
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    frame.setSize(300, 200);
    frame.setVisible(true);
}
} //:~

```

Създаването и прибавянето на менюто и **TextArea** трява вече да е банална дейност. Различното е създаването на полето **clipbd** на **Clipboard** което е направено чрез **Toolkit**.

Всичката работи се върши в слушателите. **CopyL** и **CutL** слушателите са еднакви с изключение на последния ред на **CutL**, който изтрива реда, който се копира. Специалните два реда са създаването на **StringSelection** обект от **String** и викането на **setContents()** с този **StringSelection**. Това е необходимото за слагане на **String** в джоба.

В **PasteL** данните се вадят от джоба чрез **getContents()**. Върнатото е съвсем анонимен **Transferable** обект, даже фактически не се знае съдържанието му. Един начин да се намери е **getTransferDataFlavors()**, който връща масив от **DataFlavor** обект индициращ какви вкусове се поддържат от този конкретен обект. Може също да проверите директно с **isDataFlavorSupported()**, подавайки вкуса който ви интересува. Тук обаче е използван дързък подход: **getTransferData()** се вика като се предполага, че се поддържа вкуса **String** и ако не е така, проблемът се отсортира от обработчика на изключения.

В бъдеще може да се очаква поддържането на повече видове данни.

Теглене и Пускане

В следния пример⁸, когато изпълнявате класа **DragDemo**, ще се появят две рамки. Това ще позволи да се демонстрира ТП с една JVM. Може да пуснете втори екземпляр на класа за да видите ТП с две JVM-ини.

```

//: c13:TextData.java
import java.awt.datatransfer.*;
import java.io.*;
import javax.swing.*;
import java.util.*;

public class TextData implements Transferable {
    private String text;
    private DataFlavor[] flavors = {
        DataFlavor.stringFlavor,
        DataFlavor.plainTextFlavor };
    private java.util.List flavorList =
        Arrays.asList( flavors );
    private static void print(String s) {
        System.out.println(s);
    }
    public TextData(String txt) { text = txt; }
    public DataFlavor[] getTransferDataFlavors() {
        return flavors;
    }
}

```

⁸ The original version of this example was contributed by Rahim Adatia.

```

}
public boolean
isDataFlavorSupported(DataFlavor flavor) {
    print("isDataFlavorSupported");
    return flavorList.contains(flavor);
}
public Object
getTransferData(DataFlavor flavor) throws
    UnsupportedFlavorException, IOException {
    if(flavor.equals(DataFlavor.stringFlavor)) {
        print("stringflavor request");
        return text;
    }
    if(flavor.equals(DataFlavor.plainTextFlavor)){
        print("plainTextFlavor request");
        return new ByteArrayInputStream(
            text.getBytes("Unicode"));
    }
    print("returning null");
    return null;
}
} ///:~

```

```

//: c13:Draggable.java
import java.awt.*;
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
import javax.swing.text.*;

public class Draggable extends JLabel
implements DragSourceListener {
    private DragSource source;
    private TextData transferable;
    private int dragAction;
    private static void print(String s) {
        System.out.println(s);
    }
    public Draggable(String text, int action) {
        super(text);
        dragAction = action;
        source = new DragSource();
        DGListener listener = new DGListener();
        transferable = new TextData(text);
        try {
            getToolkit().createDragGestureRecognizer(
                Class.forName("java.awt.dnd." +
                    "MouseDragGestureRecognizer"),
                source, this, dragAction, listener);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}

```

```

public void
dropActionChanged(DragSourceDragEvent dsde) {
    print("Source:dragActionChanged");
}
public void
dragEnter(DragSourceDragEvent dsde) {
    print("Source:dragEnter");
}
public void dragOver(DragSourceDragEvent dsde){
    print("Source:dragOver");
}
public void
dragGestureChanged(DragSourceDragEvent dsde) {
    print("Source:dragGestureChanged");
}
public void dragExit(DragSourceEvent dse) {
    print("Source:dragExit");
}
public void
dragDropEnd(DragSourceDropEvent dsde) {
    print("Source:dragDropEnd");
    JPanel parent = (JPanel)getParent();
    if(dsde.getDropSuccess() == true) {
        print("The drop was a success");
        parent.remove(this);
    }
    parent.revalidate();
}
public class DGListener
implements DragGestureListener {
    public void
    dragGestureRecognized(DragGestureEvent e) {
        print("DGListener:Starting to Drag");
        source.startDrag(e,
            DragSource.DefaultCopyDrop,
            transferable, Draggable.this);
    }
}
} ///:~

```

```

//: c13:DropPanel.java
import javax.swing.*;
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import javax.swing.border.*;
import java.io.*;
import java.util.*;

public class DropPanel extends JPanel
    implements DropTargetListener {
    DropTarget target;
    private static void print(String s) {
        System.out.println(s);
    }
    public DropPanel() {

```

```

super();
setSize(300,150);
setBorder(
    new BevelBorder(BevelBorder.LOWERED));
target = new DropTarget(this,this);
target.setActive(true);
}
public void dragEnter(DropTargetDragEvent e) {
    print("Target:dragEnter");
    int sourceAction = e.getSourceActions();
    int dropAction = e.getDropAction();
    if(sourceAction ==
        DnDConstants.ACTION_COPY_OR_MOVE
    || sourceAction ==
        DnDConstants.ACTION_MOVE) {
        List flavorList =
            Arrays.asList(e.getCurrentDataFlavors());
        if(flavorList.contains(
            DataFlavor.stringFlavor)
        || flavorList.contains(
            DataFlavor.plainTextFlavor) ) {
            e.acceptDrag(dropAction);
            print("Target:Accepted in dragEnter");
            return;
        }
    }
    print("Target: Rejecting drag");
    e.rejectDrag();
}
public void drop(DropTargetDropEvent e) {
    print("Target: in drop");
    int sourceAction = e.getSourceActions();
    if(sourceAction ==
        DnDConstants.ACTION_MOVE
    || sourceAction ==
        DnDConstants.ACTION_COPY_OR_MOVE) {
        e.acceptDrop(e.getDropAction());
        print("Target:Accepted in drop");
        Transferable trans = e.getTransferable();
        Object obj = null;
        if(e.isLocalTransfer()) {
            if(e.isDataFlavorSupported(
                DataFlavor.stringFlavor)) {
                try {
                    obj = trans.getTransferData(
                        DataFlavor.stringFlavor);
                } catch(Exception ex) {
                    ex.printStackTrace();
                }
            }
        } else if(e.isDataFlavorSupported(
            DataFlavor.plainTextFlavor)) {
            try {
                obj = trans.getTransferData(
                    DataFlavor.plainTextFlavor);
            } catch(Exception ex) {
                ex.printStackTrace();
            }
        }
    }
}

```

```

        }
    }
    if(obj == null) {
        e.dropComplete(false);
        return;
    }
    print("Got the obj in drop");
    if(obj instanceof String) {
        try {
            String input = (String)obj;
            add(new Draggable(
                input,DnDConstants.ACTION_MOVE));
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    } else if(obj instanceof InputStream) {
        InputStream in = (InputStream)obj;
        InputStreamReader isr = null;
        try {
            isr=new InputStreamReader(in,"Unicode");
        } catch(UnsupportedEncodingException ex){
            ex.printStackTrace();
            return;
        }
        StringBuffer str = new StringBuffer();
        int bytesRead=-1;
        try {
            while((bytesRead = isr.read()) >= 0 ) {
                if(bytesRead != 0)
                    str.append((char)bytesRead);
            }
            add(new Draggable(str.toString(),
                DnDConstants.ACTION_MOVE));
        } catch(IOException ex) {
            ex.printStackTrace();
            e.dropComplete(false);
            return;
        }
    } else {
        print("Target: Drop Rejected");
        e.dropComplete(false);
        return;
    }
    e.dropComplete(true);
    revalidate();
}
}

public void dragOver(DropTargetDragEvent t) {
    print("Target:dragOver");
}
public void dragScroll(DropTargetDragEvent t) {
    print("Target:dragScroll");
}
public void dragExit(DropTargetEvent dte) {
    print("Target:dragExit");
}
public void

```

```
dropActionChanged(DropTargetDragEvent t){
    print("Target:dragActionChanged");
}
} //:/~
```

```
//: c13:DragDemo.java
import java.awt.*;
import java.awt.dnd.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class DragDemo {
    Draggable label1;
    Draggable label2;
    Draggable label3;
    public DragDemo() {
        DragFrame dragFrame =
            new DragFrame("Drag Frame");
        dragFrame.setSize(200, 150);
        dragFrame.setLocation(200,200);
        dragFrame.setVisible(true);
        dragFrame.pack();
        DropFrame dropFrame =
            new DropFrame("Drop Frame");
        dropFrame.setSize(200,150);
        dropFrame.setLocation(300,300);
        dropFrame.setVisible(true);
    }
    public static void main(String args) {
        DragDemo dragDemo = new DragDemo();
    }
    public class DragFrame extends JFrame {
        public DragFrame(String title) {
            super(title);
            Container cp = getContentPane();
            addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e){
                    System.exit(0);
                }
            });
            label1 = new Draggable("ACTION_COPY",
                DnDConstants.ACTION_COPY);
            label2 = new Draggable("ACTION_MOVE",
                DnDConstants.ACTION_MOVE);
            label3 = new Draggable(
                "ACTION_COPY_OR_MOVE",
                DnDConstants.ACTION_COPY_OR_MOVE);
            DropPanel dragPanel = new DropPanel();
            dragPanel.setLayout(new GridLayout(3,1));
            dragPanel.setSize(new Dimension(300,150));
            dragPanel.setBorder(
                new BevelBorder(BevelBorder.LOWERED));
            dragPanel.add(label1);
            dragPanel.add(label2);
```

```

        dragPanel.add(label3);
        cp.add(dragPanel);
    }
}

public class DropFrame extends JFrame {
    public DropFrame(String title) {
        super(title);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
        getContentPane().add(new DropPanel());
    }
}
} //:~

```

Визуално програмиране и Bean-ове

До тук в тази книга вече сте видели колко е ценен Java за създаване на многократно използваеми фрагменти код. „Най-многократно използваемата“ единица код бе класът, понеже съдържа свързани помежду си данни (полета) и поведение (методи) които могат да бъдат използвани или директно чрез композиция или чрез наследяване.

Наследяването и полиморфизъмът са основна част от ООП, но в преобладаващата част от случаите, когато съставяте приложение, реалната нужда е от компоненти, които правят точно каквото трябва. Бихте желали да ги шляпнете във вашата програма както електронният инженер слага частите на своята платка (или даже, в случая на Java, на Web страница). Изглежда, също, че трябва да има начин да се ускори това „събиране на модули“ като стил на програмиране.

„Визуалното програмиране“ за пръв път стана успешно – много успешно – с майкрософтския Visual Basic (VB), след това с дизайна от второ поколение на Delphi на Borland (главния вдъхновител на дизайна на Java Beans). С тези програмни инструменти компонентите се представят визуално, което има смисъл понеже те обикновено са именно визуални компоненти като бутон или текстово поле. Визуалното представяне, фактически, е точно каквото ще представлява компонента в работещата програма. Така че част от процеса на визуално програмиране включва изтегляне на елемента от палета и слагането му във вашия образец. Инструментът който строи приложението пише код като направите това, а този код ще предизвика създаването на компонента като си пускате после програмата.

Просто разполагането на компонента на мястото му обикновено не е достатъчно за създаването на програма. Често трябва да се променят характеристиките на компонента, такива като цвета му, текста върху него, към каква база данни е свързан и т.н. Характеристиките които може да се променят по време на проектирането се наричат *properties*. Може да ги манипулирате по време на създаване на програмата с помощта на програмната среда, а когато съзدادете програмата тази конфигурация се запазва с оглед да може да бъде подмладена когато пускате програмата.

Вече трябва да сте свикнали с идеята, че елементът е нещо повече от характеристики; той също е и поведение. По време на проектирането поведението на компонента се определя частично от събития, тоест „Ето нещо, което може да му се случи на компонента.“

Обикновено решавате какво ще става като се случи събитие като свържете код със събитието.

Ето критичната част: програмната среда може динамично да разпитва (чрез рефлексия) компонента, за да намери какви свойства и събития той има. Щом се разбере какви са, средата може да ги изобрази и да позволи да бъдат променени (запазвайки каквото трябва когато построите програмата), също и събитията. Изобщо правите нещо като например двойно кликване върху елемента и средата произвежда код и го свързва с конкретното събитие. Това което вие трябва да направите после е да напишете кода, който осигурява необходимата реакция на въпросното събитие.

Всичко това значително увеличава частта от работата, прехвърлена върху програмната среда. Като резултат може да се съсредоточите върху изгледа на програмата и това което тя прави, като разчитате на средата да се справи със свързващите детайли. Причината визуалните среди за програмиране да са толкова успешни е че те драматично увеличават скоростта на производство на приложениета – разбира се, потребителския интерфейс, но често и други части на програмата също така.

Какво е Bean?

След като улегне прахоляка, тогава, компонентът е просто блок код, типично вграден в клас. Ключовият момент е възможността за програмната среда да намира характеристиките му – полета и събития. За създаване на VB компонент програмистът трябва да пише доста сложен код, следвайки някои конвенции, за да изложи характеристиките и събитията. Delphi беше втора генерация визуална среда за програмиране и езикът беше активно разработван за визуално програмиране така че е много по-лесно да се създаде визуален компонент. Обаче Java е придвижила работата до най-напредналия стадий с Java Beans, понеже Bean е просто клас. Няма нужда да пишете код за да направите нещо да е Bean. Единственото нужно нещо е, на практика, малко да промените начина на именуване на класовете си. Именно името казва на програмната среда дали нещото е свойство, събитие или просто обикновен метод.

В документацията на Java тоази конвенция за имената е погрешно означена с термина “design pattern.” Това е за нещастие, тъй като същите (виж глава 16) са достатъчно тежки и без този род обърквания. Това не е образец за проектиране, ами си е конвенция за имената и то доста проста:

1. За свойството с име **xxx** типично се създават два метода: **getXxx()** и **setXxx()**. Забележете промяната на регистъра на първата буква след **set** или **get**. Типа произведен от “**get**” метода е същия като типа на аргумента на “**set**” метода. Името на свойството и името на “**get**” и “**set**” не са свързани.
2. За булево свойство може да използвате “**get**” и “**set**” подхода от по-горе, но също може да използвате “**is**” вместо “**get**.”
3. Обикновените методи на един Bean не се съобразяват с конвенцията за имената, но са **public**.
4. За събитията се използва “слушателския” подход. Точно както сте го виждали: **addFooBarListener(FooBarListener)** и **removeFooBarListener(FooBarListener)** за обработка на **FooBarEvent**. Повечето вградените слушатели ще удовлетворяват нуждите ви, но също може да създавате свои собствени събития и слушатели за тях.

Точка 1 по-горе отговаря на нещо, което може да сте забелязали като промяна от Java 1.0 към Java 1.1: много имена на методи да претърпели малки, видимо безсмислени промени на имената. Сега се вижда, че повечето промени са поради съгласуването с “**get**” и “**set**” конвенциите за да може нещото да стане Bean.

Може да използваме тези неща за създаване на прост Bean:

```

//: frogbean:Frog.java
// A trivial Java Bean
package frogbean;
import java.awt.*;
import java.awt.event.*;

class Spots {}

public class Frog {
    private int jumps;
    private Color color;
    private Spots spots;
    private boolean jmpr;
    public int getJumps() { return jumps; }
    public void setJumps(int newJumps) {
        jumps = newJumps;
    }
    public Color getColor() { return color; }
    public void setColor(Color newColor) {
        color = newColor;
    }
    public Spots getSpots() { return spots; }
    public void setSpots(Spots newSpots) {
        spots = newSpots;
    }
    public boolean isJumper() { return jmpr; }
    public void setJumper(boolean j) { jmpr = j; }
    public void addActionListener(
        ActionListener l) {
        //...
    }
    public void removeActionListener(
        ActionListener l) {
        // ...
    }
    public void addKeyListener(KeyListener l) {
        // ...
    }
    public void removeKeyListener(KeyListener l) {
        // ...
    }
    // An "ordinary" public method:
    public void croak() {
        System.out.println("Ribbet!");
    }
} ///:~

```

Първо, може да се уверите че е просто клас. Обикновено всички полета ще са **private** и достъпни само чрез методи. Следвайки конвенцията за имената, свойствата са **jumps**, **color**, **spots** и **jumper** (забележете промяната на регистъра на първата буква на името на свойството). Макар и името на идентификатора да е идентично с вътрешното име в първите три случая, при **jumper** може да се види, че името на характеристиката не ви насила да използвате някакво конкретно име за вътрешните променливи (и, разбира се, даже да имате някаква вътрешна променлива за него свойство).

Събитията поддържани от този Bean са **ActionEvent** и **KeyEvent**, базирани на наименоването на "add" и "remove" методите за асоциирания слушател. Накрая, може да видите че

обикновения метод `croak()` е част от Bean-а просто защото е **public** метод, не защото отговаря на някаква конвенция за имената.

Извличане на BeanInfo с Introspector

Един от най-критичните за схемата Bean моменти е когато дръпнете един Bean от палитрата и го пуснете в работното място. Програмната среда трябва да е в състояние да създаде този Bean (което може да стане ако той има конструктор по подразбиране) и после, без да има достъп до сорса на Bean-а, да извлече всичката необходима информация за да създаде системата на характеристиките му и обработчиците на събития.

Част от решението е вече видна от края на глава 11: *рефлексията при Java 1.1* позволява да се открият всичките методи на анонимен клас. Това е перфектно за решаване на проблема на Bean без използване на допълнителни ключови думи и пр. както е при други визуални програмни езици. Фактически една от първопричините да се добави рефлексията към Java 1.1 беше за поддръжка на Beans (макар и чрез нея да се поддържат също и сериализацията на обекти и отдалеченото викане на методи). Така че може да се очаква създателят на визуална програмна среда да трябва да рефлексира всеки Bean и да ловува сред неговите методи за да извлече необходимото за него конкретен Bean.

Това е напълно възможно, но проектантите на Java искаха да дадат стандартен интерфейс за използване от всеки, не само да направят Beans прости за използване но да дадат стандартен път за създаване на по-сложни Beans. Този интерфейс е класът **Introspector** и най-важният метод в този клас е **static getBeanInfo()**. Давате **Class** манипулятор на този метод и той пълно разпилва класа и връща **BeanInfo** обект който после може да дисецирате за да намерите характеристики, методи и събития.

Обикновено не се занимавате с тези неща – вероятно ще си купите повечето Bean-ове готови от производители и не е необходимо да знаете магията която лежи под тях. Просто ще влечите вашите Beans върху полето, после ще конфигурирате параметрите им и пишете обработчици за събитията които ви интересуват. Обаче да се използва **Introspector** е поучително и образоващо упражнение като се накара да изобрази информация за Bean, така че ето инструмент който прави това (ще го намерите в поддиректорията **frogbean**):

```
//: c13:BeanDumper.java
// A method to introspect a Bean
import java.beans.*;
import java.lang.reflect.*;

public class BeanDumper {
    public static void dump(Class bean){
        BeanInfo bi = null;
        try {
            bi = Introspector.getBeanInfo(
                bean, java.lang.Object.class);
        } catch(IntrospectionException ex) {
            System.out.println("Couldn't introspect " +
                bean.getName());
            System.exit(1);
        }
        PropertyDescriptor[] properties =
            bi.getPropertyDescriptors();
        for(int i = 0; i < properties.length; i++) {
            Class p = properties[i].get.PropertyType();
            System.out.println(
```

```

    "Property type:\n " + p.getName());
System.out.println(
    "Property name:\n " +
    properties(i).getName());
Method readMethod =
    properties(i).getReadMethod();
if(readMethod != null)
    System.out.println(
        "Read method:\n " +
        readMethod.toString());
Method writeMethod =
    properties(i).getWriteMethod();
if(writeMethod != null)
    System.out.println(
        "Write method:\n " +
        writeMethod.toString());
System.out.println("=====");
}
System.out.println("Public methods:");
MethodDescriptor[] methods =
    bi.getMethodDescriptors();
for(int i = 0; i < methods.length; i++)
    System.out.println(
        methods(i).getMethod().toString());
System.out.println("=====");
System.out.println("Event support:");
EventSetDescriptor[] events =
    bi.getEventSetDescriptors();
for(int i = 0; i < events.length; i++) {
    System.out.println("Listener type:\n " +
        events(i).getListenerType().getName());
    Method[] lm =
        events(i).getListenerMethods();
    for(int j = 0; j < lm.length; j++)
        System.out.println(
            "Listener method:\n " +
            lm(j).getName());
    MethodDescriptor[] lmd =
        events(i).getListenerMethodDescriptors();
    for(int j = 0; j < lmd.length; j++)
        System.out.println(
            "Method descriptor:\n " +
            lmd(j).getMethod().toString());
    Method addListener =
        events(i).getAddListenerMethod();
    System.out.println(
        "Add Listener Method:\n " +
        addListener.toString());
    Method removeListener =
        events(i).getRemoveListenerMethod();
    System.out.println(
        "Remove Listener Method:\n " +
        removeListener.toString());
    System.out.println("=====");
}
}
// Dump the class of your choice:

```

```

public static void main(String[] args) {
    if(args.length < 1) {
        System.err.println("usage: \n" +
            "BeanDumper fully.qualified.class");
        System.exit(0);
    }
    Class c = null;
    try {
        c = Class.forName(args[0]);
    } catch(ClassNotFoundException ex) {
        System.err.println(
            "Couldn't find " + args[0]);
        System.exit(0);
    }
    dump(c);
}
} ///:~

```

BeanDumper.dump() е методът, в който се върши цялата работа. Първо се опитва да създаде **BeanInfo** обект и при успех вика методите на **BeanInfo** които дават информация за характеристики, методи и събития. В **Introspector.getBeanInfo()** ще видите и втори аргумент. Той казва на **Introspector** къде да спре по йерархията на наследяването. Тук тя спира преди да отдели всичките методи на **Object**, понеже не сме заинтересувани от тях.

За характеристиките, **getPropertyDescriptors()** връща масив от **PropertyDescriptor**. За всеки **PropertyDescriptor** може да извикате **get.PropertyType()** за да се намери клас на обекта който се подава вътре и навън чрез методите на характеристиката. Тогава, за всяка характеристика може да вземете нейния псевдоним (извлечен от имената на методите) с **getName()**, метод за четене с **getReadMethod()**, и метод за писане с **getWriteMethod()**. Двета последни метода връщат **Method** обект който може фактически да се използва за викане на съответния метод в обекта (това е част от рефлексията).

За публичните методи (включително методите на характеристиките), **getMethodDescriptors()** връща масив от **MethodDescriptor**. За всеки може да вземете асоцииран **Method** обект и да изведете неговото име.

За събитията: **getEventSetDescriptors()** връща масив от (какво друго?) **EventSetDescriptor**. Всеки от тях може да бъде прегледан за намиране на слушатели, методите на него слушателски клас и add- и remove-слушателските методи. Програмата **BeanDumper** печата цялата тази информация.

Ако извикате **BeanDumper** с **Frog** клас така:

```
| java BeanDumper frogbean.Frog
```

изходът, след мащане на ненужни тук детайли е като този:

```

class name: Frog
Property type:
Color
Property name:
color
Read method:
public Color getColor()
Write method:
public void setColor(Color)
=====
Property type:

```

Spots
Property name:
spots
Read method:
public Spots getSpots()
Write method:
public void setSpots(Spots)
=====

Property type:
boolean
Property name:
jumper
Read method:
public boolean isJumper()
Write method:
public void setJumper(boolean)
=====

Property type:
int
Property name:
jumps
Read method:
public int getJumps()
Write method:
public void setJumps(int)
=====

Public methods:
public void setJumps(int)
public void croak()
public void removeActionListener(ActionListener)
public void addActionListener(ActionListener)
public int getJumps()
public void setColor(Color)
public void setSpots(Spots)
public void setJumper(boolean)
public boolean isJumper()
public void addKeyListener(KeyListener)
public Color getColor()
public void removeKeyListener(KeyListener)
public Spots getSpots()
=====

Event support:
Listener type:
KeyListener
Listener method:
keyTyped
Listener method:
keyPressed
Listener method:
keyReleased
Method descriptor:
public void keyTyped(KeyEvent)
Method descriptor:
public void keyPressed(KeyEvent)
Method descriptor:
public void keyReleased(KeyEvent)
Add Listener Method:

```

public void addKeyListener(KeyListener)
Remove Listener Method:
public void removeKeyListener(KeyListener)
=====
Listener type:
ActionListener
Listener method:
actionPerformed
Method descriptor:
public void actionPerformed(ActionEvent)
Add Listener Method:
public void addActionListener(ActionListener)
Remove Listener Method:
public void removeActionListener(ActionListener)
=====
```

Това показва повечето от нещата които **Introspector** вижда като произвежда **BeanInfo** обект от вашия Bean. Може да видите че типа на характеристиката и нейното име са независими. Забележете долния регистър на името на характеристиката. (Единствено това не се случва ако името започва с повече от една голяма буква в ред.) И помнете че имената на методи които виждате тук (като `read` и `write` методите) са фактически произведени от **Method** обект който може да бъде използван за викане на асоциирания метод с обект.

Списъкът на публичните методи съдържа такива които не са асоциирани с характеристика или събитие, като `croak()`, както и такива, които са (ассоциирани-б.пр.). Това са всички методи които могат да се викат програмно за този Bean и инструментът за строене на приложение може да избере да ги изпише всичките когато викате методи, за да ви улесни в работата ви.

Накрая, може да се види, че събитията напълно се разделят в слушателя, методите му и `add-` и `remove-`слушател методите. Основно, щом веднъж имате **BeanInfo**, може да намерите всичко което има значение за този Bean. Може също да викате методите за него Bean, даже и да нямате никаква друга информация освен обекта (отново, черта на рефлексията).

По-усложнен Bean

Този следващ пример е по-усложнен, обаче несериозен. Това е основа която чертае малко кръгче около мишката щом последната бъде mrъдната. Когато натиснете (бутон на-б.пр) мишката, думата "Bang!" се появява в средата на екрана и се задейства слушателят на действието.

Характеристиките на кръгчето които може да мените са големината му и цвета, големината и самата дума която се изобразява като натиснете бутона. **BangBean** също има свой собствен `addActionListener()` и `removeActionListener()` така че може да свържете свой собствен слушател който да бъде задействан при натискане на **BangBean**. Трябва да сте в състояние да разпознаете поддръжката на характеристиките и събитията:

```

//: bangbean:BangBean.java
// A graphical Bean
package bangbean;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;

public class BangBean extends JPanel
    implements Serializable {
```

```

protected int xm, ym;
protected int cSize = 20; // Circle size
protected String text = "Bang!";
protected int fontSize = 48;
protected Color tColor = Color.red;
protected ActionListener actionListener;
public BangBean() {
    addMouseListener(new ML());
    addMouseMotionListener(new MML());
}
public int getCircleSize() { return cSize; }
public void setCircleSize(int newSize) {
    cSize = newSize;
}
public String getBangText() { return text; }
public void setBangText(String newText) {
    text = newText;
}
public int getFontSize() { return fontSize; }
public void setFontSize(int newSize) {
    fontSize = newSize;
}
public Color getTextColor() { return tColor; }
public void setTextColor(Color newColor) {
    tColor = newColor;
}
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.black);
    g.drawOval(xm - cSize/2, ym - cSize/2,
               cSize, cSize);
}
// This is a unicast listener, which is
// the simplest form of listener management:
public void addActionListener(
    ActionListener l)
    throws TooManyListenersException {
    if(actionListener != null)
        throw new TooManyListenersException();
    actionListener = l;
}
public void removeActionListener(
    ActionListener l) {
    actionListener = null;
}
class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        Graphics g = getGraphics();
        g.setColor(tColor);
        g.setFont(
            new Font(
                "TimesRoman", Font.BOLD, fontSize));
        int width =
            g.getFontMetrics().stringWidth(text);
        g.drawString(text,
                    (getSize().width - width) /2,
                    getSize().height/2);
    }
}

```

```

g.dispose();
// Call the listener's method:
if(actionListener != null)
    actionListener.actionPerformed(
        new ActionEvent(BangBean.this,
            ActionEvent.ACTION_PERFORMED, null));
}

}

class MML extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
        xm = e.getX();
        ym = e.getY();
        repaint();
    }
}

public Dimension getPreferredSize() {
    return new Dimension(200, 200);
}

// Testing the BangBean:
public static void main(String[] args) {
    BangBean bb = new BangBean();
    try {
        bb.addActionListener(new BBL());
    } catch(TooManyListenersException e) {}
    JFrame frame = new JFrame("BangBean Test");
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    frame.getContentPane().add(bb);
    frame.setSize(300,300);
    frame.setVisible(true);
}
}

// During testing, send action information
// to the console:
static class BBL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("BangBean action");
    }
}
}

} ///:~

```

Първото нещо което ще забележите е, че **BangBean** прилага интерфейса **Serializable**. Това значи че развойната среда може да "направи туршия (консервира-б.пр.)" всичката информация за **BangBean** чрез сериализация след като проектантът на програмата е нагласил стойностите за характеристиките. Когато се създава Bean-ът като част от способна да бъде пускана програма и когато тя се пуска, тези "турширани" характеристики се възстановяват така, че получавате точно както е било проектирано.

Може да се види че всички полета са **private**, което обикновено се прави с един Bean – позволява се достъп само чрез методи, обикновено използвайки схемата на "характеристиките".

Като гледате сигнатурата на **addActionListener()**, ще видите че той може да изхвърля **TooManyListenersException**. Това индицира уникаст, което означава че се сигнализира само

на един слушател когато възникне събитие. Обикновено вие ще използвате мултикаст за събитията така че за дадено събитие ще се съобщава на много слушатели. Това обаче води до въпроси за които вие няма да бъдете подгответи до следващата глава, така че това ще се разгледа пак там (под заглавие "Java Beans разгледани пак"). Уникастът заобикаля проблема.

Когато натиснете мишката, текстът сложен в средата на **BangBean** и ако полето **actionListener** не е нула **null**, вика се неговия **actionPerformed()**, създавайки нов **ActionEvent** обект в процеса. Винаги когато се мърда мишката, новите координати се вземат и фовът се прерисува (изтривайки всянакъв надпис както ще видите).

main() е добавено за да се подвولي да се тества програмата от командния ред. Когато един Bean е в развойната среда, **main()** няма да се използува, но е полезно да има **main()** във всеки от вашите Bean-ове понеже дава възможност за бързо тестване. **main()** създава **Frame** и слага **BangBean** вътре, закачайки прост **ActionListener** към **BangBean** за печatanе на конзолата винаги когато се случи **ActionEvent**. Обикновено, разбира се, развойната среда ще създада повечето код в Bean-а.

Когато пуснете **BangBean** чрез **BeanDumper** или сложите **BangBean** вътре в развойна среда работеща с Bean-ове, ще видите че има много повече характеристики и акции което е видно от горния код. Това е защото **BangBean** е наследено от **Canvas**, а **Canvas** е Bean, така че виждате неговите характеристики и събития също така.

Пакетиране на Bean

Преди да може да вземете Bean в подходяща развойна среда, той трябва да се сложи в стандартен Bean контейнер, което е JAR (Java ARchive) който включва всичките класове на Bean както и "манифестен" файл който назава "Това е Bean." Манифестният файл е просто текстов файл който следва конкретна форма. За **BangBean** манифестният файл изглежда така:

```
Manifest-Version: 1.0  
  
Name: bangbean/BangBean.class  
Java-Bean: True
```

Първата линия показва версията на схемата на манифеста, която до съобщение на Sun е 1.0. Вторият ред (празните редове не се считат) садаржа името на файла **BangBean.class**, третия назава "Това е Bean." Без третия ред развойната среда няма да разпознае класа като Bean.

Единствената тънка част е да се осигури коректен път в "Name:" полето. Ако погледнете **BangBean.java** ще видите че е в пакета **package bangbean** (и с това в поддиректория с име "bangbean" която не стои на classpath), а името в манифестния файл трябва да включва тази информация. Освен това трябва да сложите манифестния файл в директория над корена на пътя на вашия пакет, което в този случай значи слагане в директория над "bangbean" поддиректорията. После трябва да извикате **jar** от същата директория като на манифестния файл, ето така:

```
| jar cfm BangBean.jar BangBean.mf bangbean
```

Това предполага че искате получения JAR файл да бъде наречен **BangBean.jar** и че сте сложили манифеста във файл с име **BangBean.mf**.

Може би се чудите "Какво става с другите файлове които се генерираха когато компилирах **BangBean.java**?" Ами, те всички завършиха в **bangbean** поддиректорията и вие виждате, че аргументът на горния **jar** команден ред е **bangbean** поддиректорията. Когато дадете на **jar** име на поддиректория, той пакетира всичко от нея в произведенияния файл (включително, в този случай, сорса **BangBean.java** – бихте могли да изберете да не включвате сорса на ваши собствени Beanове). Освен това ако разплаковате JAR файла който току-що създадохте, ще

видите че вашият манифестен файл не е вътре, но **jar** е създал собствен манифест-файл (основан частично на вашия) наречен **MANIFEST.MF** и сложен в поддиректорията **META-INF** (за "мета-информация"). Ако отворите този файл ще видите, че е добавена информация за електронен подпис от **jar** за всеки файл, с формата:

```
Digest-Algorithms: SHA MD5  
SHA-Digest: pDpEAG9NaeCx8aFtqPl4udSX/O0=  
MD5-Digest: O4NcS1hE3Smnzlp2hj6qeg==
```

Изобщо, не трябва да се притеснявате за всички тези неща и ако правите промени трябва да напишете нов манифест и да извикате пак **jar** да създаде нов JAR файл за вашия Bean. Може също да добавите други Beanове към JAR просто като добавите тяхната информация във вашия манифест.

Трябва да се отбележи, че вероятно ще сложите всеки Bean в негова собствена поддиректория, понеже като създадете JAR файла давате на **jar** ютилитата името на поддиректорията и то (ютилитата) слага всичко от нея в JAR файла. Може да се види че и **Frog** и **BangBean** са в техни собствени поддиректории.

Щом имате вашия Bean както трябва в JAR файл може да го дадете на развойна система поддържаща Beans. Начинът това да се направи варира от инструмент към инструмент, но Sun доставя безплатна основа за тестване на Java Beans в техния "Beans Development Kit" (BDK) наречен "beanbox." (Разтоварете BDK от www.javasoft.com.) За да сложите вашия Bean в beanbox-а, копирайте JAR файла в "jars" поддиректорията на BDK преди да стартирате бийнбокса.

По-сложна поддръжка за Bean

Вижда се колко забележително просто е да се направи Bean. Но не сте ограничени с видяното до тук. Дизайнът на Java Bean дава начин за лесно навлизане но също може да се мащабира и с по-сложни решения. Тези ситуации са извън обхвата на тази книга но ще се въведат накратко. Може да намерите повече подробности на <http://java.sun.com/beans>.

Едно място където можете да добавите изтънченост е с характеристиките. Примерите по-горе показват само единични характеристики, но също е възможно да се представят повече характеристики в масив. Това се казва *indexed property*. Просто доставяте подходящи методи (отново следвайки конвенцията за имената на методите) и **Introspector** разпознава индексираните характеристики така че вашата развойна среда може да открие съответно.

Характеристиките могат да бъдат *bound*, което значи че ще сигнализират другите обекти чрез **PropertyChangeEvent**. Другите обекти може тогава да решат да се самоизменят съобразно промяната на Bean-а.

Характеристиките могат да бъдат *constrained*, което значи че другите обекти може да налагат вето на промяната ако е неприемлива. Другите обекти се сигнализират чрез **PropertyChangeEvent** и те могат да изхвърлят **PropertyVetoException** за да се предотврати промяната и възстановят старите стойности.

Може също да промените начина на представяне на вашия Bean по време на проектирането:

1. Може да зададете собствени характеристики за вашия конкретен Bean. Обикновените ще се използват за всички Beanове, но вашите ще се викат автоматично когато се избере вашия Bean.
2. Може да създадете собствен редактор за конкретна характеристика, така че да се използват обикновените характеристики, но когато се редактира вашата специална характеристика, ще се вика автоматично вашия редактор.

3. Може да създадете собствен **BeanInfo** клас за вашия Bean който дава информация различна от тази на **Introspector**.
4. Възможно е да се включва/изключва “експертния” режим за всички **FeatureDescriptor**и за работа с усложнените възможности и основните такива.

Повече за Beans

Има друг въпрос който не може да се включи тук. Винаги когато създавате Bean, трябва да очаквате че той ще работи в многонишкова среда. Това значи че трябва да разбирате въпросите на многонишовостта, с които ще се запознаете в следващата глава. Там ще намерите секция “Java Beans разгледани пак” в която се показва проблемът и неговото решение.

Въведение в Swing⁹

 Ад извъряването на вашия път дотук и като видяхте огромните промени в Swing (види се, авторът счита че тя е позната от предишни ми книги например - б.пр.) (макар че, ако може да си спомните толкова далечни неща, Sun твърдеше че Java е “стабилен” език когато той за пръв път се появи), може все пак да имате усещането, че нещата не са донаправени. Да, сега има добър модел на събитията, JavaBeans са чудесни за повторно използване на кода. Но GUI компонентите пак са някакси минимални, примитивни и тромави.

Ето тук се намесва Swing. Swing библиотеката се появи след Java 1.1 така че естествено е да предполагате, че е част от Java 2. Обаче тя е проектирана да работи с Java 1.1 като добавка. По този начин не е необходимо да чакате вашата платформа да почне да поддържа Java 2 за да може да се радвате на библиотека с UI компоненти. Вашите потребители може да им се наложи да си свалят Swing библиотеката ако я нямат към Java 1.1 поддръжката, а това може да има някои спънки. Но работи.

Swing съдържа всичко, което ни липсваше в тази глава: съдържанието на един модерен UI, всичко от бутони които имат картички до дървета и мрежи. Библиотеката е голяма, но е проектирана да има подходяща за изпълняваната задача сложност – ако нещо е просто, няма нужда да се пише много код, но щом се опитате да направите повече вашият код се оказва с нарастваща сложност. Това значи леснота при започването, но може да получите и мощност, ако ви е необходима.

Swing е дълбоко нещо. Тази секция не се опитва да бъде изчерпателна, ами ви запознава с мощността и простотата на Swing за да може да започнете използването на библиотеката. Моля имайте пред вид че това което виждате тук нарочно е направено да бъде просто. Ако искате повече, тогава Swing вероятно може да ви го даде ако сте склонни да лолувате в онлайн документацията на Sun.

Ползите от Swing

Когато започнете да използвате библиотеката Swing ще видите че тя е огромна стъпка напред. Swing компонентите са Beans (и затова използват модела на събития на Java 1.1), така че могат да бъдат изролзвани във всякакви развойни среди, които поддържат Beans. Swing дава пълно множество от UI компоненти. За скорост всички компоненти са леки (не се използват “реел” компоненти), а за преносимост Swing е написана изцяло на Java.

⁹ At the time this section was written, the Swing library had been pronounced “frozen” by Sun, so this code should compile and run without problems as long as you’ve downloaded and installed the Swing library. (You should be able to compile one of Sun’s included demonstration programs to test your installation.) If you do encounter difficulties, check www.BruceEckel.com for updated code.

Много от хубавото на Swing би могло да бъде наречено "ортогоналност на използването;" тоест когато схванете основните идеи може да ги прилагате навсякъде. Първо, поради конвенциите за имената на Beans повечето време през което пишех упражненията аз можех да позная имената на методите и то от пръв път, без да гледам някъде. Това несъмнено е знак за качество на всеки дизайн на библиотека. Освен това изобщо може да включвате компоненти към други компоненти и нещата работят.

Навигацията чрез клавиатурата е автоматична – може да използвате Swing приложение без мишка, но няма нужда да програмирате нищо в повече (старият AWT изискваше малко недодялан код за да се осигури навигация с клавиатурата). Скролингът е без усилия – просто обгръщате всяка компонент в **JScrollPane** както го слагате на място. Други черти като хвърчащите подсказки например изискват един ред за реализацията си.

Swing също поддържа нещо наречено "закачвам изглед и чувство," което значи че изгледът на UI може да бъде автоматично менен съобразно очакванията на потребители работещи под различни платформи и операционни системи. Даже е възможно да изобретите собствен изглед и чувство/интуитивност при работа.

Лесно преобразуване

Ако сте се борили дълго и упорите да построите вашия UI за Java 1.1 няма да искате да го изхвърлите за да минете към Swing. За щастие библиотеката е проектирана да позволява лесно преобразуване – в много случай може просто да сложите 'J' пред имената на класовете във вашите стари AWT компоненти. Ето пример който да направи това обичайно:

```
//: c13:ButtonDemo.java
// Looks like Java 1.1 but with J's added
// <applet code=ButtonDemo width=200 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ButtonDemo extends JApplet {
    JButton
        b1 = new JButton("JButton 1"),
        b2 = new JButton("JButton 2");
    JTextField t = new JTextField(20);
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        ActionListener al = new ActionListener() {
            public void actionPerformed(ActionEvent e){
                String name =
                    ((JButton)e.getSource()).getText();
                t.setText(name + " Pressed");
            }
        };
        b1.addActionListener(al);
        cp.add(b1);
        b2.addActionListener(al);
        cp.add(b2);
        cp.add(t);
    }
    public static void main(String args[]) {
        JApplet applet = new ButtonDemo();
```

```

JFrame frame = new JFrame("TextAreaNew");
frame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e){
        System.exit(0);
    }
});
frame.getContentPane().add(applet);
frame.setSize(300,100);
applet.init();
applet.start();
frame.setVisible(true);
}
} //:~

```

Има нов **import** оператор, но всичко друго изглежда като Java 1.1 AWT с добавка на J-та. Също, не просто **add()** нещо към **JFrame** на Swing, ами трябва да вземете "content pane" първо, както се вижда по-горе. Лесно може да се възползвате от всичко това в Swing с просто преобразумане.

Поради оператора **package** ще трябва да викате тази програма с:

```
| java c13.swing.JButtonDemo
```

Всички програми от тази секция ще се викат по подобен начин.

Работна рамка за дисплей

Макар и приложения които могат да бъдат пускани и като аплети да са ценни, ако се използват навсякъде те стават разточителни. Вместо това ще се използва работна рамка за Swing примерите в останалата част на тази секция:

```

//: c13>Show.java
// Tool for displaying Swing demos
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Show {
    public static void
    inFrame(JPanel jp, int width, int height) {
        String title = jp.getClass().toString();
        // Remove the word "class":
        if(title.indexOf("class") != -1)
            title = title.substring(6);
        JFrame frame = new JFrame(title);
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
        frame.getContentPane().add(
            jp, BorderLayout.CENTER);
        frame.setSize(width, height);
        frame.setVisible(true);
    }
} //:~

```

Класовете които ще искат да се изобразяват ще наследяват от **JPanel** и после ще добавят всякакви изображаеми компоненти към себе си. Накрая те създават **main()** съдържащ реда:

```
| Show.inFrame(new MyClass(), 500, 300);
```

Където последните два аргумента са ширината и височината на дисплея (на прозореца - б.пр.).

Забележете че заглавието на **JFrame** се получава чрез RTTI.

Хвърчащи подсказки

Почти всички класове които ще използвате в свой потребителски интерфейс ще бъдат извлечени от **JComponent**, който съдържа метод наречен **setToolTipText(String)**. Така за практически всеки компонент който слагате на екрана е достатъчно да напишете (за обект **jc** от всякакъв **JComponent**-извлечен клас):

```
| jc.setToolTipText("My tip");
```

и когато мишката стои над въпросния **JComponent** предварително определен период от време близо до мишката ще се появи мъничка кутия съдържаща подсказката.

Ограничителни линии

JComponent също съдържа метод наречен **setBorder()**, който позволява да сложите различни интересни ограничители около всеки компонент. Следния пример показва наличните възможности използвайки метод **showBorder()** който създава **JPanel** и слага бордер във всеки отделен случай. Също се използва RTTI за намиране на името на бордера който използвате (с мащата информация за пътя), сетне слага името в **JLabel** в средата на панела:

```
//: c13:Borders.java
// Different Swing borders
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;

public class Borders extends JPanel {
    static JPanel showBorder(Border b) {
        JPanel jp = new JPanel();
        jp.setLayout(new BorderLayout());
        String nm = b.getClass().toString();
        nm = nm.substring(nm.lastIndexOf('.') + 1);
        jp.add(new JLabel(nm, JLabel.CENTER),
              BorderLayout.CENTER);
        jp.setBorder(b);
        return jp;
    }
    public Borders() {
        setLayout(new GridLayout(2,4));
        add(showBorder(new TitledBorder("Title")));
        add(showBorder(new EtchedBorder()));
        add(showBorder(new LineBorder(Color.blue)));
        add(showBorder(
            new MatteBorder(5,5,30,30,Color.green)));
        add(showBorder(
            new BevelBorder(BevelBorder.RAISED)));
    }
}
```

```

    add(showBorder(
        new SoftBevelBorder(BevelBorder.LOWERED)));
    add(showBorder(new CompoundBorder(
        new EtchedBorder(),
        new LineBorder(Color.red))));}
}

public static void main(String args) {
    Show.inFrame(new Borders(), 500, 300);
}

} //:~

```

Повечето примери в секцията използват **TitledBorder**, но може да се види че и останалите лесно се използват. Може също да създадете свои собствени бордери и да ги сложите около бутони, етикети и т.н. – всичко извлечено от **JComponent**.

БУТОНИ

Swing добавя различни типове бутони, променя организацията на избора на компоненти: всички бутони, отметки, радиобутони, даже елементите на меню са наследени от **AbstractButton** (който, след като са включени и менюта, би следвало да се нарече "AbstractChooser" или нещо с подобна степен на общност). Ще видите използването на елементи на меню накъсо, но следния пример показва различни варианти на бутони:

```

//: c13:Buttons.java
// Various Swing buttons
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.plaf.basic.*;
import javax.swing.border.*;

public class Buttons extends JPanel {
    JButton jb = new JButton("JButton");
    BasicArrowButton
        up = new BasicArrowButton(
            BasicArrowButton.NORTH),
        down = new BasicArrowButton(
            BasicArrowButton.SOUTH),
        right = new BasicArrowButton(
            BasicArrowButton.EAST),
        left = new BasicArrowButton(
            BasicArrowButton.WEST);
    public Buttons() {
        add(jb);
        add(new JToggleButton("JToggleButton"));
        add(new JCheckBox("JCheckBox"));
        add(new JRadioButton("JRadioButton"));
        JPanel jp = new JPanel();
        jp.setBorder(new TitledBorder("Directions"));
        jp.add(up);
        jp.add(down);
        jp.add(left);
        jp.add(right);
        add(jp);
    }
    public static void main(String args) {
        Show.inFrame(new Buttons(), 300, 200);
    }
}

```

```
    }
} //:/~
```

JButton изглежда като AWT бутона, но повече може да се прави с него (като добавяне на образи както ще видите по-нататък). В **javax.swing.plaf.basic** има също **BasicArrowButton** който е удобен.

Когато стартирате примера, ще видите че превключващия бутон задържа на последната си позиция, включено или изключено. Но отметките и радиобутона се държат по един и същ начин, просто ставайки включени или изключени (те са наследени от **JToggleButton**).

Групи бутони

Ако искате група радиобутони да работи по схемата "изключващо или", трябва да ги добавите към бутона група, по подобен но по-малко тромав начин от стария AWT. Но както примерът по-долу демонстрира, всеки **AbstractButton** може да бъде добавен към **ButtonGroup**.

За да се избегне повтарянето на много код този пример използва рефлексия за генерацията на групи от бутони. Това се вижда в **makeBPanel**, който създава група бутони и **JPanel**, а за всеки **String** в масива който е втори аргумент на **makeBPanel()** добавя обект от класа който е представен с първия аргумент:

```
//: c13:ButtonGroups.java
// Uses reflection to create groups of different
// types of AbstractButton.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import java.lang.reflect.*;

public class ButtonGroups extends JPanel {
    static String[] ids = {
        "June", "Ward", "Beaver",
        "Wally", "Eddie", "Lumpy",
    };
    static JPanel
    makeBPanel(Class bClass, String[] ids) {
        ButtonGroup bg = new ButtonGroup();
        JPanel jp = new JPanel();
        String title = bClass.getName();
        title = title.substring(
            title.lastIndexOf('.') + 1);
        jp.setBorder(new TitledBorder(title));
        for(int i = 0; i < ids.length; i++) {
            AbstractButton ab = new JButton("failed");
            try {
                // Get the dynamic constructor method
                // that takes a String argument:
                Constructor ctor = bClass.getConstructor(
                    new Class[] { String.class });
                // Create a new object:
                ab = (AbstractButton)ctor.newInstance(
                    new Object[]{ids[i]});
            } catch(Exception ex) {
                System.out.println("can't create " +
                    bClass);}
```

```

        }
        bg.add(ab);
        jp.add(ab);
    }
    return jp;
}
public ButtonGroups() {
    add(makeBPanel(JButton.class, ids));
    add(makeBPanel(JToggleButton.class, ids));
    add(makeBPanel(JCheckBox.class, ids));
    add(makeBPanel(JRadioButton.class, ids));
}
public static void main(String args[]) {
    Show.inFrame(new ButtonGroups(), 500, 300);
}
} //:~

```

Заглавието за бордера е взето от името на класа, с махната информация за пътя. **AbstractButton** е инициализиран като **JButton** който има етикет "Failed" така че ако пренебрегнете съобщението за изключение, пак ще виждате проблема на екрана. Методът **getConstructor()** дава **Constructor** който взема масива от аргументи от типовете на масива **Class** дадени на **getConstructor()**. После остава да се извика **newInstance()**, като му се даде масив от **Object** съдържащ вашите фактически аргументи – в този случай **Stringa** от масива **ids**.

Това малко усложнява иначе простия процес. За да имаме "изключващо или" на бутоните създаваме групата бутона и добавяме всеки бутон който ще е с такова поведение в нея. Когато стартирате програмата, ще видите че всички бутона освен **JButton** показват такова поведение на "изключващо или".

ИКОНИ

Може да използвате **Icon** в **JLabel** или каквото и да е което наследява от **AbstractButton** (включително **JButton**, **JCheckbox**, **JradioButton** и различните видове **JMenuItem**). Използването на **Iconi** с **Jlabeli** е твърде праволинейно (ще видите пример по-късно). Следния пример изследва допълнителни пътища за използване на **Iconi** с бутона и техните наследници.

Може да използвате всякали **gif** файлове които искате, а тези които виждате тук са част от книгата, достъпна на www.BruceEckel.com. За да отворите файл и да вземете образа просто създавате **ImageIcon** и му давате името на файла. От там нататък може да използвате получената **Icon** във вашата програма.

```

//: c13:Faces.java
// Icon behavior in JButtons
import javax.swing.*;
import java.awt.*;
import java.awt.event;

public class Faces extends JPanel {
    static Icon[] faces = {
        new ImageIcon("face0.gif"),
        new ImageIcon("face1.gif"),
        new ImageIcon("face2.gif"),
        new ImageIcon("face3.gif"),
        new ImageIcon("face4.gif"),
    };
    JButton
    jb = new JButton("JButton", faces[3]),

```

```

jb2 = new JButton("Disable");
boolean mad = false;
public Faces() {
    jb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
            if(mad) {
                jb.setIcon(faces(3));
                mad = false;
            } else {
                jb.setIcon(faces(0));
                mad = true;
            }
            jb.setVerticalAlignment(JButton.TOP);
            jb.setHorizontalAlignment(JButton.LEFT);
        }
    });
    jb.setRolloverEnabled(true);
    jb.setRolloverIcon(faces(1));
    jb.setPressedIcon(faces(2));
    jb.setDisabledIcon(faces(4));
    jb.setToolTipText("Yow!");
    add(jb);
    jb2.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
            if(jb.isEnabled()) {
                jb.setEnabled(false);
                jb2.setText("Enable");
            } else {
                jb.setEnabled(true);
                jb2.setText("Disable");
            }
        }
    });
    add(jb2);
}
public static void main(String args[]) {
    Show.inFrame(new Faces(), 300, 200);
}
} //:~

```

Една **Icon** може да бъде използвана в много конструктори, но също може да използвате **setIcon()** за да добавите или смените **Icon**. Този пример също показва как **JButton** (или произволен **AbstractButton**) може да сложи различни видове икони които се появяват при ставането на нещо с бутона: когато е натиснат, неактивен, "rolled over" (мишката минава отгоре му без кликване). Ще видите че това дава на бутона малко анимиран характер.

Забележете че към бутона е добавена и хвърчаща подсказка.

Менюта

Менютата са много подобрени и по-гъвкави в Swing – например може да ги използвате навсякъде, включително в панели и аплети. Синтаксисът на използването им е повечето като на стария AWT, а това съхранява същия проблем от стария AWT: трябва да се кодират твърдо всички менюта и няма поддръжка на менюта във вид на ресурси (която, заедно с всичко друг, би улеснила приспособяването на менютата за други езици). Освен това кодът за менюта става твърде голям и понякога объркан. Следният подход прави стъпка по посока решаването на този проблем чрез слагането на информацията за всяко меню в двумерен масив от

Object (по този начин може да сложите каквото и да е в масив). Този масив е организиран така, че първият ред представя името на менюто, а останалите представят елементите на менюто и характеристиките им. Ще забележите че редовете не е необходимо да са с една форма от едно меню към друго – доколкото вашият код знае кое къде е, всеки ред може да бъде напълно различен.

```
//: c13:Menus.java
// A menu-building system; also demonstrates
// icons in labels and menu items.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Menus extends JPanel {
    static final Boolean
        bT = new Boolean(true),
        bF = new Boolean(false);
    // Dummy class to create type identifiers:
    static class MType { MType(int i) {} };
    static final MType
        mi = new MType(1), // Normal menu item
        cb = new MType(2), // Checkbox menu item
        rb = new MType(3); // Radio button menu item
    JTextField t = new JTextField(10);
    JLabel l = new JLabel("Icon Selected",
        Faces.faces(0), JLabel.CENTER);
    ActionListener a1 = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            t.setText(
                ((JMenuItem)e.getSource()).getText());
        }
    };
    ActionListener a2 = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            JMenuItem mi = (JMenuItem)e.getSource();
            l.setText(mi.getText());
            l.setIcon(mi.getIcon());
        }
    };
    // Store menu data as "resources":
    public Object[] fileMenu = {
        // Menu name and accelerator:
        { "File", new Character('F') },
        // Name type accel listener enabled
        { "New", mi, new Character('N'), a1, bT },
        { "Open", mi, new Character('O'), a1, bT },
        { "Save", mi, new Character('S'), a1, bF },
        { "Save As", mi, new Character('A'), a1, bF },
        { null }, // Separator
        { "Exit", mi, new Character('x'), a1, bT },
    };
    public Object[] editMenu = {
        // Menu name:
        { "Edit", new Character('E') },
        // Name type accel listener enabled
        { "Cut", mi, new Character('t'), a1, bT },
        { "Copy", mi, new Character('C'), a1, bT },
    };
}
```

```

{ "Paste", mi, new Character('P'), a1, bT },
{ null }, // Separator
{ "Select All", mi,new Character('I'),a1,bT},
};

public Object() helpMenu = {
// Menu name:
{ "Help", new Character('H') },
// Name type accel listener enabled
{ "Index", mi, new Character('I'), a1, bT },
{ "Using help", mi,new Character('U'),a1,bT},
{ null }, // Separator
{ "About", mi, new Character('t'), a1, bT },
};

public Object() optionMenu = {
// Menu name:
{ "Options", new Character('O') },
// Name type accel listener enabled
{ "Option 1", cb, new Character('1'), a1,bT},
{ "Option 2", cb, new Character('2'), a1,bT},
};

public Object() faceMenu = {
// Menu name:
{ "Faces", new Character('a') },
// Optinal last element is icon
{ "Face 0", rb, new Character('0'), a2, bT,
Faces.faces(0) },
{ "Face 1", rb, new Character('1'), a2, bT,
Faces.faces(1) },
{ "Face 2", rb, new Character('2'), a2, bT,
Faces.faces(2) },
{ "Face 3", rb, new Character('3'), a2, bT,
Faces.faces(3) },
{ "Face 4", rb, new Character('4'), a2, bT,
Faces.faces(4) },
};

public Object() menuBar = {
fileMenu, editMenu, faceMenu,
optionMenu, helpMenu,
};

static public JMenuBar
createMenuBar(Object() menuBarData) {
JMenuBar menuBar = new JMenuBar();
for(int i = 0; i < menuBarData.length; i++)
menuBar.add(
createMenu((Object())menuBarData(i)));
return menuBar;
}

static ButtonGroup bgroup;
static public JMenu
createMenu(Object() menuData) {
JMenu menu = new JMenu();
menu.setText((String)menuData(0)(0));
menu.setMnemonic(
((Character)menuData(0)(1)).charValue());
// Create redundantly, in case there are
// any radio buttons:
bgroup = new ButtonGroup();
}

```

```

for(int i = 1; i < menuData.length; i++) {
    if(menuData(i)(0) == null)
        menu.add(new JSeparator());
    else
        menu.add(createMenuItem(menuData(i)));
}
return menu;
}

static public JMenuItem
createMenuItem(Object() data) {
    JMenuItem m = null;
    MType type = (MType)data(1);
    if(type == mi)
        m = new JMenuItem();
    else if(type == cb)
        m = new JCheckBoxMenuItem();
    else if(type == rb) {
        m = new JRadioButtonMenuItem();
        bgroup.add(m);
    }
    m.setText((String)data(0));
    m.setMnemonic(
        ((Character)data(2)).charValue());
    m.addActionListener(
        (ActionListener)data(3));
    m.setEnabled(
        ((Boolean)data(4)).booleanValue());
    if(data.length == 6)
        m.setIcon((Icon)data(5));
    return m;
}

Menus() {
    setLayout(new BorderLayout());
    add(createMenuBar(menuBar),
        BorderLayout.NORTH);
    JPanel p = new JPanel();
    p.setLayout(new BorderLayout());
    p.add(t, BorderLayout.NORTH);
    p.add(l, BorderLayout.CENTER);
    add(p, BorderLayout.CENTER);
}

public static void main(String args()) {
    Show.inFrame(new Menus(), 300, 200);
}

} //:~

```

Целта е да се позволи на програмиста просто да състави таблица за представянето на всяко меню, вместо да пише редове код които правят менютата. Всяка таблица прави едно меню и първият ред съдържа името на менюто и клавиатурния ускорител. Останалите редове съдържат данни за всеки елемент на менюто: стрингът който трябва да се сложи в елемента, какъв е типът на елемента, клавиатурният ускорител, слушателят който се задейства когато този елемент е избран и дали елементът е активен. Ако ред започва с **null** той се третира като разделител.

За да се избегне многоократното и досадно създаване на **Boolean** обекти и флагове за типа, те се създават като **static final** стойности в началото на класа: **bT** и **bF** за представяне на **Boolean**и и различни обекти от класа **MType** за описание на нормални елементи на меню (**mi**),

отметки-елементи на меню (**cb**), радиобутони-елементи на меню (**rb**). Помните че масив от **Object** може да съдържа само **Object** манипулатори но на и примитиви.

Този пример също показва как **Jlabel** и **JMenuItem** (и наследниците им) могат да владеят **Icon**. **Icon** е сложена в **JLabel** чрез конструктора му и сменена когато съответният елемент на менюто е избран.

Масивът **menuBar** съдържа манипулаторите към всичките менюта в реда в който ги искате да се появяват в лентата на менюто. Давате този масив на **createMenuBar()**, който го разделя на индивидуални масиви от данни за менютата, давайки всеки на **createMenu()**. Този метод на свой ред взема първата линия от данните на менюто и създава **JMenu** от тях, после вика **createMenuItem()** за всеки от останалите редове от данните на менюто. Накрая **createMenuItem()** дели всеки ред от данни на менюто и определя типа на елемента и атрибутите му, та създава елемента съответно. Накрая, както може да видите в конструктора на **Menus()**, за да създава меню от тези таблици пишем **createMenuBar(menuBar)** и всичко се прави рекурсивно.

Този пример не се занимава с постройката на каскадирани менюта, но би трявало да имате достатъчна концепция да можете да ги добавите ако ви са нужни.

Изскачащи менюта

Най-праволинейният начин да се приложи **JPopupMenu** е да се създава вътрешен клас който разширява **MouseAdapter**, после да се добави обект от този вътрешен клас към всеки компонент който трябва да има поведението на изскачащо меню:

```
//: c13:Popup.java
// Creating popup menus with Swing
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Popup extends JPanel {
    JPopupMenu popup = new JPopupMenu();
    JTextField t = new JTextField(10);
    public Popup() {
        add(t);
        ActionListener al = new ActionListener() {
            public void actionPerformed(ActionEvent e){
                t.setText(
                    ((JMenuItem)e.getSource()).getText());
            }
        };
        JMenuItem m = new JMenuItem("Hither");
        m.addActionListener(al);
        popup.add(m);
        m = new JMenuItem("Yon");
        m.addActionListener(al);
        popup.add(m);
        m = new JMenuItem("Afar");
        m.addActionListener(al);
        popup.add(m);
        popup.addSeparator();
        m = new JMenuItem("Stay Here");
        m.addActionListener(al);
        popup.add(m);
        PopupListener pl = new PopupListener();
        t.addMouseListener(pl);
    }
}
```

```

    addMouseListener(pl);
    t.addMouseListener(pl);
}
class PopupListener extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        maybeShowPopup(e);
    }
    public void mouseReleased(MouseEvent e) {
        maybeShowPopup(e);
    }
    private void maybeShowPopup(MouseEvent e) {
        if(e.isPopupTrigger()) {
            popup.show(
                e.getComponent(), e.getX(), e.getY());
        }
    }
}
public static void main(String args[]) {
    Show.inFrame(new Popup(),200,150);
}
} //:~

```

Същия **ActionListener** е добавен към всеки **JMenuItem**, така че намира текста на етикета на менюто и го вмъква в **JTextField**.

Списъчни кутии и комбинирани кутии

Списъчните кутии и комбобоксовете работят в Swing доста подобно на това в стария AWT, но също имат увеличена функционалност ако ви трябва. Освен това е подобрено удобството. Например **JList** има конструктор който взема масив от **String**ове за изобразяване (достатъчно злополучно тази черта я няма **JComboBox**). Ето прост пример който показва използването на всеки:

```

//: c13>ListCombo.java
// List boxes & Combo boxes
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ListCombo extends JPanel {
    public ListCombo() {
        setLayout(new GridLayout(2,1));
        JList list = new JList(ButtonGroups.ids);
        add(new JScrollPane(list));
        JComboBox combo = new JComboBox();
        for(int i = 0; i < 100; i++)
            combo.addItem(Integer.toString(i));
        add(combo);
    }
    public static void main(String args[]) {
        Show.inFrame(new ListCombo(),200,200);
    }
} //:~

```

Нещо друго на пръв поглед несполучливо е че **Jlistовете** не дават автоматично скролинг, макар че това се очаква от тях. Добавянето на поддръжка за скролни ленти се оказва доста

лесно, както е показано по-горе – просто обгръщате **JList** в **JScrollPane** и всички детайли са автоматично оправени за вас.

Плъзгачи и ленти за напредъка

Плъзгачът дава възможност на потребителя да въвежда данни мърдайки точка напред-назад, което е интуитивно в някои ситуации (управление на силата на звука, например). Лентата за напредък показва относително данните от “пълно” до “празно” така че потребителят получава представа за перспективата (кога ще завърши работата-б.пр.). Моят любим пример на тази тема е да закача плъзгача към лентата така че когато мърдате плъзгача лентата да показва това съответно:

```
//: c13:Progress.java
// Using progress bars and sliders
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;
import javax.swing.border.*;

public class Progress extends JPanel {
    JProgressBar pb = new JProgressBar();
    JSlider sb =
        new JSlider(JSlider.HORIZONTAL, 0, 100, 60);
    public Progress() {
        setLayout(new GridLayout(2,1));
        add(pb);
        sb.setValue(0);
        sb.setPaintTicks(true);
        sb.setMajorTickSpacing(20);
        sb.setMinorTickSpacing(5);
        sb.setBorder(new TitledBorder("Slide Me"));
        pb.setModel(sb.getModel()); // Share model
        add(sb);
    }
    public static void main(String args[]) {
        Show.inFrame(new Progress(), 200, 150);
    }
} ///:~
```

The **JProgressBar** is fairly straightforward, but the **JSlider** has a lot of options, such as the orientation and major and minor tick marks. Notice how straightforward it is to add a titled border.

Дървета

Използването на **JTree** може да бъде просто като да се напише:

```
add(new JTree(
    new Object() {"this", "that", "other"}));
```

Това изобразява примитивно дърво. API-то за дървета е огромно, обаче – едно от най-големите в Swing. Иди се може да се прави почти всичко с дърветата, но по-засуканите работи могат да искат малко изследване и експерименти.

За щастие има следно положение дадено в библиотеката: компоненти на дървото “по подразбиране” които изобщо правят това, от което се нуждаете. Така че повечето време

може да използвате тези компоненти, а само в специални случаи ще трябва да се заровите и да разберете дърветата по-от дълбоко.

Следния пример използва компоненти на дървото "по подразбиране" за да изобрази дърво в аплет. Когато натиснете бутон, ново поддърво се добавя под текущо избрания възел (ако не е избран възел, използва се корена):

```
//: c13:Trees.java
// Simple Swing tree example. Trees can be made
// vastly more complex than this.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.tree.*;

// Takes an array of Strings and makes the first
// element a node and the rest leaves:
class Branch {
    DefaultMutableTreeNode r;
    public Branch(String[] data) {
        r = new DefaultMutableTreeNode(data[0]);
        for(int i = 1; i < data.length; i++)
            r.add(new DefaultMutableTreeNode(data[i]));
    }
    public DefaultMutableTreeNode node() {
        return r;
    }
}

public class Trees extends JPanel {
    String[] data = {
        {"Colors", "Red", "Blue", "Green"}, 
        {"Flavors", "Tart", "Sweet", "Bland"}, 
        {"Length", "Short", "Medium", "Long"}, 
        {"Volume", "High", "Medium", "Low"}, 
        {"Temperature", "High", "Medium", "Low"}, 
        {"Intensity", "High", "Medium", "Low"}, 
    };
    static int i = 0;
    DefaultMutableTreeNode root, child, chosen;
    JTree tree;
    DefaultTreeModel model;
    public Trees() {
        setLayout(new BorderLayout());
        root = new DefaultMutableTreeNode("root");
        tree = new JTree(root);
        // Add it and make it take care of scrolling:
        add(new JScrollPane(tree),
            BorderLayout.CENTER);
        // Capture the tree's model:
        model =(DefaultTreeModel)tree.getModel();
        JButton test = new JButton("Press me");
        test.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e){
                if(i < data.length) {
                    child = new Branch(data[i++]).node();
                    // What's the last one you clicked?
                    model.insertNodeInto(child, chosen, chosen.getIndex());
                }
            }
        });
        add(test, BorderLayout.SOUTH);
    }
}
```

```

chosen = (DefaultMutableTreeNode)
tree.getLastSelectedPathComponent();
if(chosen == null) chosen = root;
// The model will create the
// appropriate event. In response, the
// tree will update itself:
model.insertNodeInto(child, chosen, 0);
// This puts the new node on the
// currently chosen node.
}
}
});
// Change the button's colors:
test.setBackground(Color.blue);
test.setForeground(Color.white);
JPanel p = new JPanel();
p.add(test);
add(p, BorderLayout.SOUTH);
}
public static void main(String args[]) {
Show.inFrame(new Trees(),200,500);
}
} //:~

```

Първия клас, **Branch**, е инструмент за вземане на масив от **String** и построяване **DefaultMutableTreeNode** с първия **String** като корен и останалите **String**ове в масив от листа. После може да се извика **node()** за да се създава корена на този "клон."

Класът **Trees** съдържа двумерен масив от **String**ове от които могат да се правят **Branch**ове и **static int i** за да се брои в масива. Обектът **DefaultMutableTreeNode** държи възлите, но физическото представяне на екрана се управлява от **JTree** и асоциирания му модел, **DefaultTreeModel**. Забележете че когато **JTree** е добавено към аплета, то е обгърнато в **JScrollPane** – това е за автоматичен скролинг.

JTree се управлява от неговия модел. Когато направите промяна в него, той генерира събитие което причинява изпълнението на всички необходими осъвременявания от **JTree** що се отнася до видимия образ на дървото. В **init()**, моделът се хваща от **getModel()**. Когато се натисне бутоњът, създава се нов "клон". Когато текущо избрания компонент е намерен (или корена ако нищо не е избрано), метода на модела **insertNodeInto()** прави всичката работа за осъвременяване на дървото и неговото изображение.

В повечето случаи пример като горния ще ви даде всичко което искате от дървото. Обаче дърветата имат мощност да се прави почти всичко, което може да си представите – навсякъде където виждате думата "default" в горния пример, може да замените с ваш собствен клас за постигане на определено поведение. Но бъдете предупредени: почти всички от тези класове имат голям интерфейс, така че може да трябва много време да научите интимностите на дърветата.

Таблици

Както дърветата, таблиците в Swing са големи и мощни. Те са предимно замислени да бъдат популярен "мрежов" интерфейс към бази данни чрез Java Database Connectivity (JDBC, обсъдено в глава 15) и затова имат огромна гъвкавост, която се заплаща със сложност. Има достатъчно за пълна програма от рода на Ексел и може да заеме цяла книга. Възможно е, обаче, да се направи сравнително просто **JTable** ако разбирате основата.

JTable управлява изобразяването на данните, а **TableModel** управлява данните. Така че за да се създаде **JTable** типично ще създадете първо **TableModel**. Може просто да приложите **TableModel** интерфейса, но обикновено е по-лесно да се наследи от **AbstractTableModel**:

```
//: c13:Table.java
// Simple demonstration of JTable
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.table.*;
import javax.swing.event.*;

// The TableModel controls all the data:
class DataModel extends AbstractTableModel {
    Object[][] data = {
        {"one", "two", "three", "four"},
        {"five", "six", "seven", "eight"},
        {"nine", "ten", "eleven", "twelve"},
    };
    // Prints data when table changes:
    class TML implements TableModelListener {
        public void tableChanged(TableModelEvent e) {
            for(int i = 0; i < data.length; i++) {
                for(int j = 0; j < data[0].length; j++) {
                    System.out.print(data[i][j] + " ");
                    System.out.println();
                }
            }
        }
    }
    DataModel() {
        addTableModelListener(new TML());
    }
    public int getColumnCount() {
        return data[0].length;
    }
    public int getRowCount() {
        return data.length;
    }
    public Object getValueAt(int row, int col) {
        return data[row][col];
    }
    public void
    setValueAt(Object val, int row, int col) {
        data[row][col] = val;
        // Indicate the change has happened:
        fireTableDataChanged();
    }
    public boolean
    isCellEditable(int row, int col) {
        return true;
    }
};

public class Table extends JPanel {
    public Table() {
        setLayout(new BorderLayout());
        JTable table = new JTable(new DataModel());
```

```

JScrollPane scrollpane =
    JTable.createScrollPaneForTable(table);
add(scrollpane, BorderLayout.CENTER);
}
public static void main(String args[]) {
    Show.inFrame(new Table(), 200, 200);
}
} //:~

```

DataModel съдържа масив от данни, но също може да ги вземете от друг източник като база данни. Конструкторът добавя **TableModelListener** който печата таблицата всеки път когато масивът бъде променен. Останалите методи следват конвенцията за имената на Beans и се използват от **JTable** когато той иска да представи информацията в **DataModel**. **AbstractTableModel** дава методи по подразбиране **setValueAt()** и **isCellEditable()** които не позволяват промени на данните, така че ако искате да можете да промените данните трябва да подтиснете тези методи.

Щом веднъж имате **TableModel**, трябва само да го дадете на конструктора на **JTable**. Всички детайли по изобразяването, редактирането и осъвременяването ще стават автоматично. Забележете, че този промер също слага **Jtable** в **JScrollPane**, което изства специален **JTable** метод.

Панели с ушички

По-рано в тази глава се запознахте с положително средновековния **CardLayout** видяхте как да се справите с управлението на застрашителното разполагане на картите. Някои наистина мислеха че това е добър дизайн. За щастие, Swing оправя това чрез доставянето на **JTabbedPane**, който поддържа всичките ушички, превключване и всичко. Контрастът между **CardLayout** и **JTabbedPane** отнема дъха.

Следният пример е просто радост защото е напредък спрямо останалите до тук. Те са всичките направени с наследяване от **JPanel**, така че този пример ще сложи всичко на правилното му място в **JTabbedPane**. Ще видите че използването на RTTI прави примера много малък и елегантен:

```

//: c13:Tabbed.java
// Using tabbed panes
// <applet code=Tabbed
// width=300 height=500></applet>
import javax.swing.*;
import java.awt.*;
import javax.swing.border.*;

public class Tabbed extends JApplet {
    static Object[] q = {
        {"Felix", Borders.class},
        {"The Professor", Buttons.class},
        {"Rock Bottom", ButtonGroups.class},
        {"Theodore", Faces.class},
        {"Simon", Menus.class},
        {"Alvin", Popup.class},
        {"Tom", ListCombo.class},
        {"Jerry", Progress.class},
        {"Bugs", Trees.class},
        {"Daffy", Table.class},
    };
    static JPanel makePanel(Class c) {

```

```

String title = c.getName();
title = title.substring(
    title.lastIndexOf('.') + 1);
JPanel sp = null;
try {
    sp = (JPanel)c.newInstance();
} catch(Exception e) {
    System.out.println(e);
}
sp.setBorder(new TitledBorder(title));
return sp;
}
public void init() {
    Container cp = getContentPane();
    cp.setLayout(new BorderLayout());
    JTabbedPane tabbed = new JTabbedPane();
    for(int i = 0; i < q.length; i++)
        tabbed.addTab((String)q(i)(0),
            makePanel((Class)q(i)(1)));
    cp.add(tabbed, BorderLayout.CENTER);
    tabbed.setSelectedIndex(q.length/2);
}
} //:~

```

Отново може да видите темата масив използван за конфигурация: първият елемент е **String**ът който трябва да се сложи на ушенцето и вторият е **JPanel** класът който ще се изобразява в съответния панел. В конструктора **Tabbed()** може да видите два важни метода на **JTabbedPane** как са използвани: **addTab()** за слагане на ново ушенце и **setSelectedIndex()** за избиране на ушенце с което да се започне. (Използвано е едно от средата само за да видите, че не е необходимо да започнете от началото.)

Като викате **addTab()** го снабдявате със **String** за ушето и някакъв **Component** (тоест, AWT **Component**, не точно **JComponent**, който е извлечен от AWT **Component**). **Component**ът ще се изобрази на панела. Като направите това вече няма нужда от никакво допълнително управление –**JTabbedPane** прави всичко необходимо (както трябва).

Методът **makePanel()** взима **Class** обект от класа който искате да създадете и използва **newInstance()** да създаде един, каствайки го към **JPanel** (разбира се, това предполага че всеки клас който ще искате да добавите е наследен от **JPanel**, но такава е структурата на примерите в тази секция). Добавя **TitledBorder** който съдържа името на класа и връща резултат като **JPanel** за да бъде използван в **addTab()**.

Като пуснете програмата ще видите че **JtabbedPane** автоматично стекира ушите ако са твърде много за да се поберат в един ред.

Още Swing

Тази секция беше само да ви въведе в мощта на Swing и да ви даде възможност да започнете да видите колко относително лесно е да напипате пътя си в библиотеките. Видяното до тук може би съставя голяма порция от вашите нужди за UI дизайн. Обаче има още много в Swing – тя е замислена да бъде с пълни възможности но UI инструмент за дизайн. Ако не виждате каквото ви трябва тук, гмурнете се в онлайн документацията на Sun и търсете във Web. Вероятно има начин да направите всичко каквото ви хрумне.

Някои от въпросите които не са разгледани в тази секция включват:

- ◆ По-специфични компоненти като **JColorChooser**, **JFileChooser**, **JPasswordField**, **JHTMLPane** (който прави просто HTML форматиране и изобразяване), **JTextPane** (текстов редактор който поддържа форматиране, отрязване на думите и образи). Те са праволинейни за използване.
- ◆ Новите типове събития в Swing. В много отношения приличат на изключения: важен е типът и името може да се използва за намиране на всичко необходимо.
- ◆ Нов управител на разполагането наречен **BoxLayout**.
- ◆ Управление на разцепването: лента в стила на разделител която позволява динамично да манипулирате мястото на другите компоненти.
- ◆ **JLayeredPane** и **JInternalFrame**, използвани заедно за създаване на дъщерни прозорци в родителските, за *multiple-document interface* (MDI) приложения.
- ◆ Закачваеми изглед и интуитивно усещане, така че може да направите една програма която може да се адаптира към различни операционни среди и да изглежда като тях.
- ◆ Собствени курсори.
- ◆ Плаващи, с възможност да се слагат по краищата на екрана ленти с инструменти с **JToolbar** API.
- ◆ Двойно буфериране и Automatic repaint batching за по-гладки пречертавания на екрана.
- ◆ Вградена “undo” поддръжка.
- ◆ Поддръжка на Влач[®] и Пусн[®].

Използване на URLове от аплет

Възможно е аплет да предизвика изобразяване на URL чрез браузъра от който аплетът е пуснат. Може да направите това със следния ред:

```
| getAppletContext().showDocument(u);
```

където **u** е **URL** обект. Ето прост пример който ви пренасочва към друга Web страница. Страницата се случва да е изход от CGI програма, но лесно може да отидете към обикновена HTML страница, така че бихте могли да построите този аплет да бъде защитена с парола врата към някаква част от вашия Web сайт:

```
//: c13>ShowHTML.java
// <applet code>ShowHTML width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;

public class ShowHTML extends JApplet {
    static final String CGIProgram = "MyCGIProgram";
    JButton send = new JButton("Go");
    JLabel l = new JLabel();
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
```

```

send.addActionListener(new AI());
cp.add(send);
cp.add(l);
}
class AI implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
        try {
            // This could be an HTML page instead of
            // a CGI program. Notice that this CGI
            // program doesn't use arguments, but
            // you can add them in the usual way.
            URL u = new URL(
                getDocumentBase(),
                "cgi-bin/" + CGIProgram);
            // Display the output of the URL using
            // the Web browser, as an ordinary page:
            getAppletContext().showDocument(u);
        } catch(Exception e) {
            l.setText(e.toString());
        }
    }
}
} ///:~

```

Красотата на **URL** е в това колко ви закриля. Може да се свържете към Web сървърите без да знаете какво става зад кулисите.

Четене на файл от сървър

Вариация на горната програма чете файл намиращ се на сървър. Файлът се задава от клиента:

```

//: c13:Fetcher.java
// <applet code=Fetcher width=400 height=250>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;

public class Fetcher extends JApplet {
    JButton fetchIt= new JButton("Fetch the Data");
    JTextField f =
        new JTextField("Fetcher.java", 20);
    JTextArea t = new JTextArea(10,40);
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        fetchIt.addActionListener(new FetchL());
        cp.add(t); cp.add(f); cp.add(fetchIt);
    }
    public class FetchL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            try {
                URL url =
                    new URL(getDocumentBase(),f.getText());

```

```

t.setText(url + "\n");
InputStream is = url.openStream();
BufferedReader in = new BufferedReader(
    new InputStreamReader(is));
String line;
while ((line = in.readLine()) != null)
    t.append(line + "\n");
} catch(Exception ex) {
    t.append(ex.toString());
}
}
}
} //://:~
```

Инструмент за оглед на методите

За да завършим главата, ще изградим инструмент с който може да се разглеждат методите на класовете, за подпомагане на програмирането.

Глава 11 въведе концепцията за *рефлексия* на Java 1.1 и използва тази черта за разглеждане методите на даден клас – дали цялото множество или подмножество съгласно маска на имената дадена от вас. Магията е че може да покаже автоматично всичките на клас без да ви кара да ходите нагоре по йерархията гледайки базовия клас на всяко ниво. Така се получава спестяващ време инструмент за програмиране: понеже имената на повечето методи в Java са доста описателни и подробни, може да търсите за имена които съдържат конкретна дума. Когато мислите че сте намерили каквото ви трябва, вижте онлайн документацията.

Обаче в глава 11 не бяхте виждали Swing, така че въпросният инструмент беше развит като приложение за конзолата. Ето по-полезна GUI версия, която динамично обновява изхода и позволява да режете и вмъквате от изхода:

```

//: c13:DisplayMethods.java
// Display the methods of any class inside
// a window. Dynamically narrows your search.
// <applet code = DisplayMethods
// width=650 height=700></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.io.*;

public class DisplayMethods extends JApplet {
    Class cl;
    Method() m;
    Constructor() ctor;
    String() n = new String(0);
    JTextField
        name = new JTextField(40),
        searchFor = new JTextField(30);
    JCheckBox strip =
        new JCheckBox("Strip Qualifiers", true);
    JTextArea results = new JTextArea(40, 65);
```

```

class NameL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        String nm = name.getText().trim();
        if(nm.length() == 0) {
            results.setText("No match");
            n = new String(0);
            return;
        }
        try {
            cl = Class.forName(nm);
        } catch (ClassNotFoundException ex) {
            results.setText("No match");
            return;
        }
        m = cl.getMethods();
        ctor = cl.getConstructors();
        // Convert to an array of Strings:
        n = new String(m.length + ctor.length);
        for(int i = 0; i < m.length; i++)
            n(i) = m(i).toString();
        for(int i = 0; i < ctor.length; i++)
            n(i + m.length) = ctor(i).toString();
        reDisplay();
    }
}
class StripL implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
        reDisplay();
    }
}
class SearchForL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        reDisplay();
    }
}
void reDisplay() {
    // Create the result set:
    String() rs = new String(n.length);
    String find = searchFor.getText();
    int j = 0;
    // Select from the list if find exists:
    for (int i = 0; i < n.length; i++) {
        if(find == null)
            rs(j++) = n(i);
        else if(n(i).indexOf(find) != -1)
            rs(j++) = n(i);
    }
    results.setText("");
    if(strip.isSelected())
        for (int i = 0; i < j; i++)
            results.append(
                StripQualifiers.strip(rs(i)) + "\n");
    else // Leave qualifiers on
        for (int i = 0; i < j; i++)
            results.append(rs(i) + "\n");
}
public void init() {

```

```

name.addActionListener(new NameL());
searchFor.addActionListener(new SearchForL());
strip.addItemListener(new StripL());
JPanel
top = new JPanel(),
lower = new JPanel(),
p = new JPanel(new BorderLayout());
top.add(new JLabel("Qualified class name:"));
top.add(name);
lower.add(
    new JLabel("String to search for:"));
lower.add(searchFor);
lower.add(strip);
p.add(top, BorderLayout.NORTH);
p.add(lower, BorderLayout.SOUTH);
Container cp = getContentPane();
cp.setLayout(new BorderLayout());
cp.add(p, BorderLayout.NORTH);
cp.add(results, BorderLayout.CENTER);
}
public static void main(String[] args) {
    JApplet applet = new DisplayMethods();
    JFrame frame = new JFrame("Display Methods");
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    frame.getContentPane().add(applet);
    frame.setSize(650, 700);
    applet.init();
    applet.start();
    frame.setVisible(true);
}
}

class StripQualifiers {
    private StreamTokenizer st;
    public StripQualifiers(String qualified) {
        st = new StreamTokenizer(
            new StringReader(qualified));
        st.ordinaryChar(' ');
    }
    public String getNext() {
        String s = null;
        try {
            if(st.nextToken() != StreamTokenizer.TT_EOF) {
                switch(st.ttype) {
                    case StreamTokenizer.TT_EOL:
                        s = null;
                        break;
                    case StreamTokenizer.TT_NUMBER:
                        s = Double.toString(st.nval);
                        break;
                    case StreamTokenizer.TT_WORD:

```

```

        s = new String(st.sval);
        break;
    default: // single character in type
        s = String.valueOf((char)st.type);
    }
}
} catch(IOException e) {
    System.out.println(e);
}
return s;
}

public static String strip(String qualified) {
    StripQualifiers sq =
        new StripQualifiers(qualified);
    String s = "", si;
    while((si = sq.getNext()) != null) {
        int lastDot = si.lastIndexOf('.');
        if(lastDot != -1)
            si = si.substring(lastDot + 1);
        s += si;
    }
    return s;
}
} //:~

```

Някои неща сте виждали и преди. Както с много от GUI програмите в тази книга, и тази е създадена да може да бъде и приложение и аплет. Също, класът **StripQualifiers** е точно същия като в глава 11.

GUI съдържа **TextField name** където можете да въведете пълното име на класа за който се интересувате, а друг един, **searchFor**, където можете да въведете текст който да се търси в методите. **Checkbox** позволява да се избере дали да са напълно определени имената в изхода или квалификацията да е отрязана. Накрая, резултатите са изобразени в **TextArea**.

Ще видите че няма бутони с които да започвате търсенето. Така е защото и **TextField**овете и **Checkbox** се наблюдават от техните слушателски обекти. Щом направите промяна, автоматично се осъвременява. Ако смените текста в полето **name** новият текст се хваща в **class NameL**. Ако текстът не е празен, използва се в **Class.forName()** за опит за разглеждане на класа. Докато въвеждате, разбира се, името ще е незавършено и **Class.forName()** ще се проваля, което значи че изхвърля изключение. Това се хваща и **TextArea** се слага "No match". Но щом напишете правилно име (смята се и капитализацията), **Class.forName()** е успешен и **getMethods()** и **getConstructors()** ще върнат масиви от **Method** и **Constructor** обекти, респективно. Всеки от обектите в тези масиви се превръща в **String** чрез **toString()** (това дава завършен метод за сигнатурата на конструктора) и двата списъка се комбинират в **n**, един **String** масив. Масивът **n** е член на **class DisplayMethods** и се използва за осъвременяване на екрана винаги когато се вика **reDisplay()**.

Ако измените **Checkbox** или **searchFor** компонентите техните слушатели просто викат **reDisplay()**. **reDisplay()** създава временен масив от **String** наречен **rs** (от "result set"). Резултантата или се копира директно от **n** ако няма **find** дума, или условно се копира от **String**овете в **n** които съдържат **find** думата. Накрая **strip Checkbox** се пита да се визи дали потребителят иска да се махне квалификацията (по подразбиране "да"). Ако да, **StripQualifiers.strip()** го прави; ако не, списъкът просто се изобразява.

В **init()** може да помислите че има много работа за установяване на разположението. Фактически компонентите биха могли да се разположат и с по-малко работа, но напредъкът от използването на **BorderLayout** по този начин е че се позволява на потребителя да мени

размерите на прозореца и – в частност – да прави **TextArea** по-голям, което значи че може да нагласите да виждате имената и без скролинг.

Може да намерите за удобно да държите инструмента работещ докато програмирате, понеже той дава един от най-добрите “фронтални атаки” когато се опитвате да определите кой метод да се вика.

Резюме

От всичките библиотеки в Java, GUI библиотеката е претърпяла най-драбатични промени от Java 1.0 към Java 2. AWT на Java 1.0 бе всеобщо критикуван като най-лошият виждан дизайн, и докато би позволил да се създадат преносими програми, резултантният GUI беше “еднакво посредствен на всички платформи.” Той беше също ограничаващ, тромав, неприятен за използване в сравнение с естествените за всяка платформа езици и развойни среди.

Когато Java 1.1 въведе новия събитиен модел и Java Beans, писесата беше поставена – сега беше възможно да се създадат GUI компоненти които лесно да можеше да се влачат и пускат във визуалните среди за разработка. Освен това дизайнът на събитийния модел и Beans ясно показва строго изискване за леснота на програмирането и пристапа поддръжка (Нещо което не беше очевидно в AWT на 1.0). Но това не стана докато GUI компонентите – JFC/Swing класовете – не се появиха и тогава работата беше свършена. Със Swing компонентите междуплатформеното GUI програмиране може да бъде цивилизована практика.

Фактически единственото нещо което липсва е развойна среда и там лежи истинска революция. Visual Basic и Visual C++ на Microsoft изискват техните развойни среди, както и Delphi и C++ Builder на Borland. Ако искате развойната среда да стане по-добра, трябва да кръстосате пръсти и да се надявате че доставчиците ще дадат каквото искате. Но Java е отворена среда, така че не само дава възможност за съревнование между средите за разработка, тя го окуражава. И за да бъдат приемани сериозно тези инструменти, те трябва да поддържат Java Beans. Това значи игрално поле с нива: ако излезе по-добра развойна среда, не сте вързани към тази която използвате – може да изберете и да започнете с новата като си подобрите продуктивността. Този вид състезателна среда в GUI развойните среди не е виждана преди, а получилото се състезание може да породи само добри резултати за продуктивността на програмиста.

Упражнения

- Създайте аплет с текстово поле и три бутона. Като натиснете всеки бутон нека се паявява различен текст в текстовото поле.
- Добавете отметка към упр. 1, хванете събитието, вмъкнете различен текст в полето.
- Създайте аплет и добавете компоненти които предизвикват викането на **action()**, после хванете събитията им и изведете подходящо съобщение за всяко в текстово поле.
- Добавете към упражнение 3 компонента който може да се използва само със събитията на **handleEvent()**. Подтиснете **handleEvent()** и изобразете подходящи съобщения за всяко текстово поле.
- Създайте аплет с **Button** и **TextField**. Напишете **handleEvent()** така че ако бутона има фокуса, знаците въведени върху него да се появят в **TextField**.

6. Създайте приложение и поставете всички компоненти споменати в тази глава в главната рамка, включително меню и диалогова кутия.
7. Променете **TextNew.java** така че значите в **t2** да запазят оригиналния си регистър както са набрани, вместо да се правят в горен регистър.
8. Променете **CardLayout1.java** така че да използва Java 1.1 събитийния модел.
9. Добавете **Frog.class** към манифестния файл в тази глава и пуснете **jar** да създаде JAR файл съдържащ **Frog** и **BangBean**. Сега свалете/инсталрайте BDK от Sun или използвайте ваша собствена развойна среда работеща с Beans и добавете JAR файла в средата така че да тествате двета Beans.
10. Създайте ваш собствен Java Bean наречен **Valve** който има две характеристики: Boolean наречена "on" и цяла наречена "level." Създайте манифест файл, използвайте **jar** за пакетиране на вашия Bean, после го натоварете в бийнбокса или във ваша развойна среда работеща с Beans така, че да може да го тествате.
11. (Предизвикателство) Променете **Menus.java** така, че да поддържа каскадирани менюта.

14: Много нишки

Обектите дават начин да се раздели програмата на независими секции. Често е необходимо освен това програмата да се раздели на отзелни, независимо работещи подзадачи.

Всяка от тези независимо работещи подзадачи се нариче *thread*, програмилате сякаш всяка нишка е самостоятелна и има CPU само за нея. Някакъв по-дълбок механизъм фактически разделя времето на CPU, но изобщо не е необходимо да мислите за това, та програмирането с много нишки става много по-лесно.

Тук са полезни някои дефиниции. *process* е самостоятелна работеща програма с отделно адресно пространство. ОС с *multitasking* е в състояние да пусне повече от един процес (програма) едновременно, като създава впечатлението че всеки работи отделно, разпределяйки CPU циклите между процесите. Нишкото е един последователен поток, течене на програмата в рамките на процес. Един процес може да има много конкурентно работещи нишки.

Има много възможни сценарии с многонишковост, но изобщо имате част от вашата програма вързана с някакъв ресурс или събитие, а не искате да виси цялата програма заради недостиг на ресурса. Така че създавате нишка свързана със събитието или ресурса и я пускате независимо от главната програма да си работи. Добър пример е "quit" бутон – не бихте желали навсякъде да проверявате дали е натиснат бутона и все пак желаете когато се натисне да има ефект, като чели е проверяван постоянно. Фактически един от най-привлекателните аспекти в който се използва многонишковост е създаването на потребителски интерфейс който реагира.

Отговарящ потребителски интерфейс

Като начало да предположим че програмата изпълнява някаква интензивно използваща CPU процедура и като резултат няма отговор на потребителския вход, той се игнорира. Тази, като аплет/приложение, просто ще изобразява показанието на работещ броящ:

```
//: c14:Counter1.java
// A non-responsive user interface
// <applet code=Counter1 width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class Counter1 extends JApplet {
    private int count = 0;
    private JButton
        onOff = new JButton("Toggle"),
        start = new JButton("Start");
    private JTextField t = new JTextField(10);
    private boolean runFlag = true;
    public void init() {
```

```

Container cp = getContentPane();
cp.setLayout(new FlowLayout());
cp.add(t);
start.addActionListener(new StartL());
cp.add(start);
onOff.addActionListener(new OnOffL());
cp.add(onOff);
}
public void go() {
    while (true) {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {}
        if (runFlag)
            t.setText(Integer.toString(count++));
    }
}
class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        go();
    }
}
class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        runFlag = !runFlag;
    }
}
public static void main(String[] args) {
    JApplet applet = new Counter1();
    JFrame frame = new JFrame("Counter1");
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    frame.getContentPane().add(applet);
    frame.setSize(300, 100);
    applet.init();
    applet.start();
    frame.setVisible(true);
}
} //:~

```

Вече Swing кода и този за аплета трябва да са достатъчно познати от глава 13. Методът `go()` е мястото където програмата действа най-много: той слага текущата стойност от `count` в `TextField t`, после инкрементира `count`.

Част от безкрайния цикъл в `go()` е викането на `sleep()`. `sleep()` трябва да бъде асоцииран с `Thread` обект, излиза че всяко приложение има някаква нишка асоциирана с нея. (Разбира се, Java е основан на нишките и винаги има работещи такива.) Така че без значение дали явно използвате нишки, може да направите нишки чрез използване на `Thread` и статичния метод `sleep()`.

Забележете че `sleep()` може да изхвърли `InterruptedException`, макар че изхвърлянето на изключение се смята за неприемлив начин за напускане на нишката и трябва да се избягва.

(Още веднъж, изключениета са за изключителни ситуации, не за нормалното тече на програмата.) Прекъсване на спяща нишка ще бъде включено в следваща версия на езика.

Когато се натисне бутона **start** вика се **go()**. Като разгледате **go()** може наивно да си мислите (като мен) че той позволява многонишковост понеже отива да спи. Тоест, докато нишката спи CPU-то сякаш внимателно следи натисканията на други. Но излиза че леалният проблем е дето **go()** никога не завършва, понеже е безкраен цикъл, а така **actionPerformed()** никога не завършва. Понеже сте ограничени в **actionPerformed()** за първото натискане на клавиш, програмата не може да обработи никакви други събития. (За да излезете, трябва да убиете процеса някакси; най-лесно с Control-C в конзолния прозорец.)

Основният проблем е че **go()** трябва да продължи действието си, а в същото време трябва да завърши така че **actionPerformed()** да завърши и да може да продължи обработката на потребителския вход. Но в обикновен метод като **go()** не може да се продължи и едновременно да се даде управлението на останалата част от програмата. Това звучи като да трябва да се изпълни неизпълнимото, понеже CPU трябва да бъде на две места едновременно, но това е точно илюзията която създава многонишковостта. Моделът на многонишковост (и програмната поддръжка в Java) е програмиско облекчение да се вършат няколко работи квазиедновременно. С нишките CPU ще дава на нишките на всяка нейното време. Всяка нишка има достатъчно съзнателност колко да държи CPU за себе си, но времето на CPU фактически бива разпределено между нишките.

Многонишковостта намалява продуктивността на системата някакси, но чистият напредък в преоектирането на програмите, баланса на ресурсите и удобството са твърде ценни. Разбира се, ако имате повече от едно CPU ОС може да даде на всяко CPU да работи няколко нишки или даже една нишка и цялата програма ще работи много по-бързо. Мултитаскингът и многонишковостта (мултитрединг) са види се най-добрите начини за използване на многопроцесорните системи.

Наследяване от Thread

Най-простия начин да се създава нишка е да се наследи от **Thread**, който има всичко необходимо да създава и стартира нишки. За **Thread** най-важният метод е **run()**, който обикновено подтискате за да направите нишката по свой вкус. Така **run()** е кодът който ще се изпълнява "едновременно" с другите нишки в програмата.

Следният пример създава много нишки и слади броя им, като присвоява на всяка нишка номер, генериран със **static** променлива. Методът **run()** на **Thread** е подтиснат за да отброява когато прави нова нишка чрез цикъла си и да спира когато броячът се нулира (в точката в която **run()** завършва нишката се прекратява).

```
//: c14:SimpleThread.java
// Very simple Threading example

public class SimpleThread extends Thread {
    private int countDown = 5;
    private static int threadCount = 0;
    private int threadNumber = ++threadCount;
    public SimpleThread() {
        System.out.println("Making " + threadNumber);
    }
    public void run() {
        while(true) {
            System.out.println("Thread " +
                threadNumber + "(" + countDown + ")");
            if(--countDown == 0) return;
        }
    }
}
```

```

}
public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
        new SimpleThread().start();
    System.out.println("All Threads Started");
}
} //:~

```

Методът **run()** практически винаги има някакъв вид цикъл който продължава докато нишката вече не е нужна, така че трябва да зададете условието при което ще се прекъсне цикъла (или, в горния случай, просто **return** от **run()**). Често **run()** е с формата на безконечен цикъл, което значи че без някакво външно извикване на **stop()** или **destroy()** за нея нишка тя ще продължава вечно (докато завърши програмата).

В **main()** може да видите няколко нишки да се създават и стартират. Специалният метод който идва с **Thread** класа е **start()**, който изпълнява специална инициализация и после вика **run()**. Така стъпките са: конструкторът се вика да построи обекта, **start()** конфигурира нишката и вика **run()**. Ако не викате **start()** (което може да направите в конструктора, ако е уместно) нишкото никога да бъде стартирана.

Един изход от тази програма (той ще бъде различен всеки път) е:

```

Making 1
Making 2
Making 3
Making 4
Making 5
Thread 1(5)
Thread 1(4)
Thread 1(3)
Thread 1(2)
Thread 2(5)
Thread 2(4)
Thread 2(3)
Thread 2(2)
Thread 2(1)
Thread 1(1)
All Threads Started
Thread 3(5)
Thread 4(5)
Thread 4(4)
Thread 4(3)
Thread 4(2)
Thread 4(1)
Thread 5(5)
Thread 5(4)
Thread 5(3)
Thread 5(2)
Thread 5(1)
Thread 3(4)
Thread 3(3)
Thread 3(2)
Thread 3(1)

```

Ще забележите че в този пример никъде не се вика **sleep()** и все пак изходът свидетелства че всяка нишка взима част от времето на CPUза да работи. Това показва че **sleep()**, макар и да се основава на нишките за да може да се изпълни, няма отношение към задействането или отмяната на многонишковостта. Той е просто още един метод.

Може също да видите че нишките не се пускат в реда в който са създанети. Фактически редът в който CPU посещава съществуващите нишки е недетерминиран (по-скоро псевдослучаен-б.пр.), поне докато не нагласите приоритетите чрез метода **setPriority()** на **Thread**.

Когато **main()** създава **Thread** обекти тя не хваща никакви манипулатори за тях. Обикновен обект би бил веднага плячка на боклучаря, но не и **Thread**. Всяка **Thread** се "регистрира" така че има позоваване за нея някъде и не може да бъде "очистена" ..

Нишковост за отговарящ интерфейс

Сега е възможно да решим проблема в **Counter1.java** с нишка. Трикът е да се сложи подзадачата – тоест, цикълът в **go()** – вътре в **run()** метод на нишка. Когато потребителят натисне бутона **start** нишката се стартира, но тогава създаването на нишката завършва, така че макар и нишката да се изпълнява, главната програма (следене на потребителския вход и реагиране) може да продължи. Ето решението:

```
//: c14:Counter2.java
// A responsive user interface with threads
// <applet code=Counter2 width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class SeparateSubTask extends Thread {
    private int count = 0;
    private Counter2 c2;
    private boolean runFlag = true;
    public SeparateSubTask(Counter2 c2) {
        this.c2 = c2;
        start();
    }
    public void invertFlag() { runFlag = !runFlag; }
    public void run() {
        while (true) {
            try {
                sleep(100);
            } catch (InterruptedException e){}
            if(runFlag)
                c2.t.setText(Integer.toString(count++));
        }
    }
}

public class Counter2 extends JApplet {
    JTextField t = new JTextField(10);
    private SeparateSubTask sp = null;
    private JButton
        onOff = new JButton("Toggle"),
        start = new JButton("Start");
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(t);
        start.addActionListener(new StartL());
        cp.add(start);
    }
}
```

```

onOff.addActionListener(new OnOffL());
cp.add(onOff);
}
class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        if(sp == null)
            sp = new SeparateSubTask(Counter2.this);
    }
}
class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        if(sp != null)
            sp.invertFlag();
    }
}
public static void main(String[] args) {
    JApplet applet = new Counter2();
    JFrame frame = new JFrame("Counter2");
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    frame.getContentPane().add(applet);
    frame.setSize(300, 100);
    applet.init();
    applet.start();
    frame.setVisible(true);
}
} //:~

```

Counter2 е сега праволинейна програма, чиято работа е само да установи и да се занимава с потребителския интерфейс. Но сега, когато потребителят натисне бутона **start**, не се вика метод. Вместо това се създава нишка от клас **SeparateSubTask** (конструкторът я стартира, в този случай), а после събитийния цикъл на **Counter2** продължава. Забележете че манипулатора към **SeparateSubTask** се запомня така че когато натиснете **onOff** той може да превключва **runFlag**-а вътре в обекта **SeparateSubTask**. Тази нишка (когато гледа флагата) може да се стартира и спира сама. (Това също може да бъде осигурено ако се направи **SeparateSubTask** вътрешен клас.)

Класът **SeparateSubTask** е просто разширение на **Thread** с конструктор (който запомня манипулатора **Counter2** и после пуска нишката чрез викане на **start()**) и **run()** който основно съдържа кода от вътрешността на **go()** в **Counter1.java**. Понеже **SeparateSubTask** знае че държи манипулятор към **Counter2**, може да го намери и да достигне **TextField** на **Counter2** когато е необходимо.

Като натиснете бутона **onOff** ще видите почти мигновен отговор. Разбира се отговорът не е в действителност моментален, не като в система управлявана от прекъсвания. Броящът спира само когато нишката владее CPU-то и забелязва че флагът е сменен.

Подобряване на кода с вътрешен клас

Между другото, погледнете свързването което се получава между класовете **SeparateSubTask** и **Counter2**. **SeparateSubTask** е тясно свързан с **Counter2** – той трябва да държи манипулятор към “родителя”си обекта **Counter2** така че да може да го вика и манипулира. И вси пак двата класа не са за обединяване в един (макар че в следващата секция ще видите че Java дава

начин да бъдат комбинирани) понеже правят различни неща и се създават по различно време. Те са тясно свързани (което аз наричам "куплет") и това прави кодирането тромаво. Това е ситуация където вътрешен клас може да подобри съществено кода:

```
//: c14:Counter2i.java
// Counter2 using an inner class for the thread
// <applet code=Counter2i width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Counter2i extends JApplet {
    private class SeparateSubTask extends Thread {
        int count = 0;
        boolean runFlag = true;
        SeparateSubTask() { start(); }
        public void run() {
            while (true) {
                try {
                    sleep(100);
                } catch (InterruptedException e){}
                if(runFlag)
                    t.setText(Integer.toString(count++));
            }
        }
    }
    private SeparateSubTask sp = null;
    private JTextField t = new JTextField(10);
    private JButton
        onOff = new JButton("Toggle"),
        start = new JButton("Start");
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(t);
        start.addActionListener(new StartL());
        cp.add(start);
        onOff.addActionListener(new OnOffL());
        cp.add(onOff);
    }
    class StartL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            if(sp == null)
                sp = new SeparateSubTask();
        }
    }
    class OnOffL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            if(sp != null)
                sp.runFlag = !sp.runFlag; // invertFlag();
        }
    }
    public static void main(String[] args) {
        JApplet applet = new Counter2i();
        JFrame frame = new JFrame("Counter2i");
        frame.addWindowListener(
```

```

new WindowAdapter() {
    public void windowClosing(WindowEvent e){
        System.exit(0);
    }
});
frame.getContentPane().add(applet);
frame.setSize(300, 100);
applet.init();
applet.start();
frame.setVisible(true);
}
} //:~

```

Името **SeparateSubTask** няма да си пречи с **SeparateSubTask** в предния пример макар и да са в една директория, понеже е скрит като вътрешен клас. Може също да видите че вътрешният клас е **private**, което значи че на неговите членове и методи може да се даде достъпът по подразбиране (освен **run()**, който трябва да е **public** понеже е **public** в базовия клас). Вътрешният **private** клас не е достъпен за нищо освен за **Counter2i**, а понеже са тясно свързани добре е да се охлабят пречките за достъп помежду им. В **SeparateSubTask** може да видите че **invertFlag()** е мащнат понеже **Counter2i** сега има пряко достъп до **runFlag**.

Забележете също че конструктора на **SeparateSubTask** е опростен – сега само стартира нишката. Манипуляторът към обекта **Counter2i** се хваща както в предишната версия, но заместо това да се прави ръчно и да се правят ръчни обръщания към другия обект, механизъмът на вътрешните класове се грижи за това автоматично. В **run()** може да видите че **t** се достига просто, като да беше поле на **SeparateSubTask**. Полето **t** в родителския клас сега може да бъде направено **private** понеже **SeparateSubTask** може да го достигне без никакво специално разрешение – и винаги е добре да се правят полетата “толкова **private** колкото е възможно” така че да не могат нежелано да бъдат променени от сили извън вашия клас.

Винаги когато се получат тясно свързани класове прегледайте възможността да бъдат използвани вътрешни класове.

Комбиниране на нишката с главния клас

В горния пример може да се види че класът на нишката е отделен от главния програмен клас. Това има смисъл и е лесно за разбиране. Има обаче друга форма която не е така проста но която често се използва и е по-подходяща (което вероятно съдейства за нейната популярност). Тази форма комбинира главния програмен клас с класа на нишката като прави главния клас нишка. Понеже за GUI програма главния клас трябва да бъде наследен или от **Frame** или от **Applet**, трябва да се използва интерфейс за да се добави функционалност. Този интерфейс се нарича **Runnable** и съдържа същия основен метод като **Thread**. Фактически **Thread** също прилага **Runnable**, който специфицира само да има **run()** метод.

Използването на комбинирана програма/нишка не е така очевидно. Когато стартирате програмата, създавате обект който е **Runnable**, но не стартирате нишката. Това трябва да се направи явно. Ези неща може да се видят в следната програма, съюто повтаря функционалността на **Counter2**:

```

//:c14:Counter3.java
// Using the Runnable interface to turn the
// main class into a thread.
// <applet code=Counter3 width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

```

```

public class Counter3
    extends JApplet implements Runnable {
private int count = 0;
private boolean runFlag = true;
private Thread selfThread = null;
private JButton
    onOff = new JButton("Toggle"),
    start = new JButton("Start");
private JTextField t = new JTextField(10);
public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    start.addActionListener(new StartL());
    cp.add(start);
    onOff.addActionListener(new OnOffL());
    cp.add(onOff);
}
public void run() {
    while (true) {
        try {
            selfThread.sleep(100);
        } catch (InterruptedException e){}
        if(runFlag)
            t.setText(Integer.toString(count++));
    }
}
class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        if(selfThread == null) {
            selfThread = new Thread(Counter3.this);
            selfThread.start();
        }
    }
}
class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        runFlag = !runFlag;
    }
}
public static void main(String[] args) {
    JApplet applet = new Counter3();
    JFrame frame = new JFrame("Counter3");
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    frame.getContentPane().add(applet);
    frame.setSize(300, 100);
    applet.init();
    applet.start();
    frame.setVisible(true);
}
} ///:~

```

Сега `run()` е вътре в класа, но все още е бездеен докато завърши `init()`. Като натиснете бутона `start` се създава нишката (ако вече не съществува) в някакси тъмния израз:

```
| new Thread(Counter3.this);
```

Когато нещо има **Runnable** това просто значи че нещото има `run()` метод и нищо специално повече – това не дава вродени способности за многонишковост, като онези от клас наследен от **Thread**. Така за да се направи нишка от **Runnable** трябва да създадете нишката отделно и да ⁹ дадете **Runnable** обект; има специален конструктор който взема **Runnable** аргумент. После може да извикате `start()` за нея нишка:

```
| selfThread.start();
```

Това прави обичайната инициализация и после вика `run()`.

Удобното на **Runnable interface** е че всичко е в един клас. Ако витрябва достъп до нещо, просто го имате без да се разкарвате по други обекти. Цена обаче има – може да имате само една нишка за него конкретен обект (макар и да можете да създадете повече обекти от този тип или пък да създадете други нишки в различни от този класове).

Забележете че причината не е в **Runnable** интерфейса. Комбинацията от **Runnable** и вашият главен клас го прави, понеже може да имате само един обект от вашия главен клас в едно приложение (програма).

Правене на много нишки

Да видим създаването на много различни нишки. Не може да го направите с предишния пример, така че ще трябва да имате няколко класа наследени от **Thread** за капсулиране на `run()`. Но това е по-общото решение и е по-лесно за разбиране, та докато предишният пример показва стил на програмиране който често ще срещате, аз не мога да го препоръчам за повечето случаи понеже е много малко по-прост и е малко смущаващ.

Следният пример повтаря предишните примери с броячи и бутони. Но сега всичката информация за конкретен брояч, включително бутона и текстовото поле, е вътре в собствен обект наследен от **Thread**. Всичките полета в **Ticker** са **private**, което значи че реализацията на **Ticker** може да бъде променена при желание, включително типа и количеството на елементите за събиране и извеждане на информацията. Когато се създава **Ticker** обект конструкторът иска манипулатор на AWT **Container**, който **Ticker** попълва със своите визуални компоненти. По този начин, ако промените визуалните компоненти, кодът който използва **Ticker** не е необходимо да се променя.

```
//: c14:Counter4.java
// By separating your thread from the main class,
// you can have as many threads as you want.
// <applet code=Counter4 width=600 height=600>
// <param name=size value="20"></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class Ticker extends Thread {
    private JButton b = new JButton("Toggle");
    private JTextField t = new JTextField(10);
    private int count = 0;
    private boolean runFlag = true;
    public Ticker(Container c) {
        b.addActionListener(new ToggleL());
```

```

JPanel p = new JPanel();
p.add(t);
p.add(b);
c.add(p);
}
class ToggleL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        runFlag = !runFlag;
    }
}
public void run() {
    while (true) {
        if (runFlag)
            t.setText(Integer.toString(count++));
        try {
            sleep(100);
        } catch (InterruptedException e) {}
    }
}
}

public class Counter4 extends JApplet {
    private JButton start = new JButton("Start");
    private boolean started = false;
    private Ticker[] s;
    private boolean isApplet = true;
    private int size;
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        // Get parameter "size" from Web page:
        if (isApplet)
            size =
                Integer.parseInt(getParameter("size"));
        s = new Ticker[size];
        for (int i = 0; i < s.length; i++)
            s[i] = new Ticker(cp);
        start.addActionListener(new StartL());
        cp.add(start);
    }
    class StartL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            if(!started) {
                started = true;
                for (int i = 0; i < s.length; i++)
                    s[i].start();
            }
        }
    }
    public static void main(String[] args) {
        Counter4 applet = new Counter4();
        // This isn't an applet, so set the flag and
        // produce the parameter values from args:
        applet.isApplet = false;
        applet.size =
            (args.length == 0 ? 5 :
             Integer.parseInt(args[0]));
    }
}

```

```

JFrame frame = new JFrame("Counter4");
frame.addWindowListener(
    new WindowAdapter() {
        public void windowClosing(WindowEvent e){
            System.exit(0);
        }
    });
frame.getContentPane().add(applet);
frame.setSize(200, applet.size * 50);
applet.init();
applet.start();
frame.setVisible(true);
}
} //:~

```

Ticker съдържа не само оборудването за нишката но и начин да управлява и изобразява нишката. Може да създадете колкоти си искате нишки без явно да задавате прозоречни компоненти.

В **Counter4** Има масив от **Ticker** обекти наречен **s**. За максимална гъвкавост размерът се определя чрез достъп до Web страницата чрез параметрите на аплета. Ето как изглежда параметърът с дължината на страницата, вграден в описанието на аплета:

```

<applet code=Counter4 width=600 height=600>
<param name=size value="20">
</applet>

```

param, **name** и **value** са всичките ключови думи на Web страницата. **name** е с което споменавате страницата във вашата програма, **value** може да бъде всякакъв стринг, не непременно нещо което резутира в число.

Ще видите че определянето на дължината на масива **s** е направено в **init()**, не като част от инлайн дефиницията на **s**. Тоест, не може като част от дефиницията на клас (извън метод):

```

int size = Integer.parseInt(getParameter("size"));
Ticker() s = new Ticker(size);

```

Това може да се компилира, но ще получите странно изключение заради нулев указател по време на изпълнение. Това работи чудесно ако преместите инициализацията **getParameter()** вътре в **init()**. Аплетската рамка взима параметрите и прави нужната подготовка преди да влезе в **init()**.

Освен това този код е пригответ да бъде или аплет или приложение. Когато е приложение аргументът **size** се извлича от командния ред (или се използва стойност по подразбиране).

Щом е определена дължината на масива, създават се нови **Ticker** обекти; като част от конструктора на **Ticker** се добавя бутон и текстово поле към аплета за **Ticker**.

Натискането на бутона **start** значи циклене през целия масив от **Ticker** и викане на **start()** за всеки. Помните, **start()** изпълнява необходима инициализация на нишката и после вика **run()** за нея нишка.

Слушателят **ToggleL** просто обръща флага в **Ticker** и когато съответната нишка си вземе бележка той може да реагира съответно.

Една ценност на този пример е че дава възможност да се създават големи множества от независими подзадачи и да се наблюдава тяхното поведение. В този случай ще видите че с увеличаване броя на подзадачите вашата машина ще показва по-голяма разходимост на числата поради начина по който се обслужват нишките.

Може също да експериментирате колко важен е `sleep(100)` в `Ticker.run()`. Ако го махнете `sleep()`, нещата ще вървят добре докато натиснете превключващ бутон. Тогава тази конкретна нишка има `false` `runFlag` и `run()` е просто въвлечен в безкраен цикъл, който май че е трудно да се прекъсне по време на мултитрединга, така че производителността и реактивността на системата се скапват.

Нишки-демони

Нишката е "демон" когато се предполага тя да извършва обща услуга във фонов режим доколкото програмата работи, но не е основна част от програмата. Така, когато завършат всички нишки не-демони и програмата завършва. Обратно, ако има несвършили нишки не-демони програмата продължава. (Например нишка която е свързана с `main()`.)

Може да намерите дали нишката е демон чрез извикване на `isDaemon()` и може да включвате и изключвате демоничността на нишката чрез `setDaemon()`. Ако нишка е демон всички нишки които тя създава са демони.

Следния пример демонстрира нишки-демони:

```
//: c14:Daemons.java
// Daemonic behavior
import java.io.*;

class Daemon extends Thread {
    private static final int SIZE = 10;
    private Thread[] t = new Thread[SIZE];
    public Daemon() {
        setDaemon(true);
        start();
    }
    public void run() {
        for(int i = 0; i < SIZE; i++)
            t[i] = new DaemonSpawn(i);
        for(int i = 0; i < SIZE; i++)
            System.out.println(
                "t(" + i + ").isDaemon() = "
                + t[i].isDaemon());
        while(true)
            yield();
    }
}

class DaemonSpawn extends Thread {
    public DaemonSpawn(int i) {
        System.out.println(
            "DaemonSpawn " + i + " started");
        start();
    }
    public void run() {
        while(true)
            yield();
    }
}

public class Daemons {
    public static void main(String[] args) {
        Thread d = new Daemon();
```

```

System.out.println(
    "d.isDaemon() = " + d.isDaemon());
// Allow the daemon threads to finish
// their startup processes:
BufferedReader stdin =
    new BufferedReader(
        new InputStreamReader(System.in));
System.out.println("Waiting for CR");
try {
    stdin.readLine();
} catch(IOException e) {}
}
} //:~

```

Нишката **Daemon** слага флага си за демон “true” и после поражда други нишки за да покаже че са демони. После влиза в безкраен цикъл който вика **yield()** за да даде управлението на другите процеси. В по-раншни версии на тази програма безкрайните цикли инкрементираха **int** броячи, но това изглежда довеждаше цялата програма до спиране. Използването **yield()** прави програмата доста енергична.

Нищо не спира програмата да завърши щом **main()** свърши своята работа понеже няма нищо друго освен демони тогава. Така че може да видите резултата от пускането на всички демонски нишки, **System.in** е готово да чете така че чака знака за край на ред. Без това виждате само някои резултати от нишките-демони. (Опитайте заместване на **readLine()** кода с извиквания на **sleep()** с различна дължина за да видите поведението.)

Споделяне на ограничени ресурси

Може да си представим еднонишковата програма като нещо което се движи през вашето проблемно пространство правейки едно нещо едновременно. Понеже има една същност никога не става нужда да се мисли за възможността да трябва да се осигури достъп до нещо от повече места едновременно, като двама души опитващи се да паркират на едно място, да минат през врата едновременно, или даже да говорят едновременно.

Чрез мулитрединга нещата не са самотни вече, но сега имате възможността две нишки да се опитат едновременно да използват един и същ ограницен ресурс. Колизиите трябва да се предотвратят или ще имате две нишки които искат да работят с една банковска сметка едновременно, да печатат на един принтер, или да нагласят един и същ клапан и т.н.

Неправилно ползване на ресурси

Да видим вариация на броячите които бяха използвани до тук в тази глава. В следния пример всяка нишка има два брояча които се инкрементират и изобразяват в **run()**. Освен това има друга нишка от клас **Watcher** която следи броячите да види дали са винаги равни. Това изглежда като ненужна активност, понеже като се гледа кода изглежда очевидно че броячите винаги ще са равни. Но тук идва изненадата. Ето първата версия на програмата:

```

//: c14:Sharing1.java
// Problems with resource sharing while threading
// <applet code=Sharing1 width=650 height=500>
// <param name=size value="20">
// <param name=observers value="1">
// </applet>

```

```

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

class TwoCounter extends Thread {
    private boolean started = false;
    private JTextField t1 = new JTextField(5),
    t2 = new JTextField(5);
    private JLabel l =
        new JLabel("count1 == count2");
    private int count1 = 0, count2 = 0;
    // Add the display components as a panel
    // to the given container:
    public TwoCounter(Container c) {
        JPanel p = new JPanel();
        p.add(t1);
        p.add(t2);
        p.add(l);
        c.add(p);
    }
    public void start() {
        if(!started) {
            started = true;
            super.start();
        }
    }
    public void run() {
        while (true) {
            t1.setText(Integer.toString(count1++));
            t2.setText(Integer.toString(count2++));
            try {
                sleep(500);
            } catch (InterruptedException e){}
        }
    }
    public void synchTest() {
        Sharing1.incrementAccess();
        if(count1 != count2)
            l.setText("Unsynched");
    }
}

class Watcher extends Thread {
    private Sharing1 p;
    public Watcher(Sharing1 p) {
        this.p = p;
        start();
    }
    public void run() {
        while(true) {
            for(int i = 0; i < p.s.length; i++)
                p.s(i).synchTest();
            try {
                sleep(500);
            } catch (InterruptedException e){}
        }
    }
}

```

```

    }

}

public class Sharing1 extends JApplet {
    TwoCounter[] s;
    private static int accessCount = 0;
    private static JTextField aCount =
        new JTextField("0", 7);
    public static void incrementAccess() {
        accessCount++;
        aCount.setText(Integer.toString(accessCount));
    }
    private JButton
        start = new JButton("Start"),
        observer = new JButton("Observe");
    private boolean isApplet = true;
    private int numCounters = 0;
    private int numObservers = 0;
    public void init() {
        if(isApplet) {
            numCounters =
                Integer.parseInt(getParameter("size"));
            numObservers =
                Integer.parseInt(
                    getParameter("observers"));
        }
        s = new TwoCounter[numCounters];
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        for(int i = 0; i < s.length; i++)
            s[i] = new TwoCounter(cp);
        JPanel p = new JPanel();
        start.addActionListener(new StartL());
        p.add(start);
        observer.addActionListener(new ObserverL());
        p.add(observer);
        p.add(new JLabel("Access Count"));
        p.add(aCount);
        cp.add(p);
    }
    class StartL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            for(int i = 0; i < s.length; i++)
                s[i].start();
        }
    }
    class ObserverL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            for(int i = 0; i < numObservers; i++)
                new Watcher(Sharing1.this);
        }
    }
    public static void main(String[] args) {
        Sharing1 applet = new Sharing1();
        // This isn't an applet, so set the flag and
        // produce the parameter values from args:
        applet.isApplet = false;
    }
}

```

```

applet.numCounters =
    (args.length == 0 ? 5 :
     Integer.parseInt(args[0]));
applet.numObservers =
    (args.length < 2 ? 5 :
     Integer.parseInt(args[1]));
JFrame frame = new JFrame("Sharing1");
frame.addWindowListener(
    new WindowAdapter() {
        public void windowClosing(WindowEvent e){
            System.exit(0);
        }
    });
frame.getContentPane().add(applet);
frame.setSize(350, applet.numCounters * 50);
applet.init();
applet.start();
frame.setVisible(true);
}
} //:~

```

Както преди всеки брояч има своите компоненти за изобразяване: две текстови полета и етикет който първоначално показва че браячите са равни. Тези компоненти са добавени към **Container** в конструктора на **TwoCounter**. Понеже тази нишка е стартирана чрез натискане на бутона от потребителя, възможно е **start()** да бъде викан повече от веднъж. Не е редно **Thread.start()** да се вика повече от веднъж за нишка (изхвърля се изключение). Може да видите че машинарията да се предотврати това е флагът **started** и подтиснатия **start()** метод.

В **run()** **count1** и **count2** се инкрементират и изобразяват по начин който сякаш ги държи идентични. После се вика **sleep()**; без това викане програмата се осуетява понеже става трудно за CPU-то да сменя задачите.

Методът **synchTest()** има явно ненужната активност да проверява дали **count1** е равен на **count2**; Ако не са той слага етикета да бъде "Unsyncched" за да се индицира това. Но първо вика статичния член на класа **Sharing1** който инкрементира и изобразява брояча за достъп за да покаже колко пъти проверката е давала равенство. (Причината за това ще стане ясна в следващи версии на този пример.)

Класът **Watcher** е нишка чиято задача е да вика **synchTest()** за всичките **TwoCounter** които са активни. Той го прави вървейки по масива който се пази в обекта **Sharing1**. Може да мислим че **Watcher** постоянно надничва иззад гърба на **TwoCounter** обектите.

Sharing1 съдържа масив от **TwoCounter** които инициализира в **init()** и стартира като нишки когато натиснете бутона "start". По-късно, когато натиснете бутона "Observe", един или повече наблюдатели се създават и се пускат между нищо неподозиращите **TwoCounter** нишки.

Забележете че за да пуснем това като аплет в браузър, вашата Web страница ще трябва да има и следните редове:

```

<applet code=Sharing1 width=650 height=500>
<param name=size value="20">
<param name=observers value="1">
</applet>

```

Може да сменяте ширината, височината и параметрите за да удовлетворите експерименталните си вкусове. Чрез промяна на **size** и **observers** ще промените поведението на програмата. Може също да видите че тази програма е способна да тръгне като

самостоятелно приложение взимайки параметрите от командния ред (или слагайки ги по подразбиране).

Ето изненадващата част. В `TwoCounter.run()` безкрайния цикъл просто минава многократно през линиите:

```
t1.setText(Integer.toString(count1++));
t2.setText(Integer.toString(count2++));
```

(а също и спи, но това не е важно тук). Когато пуснете програмата, обаче, ще откриете че `count1` и `count2` ще бъдат виждани (от **Watcher**-а) неравни от време на време! Това е поради природата на нишките – те могат да бъдат стопирани по всяко време. Така понякога нишката спира между двете линии по-горе, та се случва нишката **Watcher** да дойде и да сравни в такъв момент, като намери, че броячите сочат различно.

Този пример показва фундаментален проблем с използването на нишките. Никога не знаете кога нишката може да заработи. Да си представим че седите на масата с вилица, канейки се да набучите последното парче пържола и тъкмо да го набодете, храната внезапно изчезва (понеже вашата нишка е била спряна и друга нишка е дошла и взела храната). Това е проблемът с който имаме работа.

Понякога не ви е грижа дали ресурсът който ползвате се ползва и от друг (храната е на някаква друга чиния). Но за да работи мултитрединга, трябва да има начин да се предотврати работата на много нишки с един ресурс, най-малкото в критичните периоди.

Предотвратяването на този род колизии е просто въпрос на заключване на ресурса докато някаква нишка работи с него. Първата нишка която има достъп до ресурса го заключва, след което другите нишки нямат достъп докато той се отключи, друга нишка го вземе и заключи и т.н. Ако предната седалка на автомобила е ограничения ресурс, детето което вика "Пу за мён!" го заключва.

Как Java споделя ресурси

Java има вградена поддръжка за предотвратяване на колизии с един вид ресурс: паметта на обект. Тъй като обикновено правите данновите елементи на класа **private** и ги обработвате само чрез методи, може да предотвратите колизиите чрез провенето на някой метод **synchronized**. Само една нишка едновременно може да вика **synchronized** метод за даден обект (макар и да може да се викат такива методи за различни обекти). Ето прости **synchronized** методи:

```
synchronized void f() { /* ... */ }
synchronized void g(){ /* ... */ }
```

Всеки обект съдържа единствена ключалка (наричана също *monitor*) която се включва автоматично в обекта (не е необходимо вие да пишете нещо). Когато викате какъвто и да е **synchronized** обектът се заключва и никой друг **synchronized** метод на този обект не може да се вика докато първият не завърши и не отключи. В горния пример ако се извика **f()** за обект, **g()** не може да се вика за същия обект докато **f()** не завърши и отключи. Така, има единствена ключалка която се споделя от всички **synchronized** методи на конкретен обект, като тази ключалка предотвратява писането в паметта на обекта от повече от един метод едновременно (т.е. повече от една нишка едновременно).

Има също и единствена ключалка за клас (като част от **Class** обекта за класа), така че **synchronized static** могат да се заключват едни други от **static** данни на база клас (не обект-б.пр.).

Забележете че ако искате да регулирате достъпа до друг ресурс може да направите това като прекарате достъпа през ползването на **synchronized** методи.

Синхронизиране на броячите

Въоръжени с новите знания виждаме решението под ръка: просто ще използваме ключовата дума **synchronized** за методите в **TwoCounter**. Следващия пример е същия като предишния, само е добавена новата дума:

```
//: c14:Sharing2.java
// Using the synchronized keyword to prevent
// multiple access to a particular resource.
// <applet code=Sharing2 width=650 height=500>
// <param name=size value="20">
// <param name=observers value="1">
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class TwoCounter2 extends Thread {
    private boolean started = false;
    private JTextField t1 = new JTextField(5),
    t2 = new JTextField(5);
    private JLabel l =
        new JLabel("count1 == count2");
    private int count1 = 0, count2 = 0;
    public TwoCounter2(Container c) {
        JPanel p = new JPanel();
        p.add(t1);
        p.add(t2);
        p.add(l);
        c.add(p);
    }
    public void start() {
        if(!started) {
            started = true;
            super.start();
        }
    }
    public synchronized void run() {
        while (true) {
            t1.setText(Integer.toString(count1++));
            t2.setText(Integer.toString(count2++));
            try {
                sleep(500);
            } catch (InterruptedException e){}
        }
    }
    public synchronized void synchTest() {
        Sharing2.incrementAccess();
        if(count1 != count2)
            l.setText("Unsynched");
    }
}

class Watcher2 extends Thread {
    private Sharing2 p;
    public Watcher2(Sharing2 p) {
```

```

        this.p = p;
        start();
    }
    public void run() {
        while(true) {
            for(int i = 0; i < p.s.length; i++)
                p.s(i).synchTest();
            try {
                sleep(500);
            } catch (InterruptedException e){}
        }
    }
}

public class Sharing2 extends JApplet {
    TwoCounter2[] s;
    private static int accessCount = 0;
    private static JTextField aCount =
        new JTextField("0", 7);
    public static void incrementAccess() {
        accessCount++;
        aCount.setText(Integer.toString(accessCount));
    }
    private JButton
        start = new JButton("Start"),
        observer = new JButton("Observe");
    private boolean isApplet = true;
    private int numCounters = 0;
    private int numObservers = 0;
    public void init() {
        if(isApplet) {
            numCounters =
                Integer.parseInt(getParameter("size"));
            numObservers =
                Integer.parseInt(
                    getParameter("observers"));
        }
        s = new TwoCounter2[numCounters];
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        for(int i = 0; i < s.length; i++)
            s[i] = new TwoCounter2(cp);
        JPanel p = new JPanel();
        start.addActionListener(new StartL());
        p.add(start);
        observer.addActionListener(new ObserverL());
        p.add(observer);
        p.add(new Label("Access Count"));
        p.add(aCount);
        cp.add(p);
    }
    class StartL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            for(int i = 0; i < s.length; i++)
                s[i].start();
        }
    }
}

```

```

class ObserverL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        for(int i = 0; i < numObservers; i++)
            new Watcher2(Sharing2.this);
    }
}
public static void main(String[] args) {
    Sharing2 applet = new Sharing2();
    // This isn't an applet, so set the flag and
    // produce the parameter values from args:
    applet.isApplet = false;
    applet.numCounters =
        (args.length == 0 ? 5 :
         Integer.parseInt(args[0]));
    applet.numObservers =
        (args.length < 2 ? 5 :
         Integer.parseInt(args[1]));
    JFrame frame = new JFrame("Sharing2");
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    frame.getContentPane().add(applet);
    frame.setSize(350, applet.numCounters * 50);
    applet.init();
    applet.start();
    frame.setVisible(true);
}
} ///:~

```

Ще забележите че и **run()** и **synchTest()** са **synchronized**. Ако синхронизирате само единия от методите, вторият може да си действа като игнорира заключването. Това е важно: Всеки метод който работи с критичен разделяем ресурс трябва да бъде **synchronized** или нещата няма да са добре.

Сега изниква нов въпрос. **Watcher2** никога не може да види какво става понеже методът **run()** е **synchronized**, а понеже **run()** винаги работи, за всеки обект клечалката е заключена и **synchTest()** не може да бъде извикан. Това може да се види защото **accessCount** никога не се мени.

Необходимото в случая е да може да се изолира само част от кода в **run()**. Секцията от кода който искате да изолирате по този начин се нарича **критична секция** и ключовата дума **synchronized** се използва по друг начин за да се оформи критичната секция. Java поддържа критични секции със **синхронизиран блок**: този път **synchronized** се използва за означаване на обекта чиято клечалка ще се използва за заключване на кода в скобите:

```

synchronized(syncObject) {
    // This code can be accessed
    // by only one thread at a time
}

```

Преди да се влезе в блока, трябва да се види клечалката на **syncObject**. Ако някоя друга нишка я владее вече, блокът не може да се изпълни докато не се освободи.

Примерът **Sharing2** може да се промени чрез мащане на ключовата дума **synchronized** от целия **run()** и слагането на **synchronized** блок включващ двата реда. Но кой обект трябва да

се използва като ключалка? Такъв който вече е уважен от `synchTest()`, какъвто е текущият обект (`this`)! Така променения `run()` е:

```
public void run() {  
    while (true) {  
        synchronized(this) {  
            t1.setText(Integer.toString(count1++));  
            t2.setText(Integer.toString(count2++));  
        }  
        try {  
            sleep(500);  
        } catch (InterruptedException e){}  
    }  
}
```

Това е единствената промяна която трябва да бъде направена в `Sharing2.java` и ще видите, че докато двата брояча никога не излизат от синхронизация (когато `Watcher` ги гледа), има адекватен достъп на `Watcher` по време на изпълнението на `run()`.

Разбира се, синхронизацията зависи от програмистката прилежност: всяко парченце код което може да получи достъп до разделяем ресурс трябва да е в синхронизиран блок.

Синхронизирана ефективност

Тъй като да имаме два метода които пишат в едно място едновременно никога не звуци като особено добра идея, сякаш има смисъл всички методи да се направят автоматично `synchronized` и да се махне ключовата дума `synchronized` въобще. (Разбира се, `synchronized run()` показва че това няма да работи.) Оказва се обаче че проверката на ключалките не е евтина операция – увеличава се цената на викането на метода (влизането и излизането от метода, не изпълнението на тялото му) най-малко четири пъти и може да бъде повече в зависимост от приложението ви. Така че ако знаете за даден метод че може без да е `synchronized`, не го правете такъв.

Java Beans разгледани пак

Сега като разбирате синхронизацията може да погледнем пак Java Beans. Винаги когато създавате Bean, трябва да предполагате че той ще работи в многонишкова среда. Това значи че:

1. Винаги когато е възможно всички публични методи на Bean-а ще са `synchronized`. Разбира се, това дава загубите от `synchronized`. Ако това е проблем някои методи могат да се оставят **не-synchronized**, но помнете че не винаги е очевидно кои. Методите които имат тенденция да са малки (такива като `getCircleSize()` в следващия пример) и/или “атомични,” тоест методът се изпълнява за толкова кратко време, че обектът не се е променил. Правенето на такива методи **не-synchronized** може и да няма значителен ефект върху скоростта на изпълнение на програмата. Бихте могли също така да направите всичките `public` методи на Bean `synchronized` и да махнете ключовата дума `synchronized` само ако знаете че може и че ще има значително влияние.
2. Когато задействате мултикастно събитие за няколко слушателя трябва да очаквате че слушатели могат да се добавят или махнат по време на обработката на събитието - както си се движат по списъка.

Първата точка е лесна за справяне, но втората изисква малко повече мисъл. Да вземем `BangBean.java` примера представен в предната глава. Там зарязахме мултитрединга чрез изпускането на ключовата дума `synchronized` (която още не беше въведена) и правенето на

събитията уникастни. Ето примера пренаписан за многонишкова среда и мултикастинг на събитията:

```
//: c14:BangBean2.java
// You should write your Beans this way so they
// can run in a multithreaded environment
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;

public class BangBean2 extends JPanel
    implements Serializable {
    private int xm, ym;
    private int cSize = 20; // Circle size
    private String text = "Bang!";
    private int fontSize = 48;
    private Color tColor = Color.red;
    private ArrayList actionListeners =
        new ArrayList();
    public BangBean2() {
        addMouseListener(new MLO());
        addMouseMotionListener(new MMO());
    }
    public synchronized int getCircleSize() {
        return cSize;
    }
    public synchronized void
    setCircleSize(int newSize) {
        cSize = newSize;
    }
    public synchronized String getBangText() {
        return text;
    }
    public synchronized void
    setBangText(String newText) {
        text = newText;
    }
    public synchronized int getFontSize() {
        return fontSize;
    }
    public synchronized void
    setFontSize(int newSize) {
        fontSize = newSize;
    }
    public synchronized Color getTextColor() {
        return tColor;
    }
    public synchronized void
    setTextColor(Color newColor) {
        tColor = newColor;
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(Color.black);
        g.drawOval(xm - cSize/2, ym - cSize/2,
```

```

cSize, cSize);
}

// This is a multicast listener, which is
// more typically used than the unicast
// approach taken in BangBean.java:
public synchronized void
    addActionListener(ActionListener l) {
    actionListeners.add(l);
}

public synchronized void
    removeActionListener(ActionListener l) {
    actionListeners.remove(l);
}

// Notice this isn't synchronized:
public void notifyListeners() {
    ActionEvent a =
        new ActionEvent(BangBean2.this,
            ActionEvent.ACTION_PERFORMED, null);
    ArrayList lv = null;
    // Make a shallow copy of the vector in case
    // someone adds a listener while we're
    // calling listeners:
    synchronized(this) {
        lv = (ArrayList)actionListeners.clone();
    }
    // Call all the listener methods:
    for(int i = 0; i < lv.size(); i++) {
        ActionListener al =
            (ActionListener)lv.get(i);
        al.actionPerformed(a);
    }
}

class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        Graphics g = getGraphics();
        g.setColor(tColor);
        g.setFont(
            new Font(
                "TimesRoman", Font.BOLD, fontSize));
        int width =
            g.getFontMetrics().stringWidth(text);
        g.drawString(text,
            (getSize().width - width) / 2,
            getSize().height/2);
        g.dispose();
        notifyListeners();
    }
}

class MM extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
        xm = e.getX();
        ym = e.getY();
        repaint();
    }
}

public static void main(String[] args) {
    BangBean2 bb = new BangBean2();
}

```

```

bb.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e){
        System.out.println("ActionEvent" + e);
    }
});
bb.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e){
        System.out.println("BangBean2 action");
    }
});
bb.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e){
        System.out.println("More action");
    }
});
JFrame frame = new JFrame("BangBean2 Test");
frame.addWindowListener(new WindowAdapter(){
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});
frame.getContentPane().add(bb);
frame.setSize(300,300);
frame.setVisible(true);
}
} //:~

```

Добавянето на **synchronized** към методите е лесна промяна. Забележете обаче в **addActionListener()** и **removeActionListener()** че **ActionListener**ите сега се добавят към и мащат от **ArrayList**, така че може да бъдат колкото си искате.

Вижда се че **notifyListeners()** не е **synchronized**. Той може да се вика от повече от една нишка едновременно. Възможно е също за **addActionListener()** или **removeActionListener()** да бъдат викани по средата на работата на **notifyListeners()**, което е проблем, понеже той обикаля **ArrayList actionListeners**. За да се избегнат проблема, **ArrayList**ът се клонира в **synchronized** клауза и се преравя клонинга. По този начин оригиналният **ArrayList** може да се манипулира без да влияе на **notifyListeners()**.

Методът **paint()** също не е **synchronized**. Решението да се оставят несинхронизирани методи не е така лесно както когато добавяте свои собствени методи. В този пример излиза че **paint()** сякаш работи добре независимо дали е **synchronized** или не. Но тук трябва да се имат предвид следните неща:

1. Променя ли методът "критични" променливи в обекта? ТЗа да се открие дали променливата е "критична" трябва да се определи дали тя се променя или чете от други нишки. (В този случай това практически винаги се прави от **synchronized** методи, така че просто трябва да видите дали са такива.) В случая на **paint()** не се прави промяна.
2. Зависи ли методът от състоянието на тези "критични" променливи? Ако **synchronized** променя състоянието на променлива която се използва от вашия метод, много добре ще е да направите също и вашия метод **synchronized**. Като имате предвид това, може да забележите че **cSize** се променя от **synchronized** методи и затова **paint()** трябва да бъде **synchronized**. Тук обаче може да се запита "Какво най-лошо може да стане ако **cSize** се промени по време на **paint()**?" Като видите че не е чак толкова страшно, може да решите да оставите **paint()** не-**synchronized** за да се избегне допълнителната работа свързана с викането на **synchronized** метод.

3. Третото е да видите дали версията `paint()` от базовия клас е **synchronized**, а тя не е. Това е херметически аргумент, лепило (има игра на думи, лепило-нишка-свидетелство - б.пр.). В този пример стойността която се променя от **synchronized** методи (тя е `cSize`) е била замесена в `paint()` формата и може да е променила ситуацията. Забележете обаче че **synchronized** не се наследява – тоест ако един метод е **synchronized** в базовия клас той не е автоматично **synchronized** в подтиснатата версия от извлечения клас.

Тестваният код в **TestBangBean2** е променен спрямо предния пример заради мултикастните възможности на **BangBean2** чрез добавяне на нови слушатели.

БЛОКИРАНЕ

Една нишка може да бъде в едно от следните четири състояния:

1. *New*: обектът на нишката е създаден но не е стартиран и нишката не действа.
2. *Runnable*: Това значи че нишката може да работи когато CPU има цикли за нея. Така нишката може да работи и може да не работи, но нищо не пречи да работи когато разпределителят на времето я пусне; тя не е мъртва или блокирана.
3. *Dead*: нормалния начин нишката да умре е да завърши нейният `run()` метод. Може също да се извика `stop()`, но това изхвърля изключение, което е подклас на **Error** (което значи че обикновено не го хващате). Помните че изхвърлянето на изключение трябва да бъде изключително събитие, а не част от нормалното изпълнение на програмата; така използването на `stop()` не се окуражава (и той е остатъл в Java 2). Има също метод `destroy()` (който никога не е бил реализиран) който никога няма да викате ако не се налага понеже е драстичен и не освобождава ключалките на обекта.
4. *Blocked*: нишката може да бъде пусната но нещо пречи. Когато нишката е блокирана планировчикът просто я пропуска и не дава CPU време. Докато нишката не стане отново рънъбл няма да работи.

Как се блокира нишка

Блокираното състояние е най-интересно и заслужава още разглеждане. Нишка може да бъде блокирана по пет причини:

1. Накарали сте я да заспи чрез `sleep(milliseconds)`, в който случай тя няма да работи опрезеленото време.
2. Прекъснали сте изпълнението на нишката със `suspend()`. Няма да стане рънъбл докато не приеме `resume()` съобщение.
3. Прекъснали сте изпълнението с `wait()`. Няма да работи докато не приеме `notify()` или `notifyAll()` съобщение. (Да, това изглежда точно като номер 2, но има значителна разлика която ще се опише.)
4. Нишката чака да завърши някакъв IO.
5. Нишката се опитва да извика **synchronized** метод от друг обект и той не е достъпен.

Може също да извикате **`yield()`** (метод на класа **`Thread`**) за доброволно оставяне на CPU на другите нишки. Същото обаче става ако планировчикът реши че вашата нишка е работила достатъчно дълго. Ако нищо не пречи на нишката да работи. Когато е блокирана, нещо пречи да работи.

Следният пример показва всичките пет начина за блокиране. Той целият е в един файл наречен **`Blocking.java`**, но ще се разгледа тук на отделни части. (Ще видите “Continued” и “Continuing” тагове които позволяват на инструментата за извлечане на код да ги събере заедно.) Първо, основната рамка:

```
//: c14:Blocking.java
// Demonstrates the various ways a thread
// can be blocked.
// <applet code=Blocking width=350 height=550>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

/////////// The basic framework ///////////
class Blockable extends Thread {
    private Peeker peeker;
    protected JTextField state = new JTextField(30);
    protected int i;
    public Blockable(Container c) {
        c.add(state);
        peeker = new Peeker(this, c);
    }
    public synchronized int read() { return i; }
    protected synchronized void update() {
        state.setText(getClass().getName()
            + " state: i = " + i);
    }
    public void stopPeeker() {
        // peeker.stop(); Deprecated in Java 1.2
        peeker.terminate(); // The preferred approach
    }
}

class Peeker extends Thread {
    private Blockable b;
    private int session;
    private JTextField status = new JTextField(30);
    private boolean stop = false;
    public Peeker(Blockable b, Container c) {
        c.add(status);
        this.b = b;
        start();
    }
    public void terminate() { stop = true; }
    public void run() {
        while (!stop) {
            status.setText(b.getClass().getName()
                + " Peeker " + (++session)
                + "; value = " + b.read());
            try {
```

```

        sleep(100);
    } catch (InterruptedException e){}
}
}
} ///:Continued

```

Класът **Blockable** ще е базовия клас за класовете в този пример който ще демонстрира блокирането. Един **Blockable** обект съдържа **TextField** наречен **state** който се използва за изобразяване на информацията за обект. Методът който извежда информацията е **update()**. Може да видите че той използва **getClass().getName()** за да състави името на класа, заместо само да го избутва навън; това е защото **update()** не може да знае точното име на класа за който е извикан, щом това е клас извлечен от **Blockable**.

Индикатор на промяна в **Blockable** е **int i**, която ще се инкрементира от **run()** метода на извлечения клас.

Има нишка от клас **Peeker** която е стартирана за всеки **Blockable** обект и работата на **Peeker** е да следи **Blockable** за промени в **i** чрез викане на **read()** и докладвайки ги в техния **status TextField**. Това е важно: Забележете че **read()** и **update()** са и двете **synchronized**, което значи че за тях е необходимо ключалката на обекта да е свободна.

Спаге

Първият тест в тази програма е със **sleep()**:

```

///:Continuing
/////////// Blocking via sleep() ///////////
class Sleeper1 extends Blockable {
    public Sleeper1(Container c) { super(c); }
    public synchronized void run() {
        while(true) {
            i++;
            update();
            try {
                sleep(1000);
            } catch (InterruptedException e){}
        }
    }
}

class Sleeper2 extends Blockable {
    public Sleeper2(Container c) { super(c); }
    public void run() {
        while(true) {
            change();
            try {
                sleep(1000);
            } catch (InterruptedException e){}
        }
    }
    public synchronized void change() {
        i++;
        update();
    }
} ///:Continued

```

В **Sleeper1** целият метод **run()** е **synchronized**. Ще видите че асоциирания с този обект **Peeker** ще продължи щастливо докато пуснете нишката, а после **Peeker** застива. Това е една форма

на блокиране: понеже **Sleeper1.run()** е **synchronized** и като се стартира нишката е винаги в **run()**, методът никога не докопва ключалката и **Peeker** е блокиран.

Sleeper2 дава решение чрез правене на **run** не-**synchronized**. Само **change()** методът е **synchronized**, което значи че докато **run()** е в **sleep()**, **Peeker** може да има достъп до **synchronized** метода който му трябва, именно **read()**. Тук ще видите че **Peeker** продължава работа след като стартирате нишката **Sleeper2**.

Спиране и продължаване

Следващата част от примера ви запознава с концепцията за спирането. Класът **Thread** има метод **suspend()** за временно спиране на нишката и **resume()** който рестартира от точката където е било прекъснато. Предполагамо, **resume()** се вика от друга, различна от прекъснатата нишка и в този случай има отделен клас **Resumer** който прави точно това. В секи от класовете демонстриращи **suspend/resume** има асоцииран продължител:

```
///:Continuing
////////// Blocking via suspend() //////////
class SuspendResume extends Blockable {
    public SuspendResume(Container c) {
        super(c);
        new Resumer(this);
    }
}

class SuspendResume1 extends SuspendResume {
    public SuspendResume1(Container c) { super(c);}
    public synchronized void run() {
        while(true) {
            i++;
            update();
            suspend(); // Deprecated in Java 1.2
        }
    }
}

class SuspendResume2 extends SuspendResume {
    public SuspendResume2(Container c) { super(c);}
    public void run() {
        while(true) {
            change();
            suspend(); // Deprecated in Java 1.2
        }
    }
    public synchronized void change() {
        i++;
        update();
    }
}

class Resumer extends Thread {
    private SuspendResume sr;
    public Resumer(SuspendResume sr) {
        this.sr = sr;
        start();
    }
    public void run() {
```

```

while(true) {
    try {
        sleep(1000);
    } catch (InterruptedException e){}
    sr.resume(); // Deprecated in Java 1.2
}
}

} ///:Continued

```

SuspendResume1 има също **synchronized run()** метод. Отново когато стартирате нишката се вижда че асоциирания **Peeker** се блокира чакайки за освобождаване на ключалката, което никога не става. Това е оправено както преди с **SuspendResume2**, който не **synchronize** целия **run()** ами използва **synchronized change()** метод.

Трябва да сте предупредени че в Java 2 са остарели **suspend()** и **resume()**, понеже **suspend()** държи ключалката на обекта и затова създава предпоставки за мъртво блокиране. Това е когато имате няколко блокирани обекта чакащи се един друг, което причинява зависване на програмата. Макар и да може да ги видите в стари програми няма да използвате **suspend()** и **resume()**. Правилното решение се дава по-нататък в тази глава.

Чакай и бъди уведомен

Номерът е че в първите два промера **sleep()** и **suspend()** не освобождават ключалката като се извикат. Трябва да сте предупредени за това когато работите с ключалки. От друга страна, методът **wait()** освобождава ключалката когато се извика, което значи че други **synchronized** в нишковия обект могат да работят докато трае **wait()**. В следващите два случая ще видите че методът **run()** е напълно **synchronized** и в двата случая, обаче **Peeker** пак има пълен достъп до **synchronized** методи по време на **wait()**. Това е защото **wait()** освобождава ключалката.

Ще видите също че има две форми на **wait()**. Първата взима аргумент в милисекунди със значение като при **sleep()**: пауза за този период от време. Разликата е че при **wait()** ключалката се освобождава и може да излезете от **wait()**-а при **notify()** както и при изтичане на времето.

Втората форма няма аргументи и значи че **wait()** ще продължава докато дойде **notify()** и няма да свърши след определено време.

Един доста уникатен аспект на **wait()** и **notify()** е че и двата са част от базовия клас **Object** и не са част от **Thread** каквото са **sleep()**, **suspend()** и **resume()**. Макар и това да изглежда странно на пръв поглед – да имате нещо което е за нишките като част от изосновния базов клас – така е понеже те манипулират ключалките които са част от всеки обект. Като резултат може да сложите **wait()** навсякъде в **synchronized** метод, без значение дали има мулитрединг или е в клас. Фактически единственото място където може да викате **wait()** е от **synchronized** метод или блок. Ако извикате **wait()** или **notify()** в метод който не е **synchronized**, програмата ще се компилира, но ще получите по време на изпълнение **IllegalMonitorStateException** с малко неинтуитивното съобщение “current thread not owner.” Забележете че **sleep()**, **suspend()** и **resume()** могат да се викат в не-**synchronized** методи тъй като не пипат ключалките.

Може да викате **wait()** или **notify()** само за ваша си ключалка. Отново кодът се компилира ако не е така, но имате същото **IllegalMonitorStateException** като преди. Не може да лъжете за чужда ключалка, но може да поискате обектът чиято е да я освободи. Така че единият подход е да се създаде **synchronized** метод който вика **notify()** за неговия си обект. Обаче в **Notifier** ще видите викане на **notify()** вътре в **synchronized** блок:

```

synchronized(wn2) {
    wn2.notify();
}

```

```
| }
```

където **wn2** е обект от тип **WaitNotify2**. Този метод който не е част от **WaitNotify2** иска ключалката на **wn2** обект, в която точка е законно за него да вика **notify()** за **wn2** и не се получава **IllegalMonitorStateException**.

```
///:Continuing
////////// Blocking via wait() ///////////
class WaitNotify1 extends Blockable {
    public WaitNotify1(Container c) { super(c); }
    public synchronized void run() {
        while(true) {
            i++;
            update();
            try {
                wait(1000);
            } catch (InterruptedException e){}
        }
    }
}

class WaitNotify2 extends Blockable {
    public WaitNotify2(Container c) {
        super(c);
        new Notifier(this);
    }
    public synchronized void run() {
        while(true) {
            i++;
            update();
            try {
                wait();
            } catch (InterruptedException e){}
        }
    }
}

class Notifier extends Thread {
    private WaitNotify2 wn2;
    public Notifier(WaitNotify2 wn2) {
        this.wn2 = wn2;
        start();
    }
    public void run() {
        while(true) {
            try {
                sleep(2000);
            } catch (InterruptedException e){}
            synchronized(wn2) {
                wn2.notify();
            }
        }
    }
} ///:Continued
```

wait() типично се използва когато сте дошли до точка, в която чакате изпълнението на някакво условие, контролтът на което не е във вашата нишка, да се промени и не щете нишката да

цикли безсмислено. Така **wait()** позволява да се приспи нишката докато се чака светът да се промени, само когато **notify()** или **notifyAll()** дойде нишката ще се събуди и ще огледа за промени. Така се дава начин за синхронизация между нишките.

Блокиране по IO

Ако поток изчаква някаква IO активност, той автоматично ще се блокира. В следващата порция пример два класа работят с родов **Reader** и **Writer** обекти (използвайки Streams на Java 1.1), но за тестовата работна рамка ще се организира канал за да се позволи на две нишки безопасно да обменят данни (което е предназначението на канализираните потоци).

Sender слага данни в **Writer** и спи случайно определено време. Обаче **Receiver** няма **sleep()**, **suspend()** или **wait()**. Но когато прави **read()** автоматично се блокира когато няма повече данни.

```
///:Continuing
class Sender extends Blockable { // send
    private Writer out;
    public Sender(Container c, Writer out) {
        super(c);
        this.out = out;
    }
    public void run() {
        while(true) {
            for(char c = 'A'; c <= 'z'; c++) {
                try {
                    i++;
                    out.write(c);
                    state.setText("Sender sent: "
                        + (char)c);
                    sleep((int)(3000 * Math.random()));
                } catch (InterruptedException e){}
                catch (IOException e) {}
            }
        }
    }
}

class Receiver extends Blockable {
    private Reader in;
    public Receiver(Container c, Reader in) {
        super(c);
        this.in = in;
    }
    public void run() {
        try {
            while(true) {
                i++; // Show peeker it's alive
                // Blocks until characters are there:
                state.setText("Receiver read: "
                    + (char)in.read());
            }
        } catch(IOException e) { e.printStackTrace();}
    }
} ///:Continued
```

И двата класа слагат информация в техните полета **state** и променят **i** та **Peeker** може да види че нишката работи.

Тестване

Главния клас на аплета е изненадващо прост защото повечето от работите са сложени в рамката **Blockable**. Основно се създава масив от **Blockable** обекти се създава и тъй като всеки е нишка, те изпълняват своите си активности когато натиснете бутона "start". Има също бутон и клауза **actionPerformed()** да се спрат всичките обекти **Peeker**, което демонстрира алтернатива за избягването на остателя метод (в Java 2) **stop()** на **Thread**.

За да се оформи връзка между **Sender** и **Receiver** обектите, **PipedWriter** и **PipedReader** се създават. Забележете че **PipedReader in** трябва да бъде закачен към **PipedWriter out** чрез аргумент на конструктора. След това всичко което се сложи в **out** може по-късно да бъде извлечено от **in**, като да е минало по канал (и от там името). **in** и **out** обектите после се дават на **Receiver** и **Sender** конструктори, респективно, които ги третират като **Reader** и **Writer** обекти от някакъв тип (тоест, ъпкаст).

Масивът от **Blockable** манипулатори **b** не е инициализиран в точката на задаването си понеже каналните потоци не могат да се организират преди да е налице дефиницията (нуждата от **try** блок предотвратява това).

```
///:Continuing
////////// Testing Everything //////////
public class Blocking extends JApplet {
    private JButton
        start = new JButton("Start"),
        stopPeekers = new JButton("Stop Peekers");
    private boolean started = false;
    private Blockable() b;
    private PipedWriter out;
    private PipedReader in;
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        out = new PipedWriter();
        try {
            in = new PipedReader(out);
        } catch(IOException e) {}
        b = new Blockable()
            new Sleeper1(cp),
            new Sleeper2(cp),
            new SuspendResume1(cp),
            new SuspendResume2(cp),
            new WaitNotify1(cp),
            new WaitNotify2(cp),
            new Sender(cp, out),
            new Receiver(cp, in)
        ;
        start.addActionListener(new StartL());
        cp.add(start);
        stopPeekers.addActionListener(
            new StopPeekersL());
        cp.add(stopPeekers);
    }
    class StartL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
```

```

if(!started) {
    started = true;
    for(int i = 0; i < b.length; i++)
        b(i).start();
}
}

class StopPeekersL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        // Demonstration of the preferred
        // alternative to Thread.stop():
        for(int i = 0; i < b.length; i++)
            b(i).stopPeeker();
    }
}

public static void main(String[] args) {
    JApplet applet = new Blocking();
    JFrame frame = new JFrame("Blocking");
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    frame.getContentPane().add(applet);
    frame.setSize(350,550);
    applet.init();
    applet.start();
    frame.setVisible(true);
}
} ///:~

```

В **init()** забележете цикъла който обикаля целия масив и добавя **state** и **peeker.status** текстовите полета в страницата.

Когато **Blockable** нишките се създават първоначално, всяка създава и стартира нейн собствен **Peeker**. Така че ще видите **Peekers** работещи преди **Blockable** нишки да са създадени. Това е важно, понеже някои **Peekers** ще бъдат блокирани и ще спрат преди **Blockable** да се стартират и това е важно за разбирането на този конкретен аспект на блокирането.

Мъртъв блокаж

Понеже нишките могат да се блокират и понеже обектите могат да имат **synchronized** методи които предотвратяват достъпа до него обект докато синхронизацията ключалка се освободи, възможно е една нишка да спре за да изчака ключалка на друга нишка, която на свой ред чака трета нишка и т.н., докато веригата се затвори на първата нишка. Така става кръг от нишки които се изчакват една друга. Това се нарича *deadlock*. Счита се че става рядко, но когато стане е разочаровашо при дебъгване.

Няма поддръжка от езика за предотвратяване на мъртвия блокаж; вие трябва да го избегнете чрез грижливо проектиране. Няма успокояващ думи за човек, който се опитва да тества програма с мъртъв блокаж.

Остарялост на **stop()**, **suspend()**, **resume()** и **destroy()** в Java 2

В Java 2 има една промяна насочена към намаляване на вероятността от мъртвъ блокаж и това е че методите на **Thread stop()**, **suspend()**, **resume()** и **destroy()** са остарели.

Причината че методът **stop()** е остатял е че той не е безопасен. Той освобождава всички ключалки на обекта и ако обектът е в ненормално състояние ("повреден") другите нишки могат да работят с него и да го променят както е повреден (от което не се знае какво ще излезе - б.пр.). Резултиращите проблеми могат да са скрити и трудни за откриване. Вместо да се използва **stop()** ще следвате примера в **Blocking.java** и ще използвате флаг за да кажете на нишката че може да се самоликвидира като излезе от **run()** метода.

Случва се нишка да блокира, например когато чака вход, и да не може да преглежда флага както в **Blocking.java**. В такива случаи пок няма да използвате **stop()**, ами метода **interrupt()** в **Thread** за да се отскубнете от блокирания код:

```
//: c14:Interrupt.java
// The alternative approach to using stop()
// when a thread is blocked
// <applet code=Interrupt width=200 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class Blocked extends Thread {
    public synchronized void run() {
        try {
            wait(); // Blocks
        } catch(InterruptedException e) {
            System.out.println("InterruptedException");
        }
        System.out.println("Exiting run()");
    }
}

public class Interrupt extends JApplet {
    private JButton interrupt = new JButton("Interrupt");
    private Blocked blocked = new Blocked();
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(interrupt);
        interrupt.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    System.out.println("Button pressed");
                    if(blocked == null) return;
                    Thread remove = blocked;
                    blocked = null; // to release it
                    remove.interrupt();
                }
            });
    }
}
```

```

    blocked.start();
}
public static void main(String[] args) {
    JApplet applet = new Interrupt();
    JFrame frame = new JFrame("Interrupt");
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    frame.getContentPane().add(applet);
    frame.setSize(200,100);
    applet.init();
    applet.start();
    frame.setVisible(true);
}
} //:~

```

wait() в **Blocked.run()** блокира нишката. Когато натиснете бутона манипулаторът който е **blocked** се прави **null** така че боклучарят ще го почисти, а после се вика метода **interrupt()** на обекта. Първия път когато натиснете бутона ще видите че нишката приключва, но след това няма нишка за убиване така че ще видите просто че е натиснат бутона.

Методите **suspend()** и **resume()** се оказват наследствено склонни към причиняване на мъртв блокаж. Като извикате **suspend()**, съответната нишка спира но все още има заключени до този момент ключалки. Така че друга нишка не може да има достъп до заключените ресурси докато не ги освободи тази. Ако някоя от нишките която събужда спящата използва заключен ресурс мъртвият блокаж е готов. Няма да използвате **suspend()** и **resume()**, а вместо това ще сложите флаг във вашия **Thread** клас да отбележи дали нишката трябва да бъде активна или спряна. Ако флагът показва че нишката трябва да е спряна, тя спира с **wait()**. Когато флагът показва че трябва да е активна, тя се продължава с **notify()**. Може да се даде пример с промяна на **Counter2.java**. Макар че ефектът е подобен, ще забележите че организацията на кода е доста различна – използват се анонимни вътрешни класове за всички слушатели и също **Thread** е вътрешен клас, което прави програмирането малко по-удобно тъй като елиминира малко от допълнителното счетоводство в **Counter2.java**:

```

//: c14:Suspend.java
// The alternative approach to using suspend()
// and resume(), which have been deprecated
// in Java 2
// <applet code=Suspend width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Suspend extends JApplet {
    private JTextField t = new JTextField(10);
    private JButton
        suspend = new JButton("Suspend"),
        resume = new JButton("Resume");
    class Suspendable extends Thread {
        private int count = 0;
        private boolean suspended = false;
        public Suspendable() { start(); }
        public void fauxSuspend() {

```

```

        suspended = true;
    }
    public synchronized void fauxResume() {
        suspended = false;
        notify();
    }
    public void run() {
        while (true) {
            try {
                sleep(100);
                synchronized(this) {
                    while(suspended)
                        wait();
                }
            } catch (InterruptedException e){}
            t.setText(Integer.toString(count++));
        }
    }
}
private Suspendable ss = new Suspendable();
public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    suspend.addActionListener(
        new ActionListener() {
            public
            void actionPerformed(ActionEvent e) {
                ss.fauxSuspend();
            }
        });
    cp.add(suspend);
    resume.addActionListener(
        new ActionListener() {
            public
            void actionPerformed(ActionEvent e) {
                ss.fauxResume();
            }
        });
    cp.add(resume);
}
public static void main(String[] args) {
    JApplet applet = new Suspend();
    JFrame frame = new JFrame("Suspend");
    frame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    frame.getContentPane().add(applet);
    frame.setSize(300,100);
    applet.init();
    applet.start();
    frame.setVisible(true);
}
} ///:~

```

Флагът **suspended** в **Suspendable** се използва за включване и изключване на спирането. За спиране той се слага **true** чрез викане на **fauxSuspend()** и това се усеща в **run()**, **wait()**, както се описа по-рано, трябва да бъде **synchronized** така че съдържа ключалка на обекта. В **fauxResume()** флагът **suspended** се слага **false** и се вика **notify()** – понеже това събужда **wait()** вътре в **synchronized** клаузата методът **fauxResume()** също трябва да бъде **synchronized** така че проверява ключалката преди да вика **notify()** (така ключалката е достъпна за **wait()** за събуждане). Ако следвате показаното в този пример може да избегнете **wait()** и **notify()**.

Методът **destroy()** на **Thread** никога не е реализиран; той е като **suspend()** който не може да продълже, така че има същите проблеми с мъртъв блокаж като **suspend()**. Обаче това не е остатял метод и може да бъде реализиран в бъдещи версии на Java (след 2) за специални ситуации където рисъкът от мъртъв блокаж е приемлив.

Може да се чудите защо тези методи, сега останали, бяха включени в Java на първото място. Простото им изхвърляне изглежда като признание за значителна грешка (и први нова дупка в аргументите за изключителността на дизайна на Java и непогрешимостта натрапвана от продавачите в Sun). Ободряващата част от замяната е че тя явно показва че техниците а не търговците водят шоуто – те откриха проблема и го оправиха. Аз намирам това много по-обещаващо и помагащо отколкото оставянето на проблема за да не се признае грешката. То значи че Java ще продължи да се подобрява, даже и това за значи някои неудобства за част от Java програмистите. Аз предпочитам да се оправям с дискомфорта отколкото да наблюдавам как стагнира езика.

Приоритети

Приоритетът на нишка казва на планировчика колко е важна тя. Ако има няколко нишки блокирани и чакащи да започнат, планировчикът ще пусне онази с най-висок приоритет. Това обаче не значи че нишките с по-малки приоритети няма да тръгнат (тоест, не може да се получи мъртъв блокаж поради приоритетите). По-малкият приоритет просто показва че ще има тенденция към по-рядко пускане.

Може да прочетете приоритета на нишка с **getPriority()** и да го промените с **setPriority()**. Формата на предишните “counter” примери може да бъде използвана за показване на влиянието на приоритетите. В този пример ще видите че броячите се забавят според понижаването на приоритетите на асоциираните им нишки:

```
//: c14:Counter5.java
// Adjusting the priorities of threads
// <applet code=Counter5 width=450 height=600>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event;

class Ticker2 extends Thread {
    private JButton
        b = new JButton("Toggle"),
        incPriority = new JButton("up"),
        decPriority = new JButton("down");
    private JTextField
        t = new JTextField(10),
        pr = new JTextField(3); // Display priority
    private int count = 0;
    private boolean runFlag = true;
    public Ticker2(Container c) {
        b.addActionListener(new ToggleL());
```

```

incPriority.addActionListener(new UpL());
decPriority.addActionListener(new DownL());
JPanel p = new JPanel();
p.add(t);
p.add(pr);
p.add(b);
p.add(incPriority);
p.add(decPriority);
c.add(p);
}
class ToggleL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        runFlag = !runFlag;
    }
}
class UpL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        int newPriority = getPriority() + 1;
        if(newPriority > Thread.MAX_PRIORITY)
            newPriority = Thread.MAX_PRIORITY;
        setPriority(newPriority);
    }
}
class DownL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        int newPriority = getPriority() - 1;
        if(newPriority < Thread.MIN_PRIORITY)
            newPriority = Thread.MIN_PRIORITY;
        setPriority(newPriority);
    }
}
public void run() {
    while (true) {
        if(runFlag) {
            t.setText(Integer.toString(count++));
            pr.setText(
                Integer.toString(getPriority()));
        }
        yield();
    }
}
}

public class Counter5 extends JApplet {
    private JButton
        start = new JButton("Start"),
        upMax = new JButton("Inc Max Priority"),
        downMax = new JButton("Dec Max Priority");
    private boolean started = false;
    private static final int SIZE = 10;
    private Ticker2[] s = new Ticker2[SIZE];
    private TextField mp = new TextField(3);
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        for(int i = 0; i < s.length; i++)
            s[i] = new Ticker2(cp);
    }
}

```

```

cp.add(new JLabel(
    "MAX_PRIORITY = " + Thread.MAX_PRIORITY));
cp.add(new JLabel("MIN_PRIORITY = "
    + Thread.MIN_PRIORITY));
cp.add(new JLabel("Group Max Priority = "));
cp.add(mp);
cp.add(start);
cp.add(upMax);
cp.add(downMax);
start.addActionListener(new StartL());
upMax.addActionListener(new UpMaxL());
downMax.addActionListener(new DownMaxL());
showMaxPriority();
// Recursively display parent thread groups:
ThreadGroup parent =
    s(0).getThreadGroup().getParent();
while(parent != null) {
    cp.add(new Label(
        "Parent threadgroup max priority = "
        + parent.getMaxPriority()));
    parent = parent.getParent();
}
}
public void showMaxPriority() {
    mp.setText(Integer.toString(
        s(0).getThreadGroup().getMaxPriority()));
}
class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        if(!started) {
            started = true;
            for(int i = 0; i < s.length; i++)
                s(i).start();
        }
    }
}
class UpMaxL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        int maxp =
            s(0).getThreadGroup().getMaxPriority();
        if(++maxp > Thread.MAX_PRIORITY)
            maxp = Thread.MAX_PRIORITY;
        s(0).getThreadGroup().setMaxPriority(maxp);
        showMaxPriority();
    }
}
class DownMaxL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        int maxp =
            s(0).getThreadGroup().getMaxPriority();
        if(--maxp < Thread.MIN_PRIORITY)
            maxp = Thread.MIN_PRIORITY;
        s(0).getThreadGroup().setMaxPriority(maxp);
        showMaxPriority();
    }
}
public static void main(String[] args) {

```

```

JApplet applet = new Counter5();
JFrame frame = new JFrame("Counter5");
frame.addWindowListener(
    new WindowAdapter() {
        public void windowClosing(WindowEvent e){
            System.exit(0);
        }
    });
frame.getContentPane().add(applet);
frame.setSize(450, 600);
applet.init();
applet.start();
frame.setVisible(true);
}
} //:~

```

Ticker2 следва формата заложена в предишните масти на тази глава, но има допълнително **TextField** за изобразяване на приоритета и още два бутона за повишаване и понижаване на приоритета.

Забележете също използването на **yield()**, който доброволно връща управлението на планировчика. Без това многонишковостта все пак работи, но ще забележите че по-бавно (опитайте с махане на извикването на **yield()**!). Бихте могли да извикате също **sleep()**, но тогава скоростта на броене ще се определя от времетраенето на **sleep()** наместо от приоритета.

init() в **Counter5** създава масив от 10 **Ticker2**и; техните бутони и текстове се слагат в полето от **Ticker2** конструктора. **Counter5** слага бутони за стартиране на всичкото както и за инкрементиране и декрементиране на максималния приоритет на групата нишки. Освен това има етикети които показват максималния и минималния приоритет възможен за нишка и **TextField** за показване на максималния приоритет на групата нишки. (Следващата секция пълно описва групите нишки.) Накрая, приоритетите на родителската група нишки също се изобразяват като етикети.

Като натиснете бутон “up” или “down” приоритетът на **Ticker2** се намира и съответно увеличава или намалява.

Като пуснете тази програма ще забележите няколко неща. Преди всичко приоритетът по подразбиране на групата е 5. Даже и да намалите приоритета под 5 преди да стартирате нишките (или преди създаване на нишките, което изисква промяна на кода), всяка нишка ще има приоритет по подразбиране 5.

Простият тест е да се вземе един брояч и да се намали приоритетът на неговата нишка с единица и да се види че той брои много по-бавно. Но сега се опитайте да го увеличите отново. Може да го докарате обратно до приоритета на групата, но не по-висок. Сега намалете приоритета на групата два пъти. Приоритетите на нишките са непроменени, но ако се опитате да ги променяте (нагоре или надолу) ще видите че те стават колкото приоритета на групата. Също, на нови нишки се дава приоритета па падреизбиране, даже ако е по-висок от приоритета на групата. (Така приоритетът на групата не е начин да се даде на новите нишки по-нисък приоритет.)

Накрая, опитайте се да увеличите груповия максимален приоритет. Това не може да бъде направено. Може само да се намаляват груповите максимални приоритети, не да се увеличават.

Групи нишки

Всяка нишка принадлежи към някаква група. Тя може да бъде или тази по подразбиране, или посочена от вас явно при създаването на нишката. При създаването нишката се свързва с някаква група и не може после да премине в друга. Всяко приложение има поне една нишка която принадлежи към системната група нишки. Ако създадете още нишки без задаване на група, те също ще принадлежат към системната група нишки.

Групите нишки трябва също да принадлежат към други групи нишки. Групата нишки към която новата трябва да принадлежи трябва да се посочи в конструктора. Ако създадете група нишки без да посочите към коя група принадлежи, то ще е към системната група нишки. Така всички групи нишки определено ще имат за родителска група системната.

Трудно е да се определи от литературата причината за съществуването на групите нишки, литературата е противоречива по тази тема. Често се цитират "причини за сигурност." Съгласно Arnold & Gosling,¹ "Нишките в група нишки могат да променят другите нишки в групата, включително надолу по йерархията. Нишка не може да променя нишки извън групата си или съдържаните в нея групи." Трудно е да се каже какво трябва да означава "променя" тук. Примерът по-долу показва нишка в подгрупа която е "листо" (т.е. няма подгрупи - б.пр.) променяща приоритетите във всички групи от нейното дърво от групи нишки както и викаща методи за всички нишки от нейното дърво.

```
//: c14:TestAccess.java
// How threads can access other threads
// in a parent thread group

public class TestAccess {
    public static void main(String[] args) {
        ThreadGroup
            x = new ThreadGroup("x"),
            y = new ThreadGroup(x, "y"),
            z = new ThreadGroup(y, "z");
        Thread
            one = new TestThread1(x, "one"),
            two = new TestThread2(z, "two");
    }
}

class TestThread1 extends Thread {
    private int i;
    TestThread1(ThreadGroup g, String name) {
        super(g, name);
    }
    void f() {
        i++; // modify this thread
        System.out.println(getName() + " f()");
    }
}

class TestThread2 extends TestThread1 {
    TestThread2(ThreadGroup g, String name) {
        super(g, name);
        start();
    }
    public void run() {
        ThreadGroup g =

```

¹ *The Java Programming Language*, by Ken Arnold and James Gosling, Addison-Wesley 1996 pp 179.

```

getThreadGroup().getParent().getParent();
g.list();
Thread[] gAll = new Thread[g.activeCount()];
g.enumerate(gAll);
for(int i = 0; i < gAll.length; i++) {
    gAll[i].setPriority(Thread.MIN_PRIORITY);
    ((TestThread1)gAll[i]).f();
}
g.list();
}
} //:~

```

В **main()** се създават няколко **ThreadGroup**, разлиствайки се една от друга: **x** няма аргументи освен името си (един **String**), така че автоматично се слага в групата "system", докато **y** е под **x** и **z** е под **y**. Забележете че инициализацията става в текстов ред така че този код е легален.

Създават се две нишки и се слагат в различни групи. **TestThread1** няма **run()** но има **f()** което променя нишката и извежда нещо така че личи че е викан. **TestThread2** е подклас на **TestThread1** и неговият **run()** е добре изработен. Той първо взима групата на нишките от текущата група, после се качва нагоре по наследственото дърво два етажа чрез **getParent()**. (Това е защото аз нарочно сложих **TestThread2** обекта две нива надолу по юерархията.) В този момент се създава масив от **Thread** чрез метода **activeCount()** за да пита колко нишки има в тази група нишки и всички дъщерни групи. Методът **enumerate()** слага манипулатори към всички тези нишки в масива **gAll**, после просто се движи през масива викайки **f()** метода за всяка нишка, променяйки също и приоритета. Така нишка в група която е "листо" променя нишките в родителските групи.

Методът за тестване **list()** извежда цялата информация за група нишки на стандартния изход и е полезен когато се изследва поведението на нишките. Ето изхода от програмата:

```

java.lang.ThreadGroup(name=x,maxpri=10)
    Thread(one,5,x)
    java.lang.ThreadGroup(name=y,maxpri=10)
        java.lang.ThreadGroup(name=z,maxpri=10)
            Thread(two,5,z)
one f()
two f()
java.lang.ThreadGroup(name=x,maxpri=10)
    Thread(one,1,x)
    java.lang.ThreadGroup(name=y,maxpri=10)
        java.lang.ThreadGroup(name=z,maxpri=10)
            Thread(two,1,z)

```

list() не само извежда името на класа на **ThreadGroup** or **Thread**, но също и името на групата нишки и нейния максимален приоритет. За нишките се извежда името на нишката, следвано от приоритета и групата. Забележете че **list()** измества надясно информацията за подгрупите за да покаже дървото.

Може да се види че **f()** се вика от **TestThread2 run()** метод, така че всички нишки от групата са уязвими. Но може да имате достъп само до нишките които се разклоняват от дървото на вашата **system** нишка и може би това се разбира под "безопасност." Не може да имате достъп до нечие друго системно дърво.

Управление на групите нишки

Оставяйки настрана въпросът с безопасността едно нещо за което нишките са полезни е управлението: може да изпълните някои операции върху цялата група нишки с една команда. Следният пример демонстрира това и ограниченията в приоритетите на групите нишки.

Поставените на коментар числа в скоби показват редовете в изхода, които трябва да се погледнат за сравнение.

```
//: c14:ThreadGroup1.java
// How thread groups control priorities
// of the threads inside them.

public class ThreadGroup1 {
    public static void main(String[] args) {
        // Get the system thread & print its Info:
        ThreadGroup sys =
            Thread.currentThread().getThreadGroup();
        sys.list(); // (1)
        // Reduce the system thread group priority:
        sys.setMaxPriority(Thread.MAX_PRIORITY - 1);
        // Increase the main thread priority:
        Thread curr = Thread.currentThread();
        curr.setPriority(curr.getPriority() + 1);
        sys.list(); // (2)
        // Attempt to set a new group to the max:
        ThreadGroup g1 = new ThreadGroup("g1");
        g1.setMaxPriority(Thread.MAX_PRIORITY);
        // Attempt to set a new thread to the max:
        Thread t = new Thread(g1, "A");
        t.setPriority(Thread.MAX_PRIORITY);
        g1.list(); // (3)
        // Reduce g1's max priority, then attempt
        // to increase it:
        g1.setMaxPriority(Thread.MAX_PRIORITY - 2);
        g1.setMaxPriority(Thread.MAX_PRIORITY);
        g1.list(); // (4)
        // Attempt to set a new thread to the max:
        t = new Thread(g1, "B");
        t.setPriority(Thread.MAX_PRIORITY);
        g1.list(); // (5)
        // Lower the max priority below the default
        // thread priority:
        g1.setMaxPriority(Thread.MIN_PRIORITY + 2);
        // Look at a new thread's priority before
        // and after changing it:
        t = new Thread(g1, "C");
        g1.list(); // (6)
        t.setPriority(t.getPriority() - 1);
        g1.list(); // (7)
        // Make g2 a child Threadgroup of g1 and
        // try to increase its priority:
        ThreadGroup g2 = new ThreadGroup(g1, "g2");
        g2.list(); // (8)
        g2.setMaxPriority(Thread.MAX_PRIORITY);
        g2.list(); // (9)
        // Add a bunch of new threads to g2:
        for (int i = 0; i < 5; i++)
            new Thread(g2, Integer.toString(i));
        // Show information about all threadgroups
        // and threads:
        sys.list(); // (10)
        System.out.println("Starting all threads:");
    }
}
```

```

Thread() all = new Thread(sys.activeCount());
sys.enumerate(all);
for(int i = 0; i < all.length; i++)
    if(!all[i].isAlive())
        all[i].start();
// Suspends & Stops all threads in
// this group and its subgroups:
System.out.println("All threads started");
sys.suspend(); // Deprecated in Java 2
// Never gets here...
System.out.println("All threads suspended");
sys.stop(); // Deprecated in Java 2
System.out.println("All threads stopped");
}
} ///:~

```

Следва изходът който е редактиран за да пасне на страницата ([java.lang](#). е маxнато) и са добавени числата които съответстват на числата в скобите по-горе.

```

(1) ThreadGroup(name=system,maxpri=10)
    Thread(main,5,system)
(2) ThreadGroup(name=system,maxpri=9)
    Thread(main,6,system)
(3) ThreadGroup(name=g1,maxpri=9)
    Thread(A,9,g1)
(4) ThreadGroup(name=g1,maxpri=8)
    Thread(A,9,g1)
(5) ThreadGroup(name=g1,maxpri=8)
    Thread(A,9,g1)
    Thread(B,8,g1)
(6) ThreadGroup(name=g1,maxpri=3)
    Thread(A,9,g1)
    Thread(B,8,g1)
    Thread(C,6,g1)
(7) ThreadGroup(name=g1,maxpri=3)
    Thread(A,9,g1)
    Thread(B,8,g1)
    Thread(C,3,g1)
(8) ThreadGroup(name=g2,maxpri=3)
(9) ThreadGroup(name=g2,maxpri=3)
(10) ThreadGroup(name=system,maxpri=9)
    Thread(main,6,system)
    ThreadGroup(name=g1,maxpri=3)
        Thread(A,9,g1)
        Thread(B,8,g1)
        Thread(C,3,g1)
    ThreadGroup(name=g2,maxpri=3)
        Thread(0,6,g2)
        Thread(1,6,g2)
        Thread(2,6,g2)
        Thread(3,6,g2)
        Thread(4,6,g2)
Starting all threads:
All threads started

```

Всички програми имат най-малко една работеща нишка и първото действие в **main()** е да извика **static** метода на **Thread** варечен **currentThread()**. От тази нишка се прави група и се вика **list()** за резултата. Изходът е:

```
(1) ThreadGroup(name=system,maxpri=10)
    Thread(main,5,system)
```

Може да видите че името на главната група нишки е **system** и името на главната нишка е **main** и че принадлежи на **system** грепата нишки.

Второто показва че максималният приоритет на групата **system** може да бъде намален и нишката **main** може да повиши своя приоритет:

```
(2) ThreadGroup(name=system,maxpri=9)
    Thread(main,6,system)
```

Третото създава нова група, **g1**, която автоматично принадлежи към **system** групата понеже не е посочено друго. Нова група **A** се слага в **g1**. След опита да се сложи този приоритет максимален и приоритета на **A** максимален резултатът е:

```
(3) ThreadGroup(name=g1,maxpri=9)
    Thread(A,9,g1)
```

Вижда се, че не е възможно да се повиши максималния приоритет на групата нишки над този на родителската група.

Четвъртото намалява максималния приоритет на **g1** и после се опитва да го повиши до **Thread.MAX_PRIORITY**. Резултатът е:

```
(4) ThreadGroup(name=g1,maxpri=8)
    Thread(A,9,g1)
```

Вижда се, че повишаването не става. Може само да се намалява максималния приоритет на групата, не да се увеличава. Забележете също че приоритета на нишката **A** не се промени, та сега е по-висок от максималния приоритет на групата. Промяната на максималния приоритет на групата не засяга съществуващите нишки.

Петото се опитва да сложи на нова нишка максималния приоритет:

```
(5) ThreadGroup(name=g1,maxpri=8)
    Thread(A,9,g1)
    Thread(B,8,g1)
```

Новата нишка не може да получи приоритет по-голям от максималния групов.

По подразбиране в тази програма приоритетът на нишките е 6; това е приоритетът с който ще се създава нишка и ще остане такъв ако не го променяте. Шестото сваля максималния приоритет на групата под този по подразбиране за да се види какво ще стане като създавате нова нишка при такива условия:

```
(6) ThreadGroup(name=g1,maxpri=3)
    Thread(A,9,g1)
    Thread(B,8,g1)
    Thread(C,6,g1)
```

Макар и максималния приоритет на групата да е 3, новата нишка се създава с този по подразбиране 6. Така максималният приоритет на групата не засяга този по подразбиране. (Фактически, излиза че няма начин да се промени приоритетът по подразбиране за нови нишки.)

След опит за намаляване на приоритета с едно резултатът е:

```
(7) ThreadGroup(name=g1,maxpri=3)
    Thread(A,9,g1)
    Thread(B,8,g1)
    Thread(C,3,g1)
```

Чак когато се опита промяна на приоритета се налага стойността на максималния групов приоритет.

Подобен експеримент е направен в (8) и (9), където нова група нишки **g2** е създадена като дъщерна на **g1** и е променен максималния приоритет. Може да се види, че е невъзможно приоритетът (макс.) на **g2** да надмине този на **g1**:

```
(8) ThreadGroup(name=g2,maxpri=3)
(9) ThreadGroup(name=g2,maxpri=3)
```

Забележете също че **g2** е автоматично поставено на груповия приоритет на **g1** щом **g2** се създаде.

След всички тези експерименти се извежда цялата система групи с приоритетите им:

```
(10) ThreadGroup(name=system,maxpri=9)
    Thread(main,6,system)
    ThreadGroup(name=g1,maxpri=3)
        Thread(A,9,g1)
        Thread(B,8,g1)
        Thread(C,3,g1)
    ThreadGroup(name=g2,maxpri=3)
        Thread(0,6,g2)
        Thread(1,6,g2)
        Thread(2,6,g2)
        Thread(3,6,g2)
        Thread(4,6,g2)
```

Така че поради правилата за групите нишки дъщерната група има винаги приоритет който е по-малък или равен на максималния приоритет на родителската група.

Последната част от този пример демонстрира методи за цялата група нишки. Първо програмата преглежда цялото дърво нишки и стартира тези, които не са стартирани. За интрига **system** групата после е спряна и накрая прекратена. (Макар че е интересно да се види че **suspend()** и **stop()** работят с цялостни групи, ще помните че тези методи са останали в Java 2.) Но когато спрете **system** групата също спирате и **main** нишката и цялата програма спира, така че никога не стига там където нишките са прекратени. Фактически ако прекратите **main** нишката тя изхвърля **ThreadDeath** изключение, така че това не е типично да се прави. Тъй като **ThreadGroup** е наследено от **Object**, който съдържа метода **wait()**, може също да изберете да спрете програмата за всякакъв брой секунди с извикване на **wait(seconds * 1000)**. Това трябва да овладее ключалката в обекта, разбира се.

Класът **ThreadGroup** също има методи **suspend()** и **resume()** така че може да спрете и стартирате цялата група и всички нейни подгрупи с една команда. (Отново, **suspend()** и **resume()** са останали в Java 2.)

Отначало групите нишки могат да изглеждат малко мистериозно, но вие вероятно няма да ги използвате много често.

Runnable отново

По-рано в главата ви съветвах добре да си помислите преди да правите аплет или **Frame** като реализация на **Runnable**. Ако приемете този подход, може да направите само една от онези

нишки във вашата програма. Това ви ограничава ако решите да имате повече от една нишка от същия тип.

Разбира се, ако трябва да наследите от клас и искате да приладете нишково поведение на класа, **Runnable** е точното решение. Последният пример в тази глава експлоатира това чрез правене на **Runnable Canvas** клас който се боядисва в различни цветове. Това приложение е аранжирано да взема параметри от командния ред за да определи колко голяма е мрежата от цветове и колко дълго да **sleep()** между смяната на цветовете. Чрез игра с тези стойности може да откриете някои интересни и неочевидни черти на нишките:

```
//: c14:ColorBoxes.java
// Using the Runnable interface
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class CBox extends JPanel implements Runnable {
    private Thread t;
    private int pause;
    private static final Color[] colors = {
        Color.black, Color.blue, Color.cyan,
        Color.darkGray, Color.gray, Color.green,
        Color.lightGray, Color.magenta,
        Color.orange, Color.pink, Color.red,
        Color.white, Color.yellow
    };
    private Color cColor = newColor();
    private static final Color newColor() {
        return colors(
            (int)(Math.random() * colors.length)
        );
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(cColor);
        Dimension s = getSize();
        g.fillRect(0, 0, s.width, s.height);
    }
    public CBox(int pause) {
        this.pause = pause;
        t = new Thread(this);
        t.start();
    }
    public void run() {
        while(true) {
            cColor = newColor();
            repaint();
            try {
                t.sleep(pause);
            } catch(InterruptedException e) {}
        }
    }
}

public class ColorBoxes extends JFrame {
    public ColorBoxes(int pause, int grid) {
        setTitle("ColorBoxes");
    }
}
```

```

Container cp = getContentPane();
cp.setLayout(new GridLayout(grid, grid));
for (int i = 0; i < grid * grid; i++)
    cp.add(new CBox(pause));
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});
}

public static void main(String[] args) {
    int pause = 50;
    int grid = 8;
    if(args.length > 0)
        pause = Integer.parseInt(args[0]);
    if(args.length > 1)
        grid = Integer.parseInt(args[1]);
    JFrame frame = new ColorBoxes(pause, grid);
    frame.setSize(500, 400);
    frame.setVisible(true);
}
} //:~

```

ColorBoxes е типично приложение чийто конструктор оформя GUI. Този конструктор взема аргумент **int grid** за да зададе **GridLayout** така че той да има **grid** клетки във всяко измерение. После добавя подходящ брой **CBox** обекти за да попълни мрежата, подавайки стойността на **pause** на всеки. В **main()** може да видите как **pause** и **grid** имат стойности по подразбиране които може да промените като зададете аргументи на командния ред.

В **CBox** се върши цялата работа. Той е наследен от **Canvas** и прилага **Runnable** интерфейс така че всеки **Canvas** може също да бъде **Thread**. Помните че когато прилагате **Runnable**, не правите **Thread** обект, само клас който има **run()** метод. Така се налага явно да създадете **Thread** обект и да подадете **Runnable** обект на конструктора, после да извикате **start()** (това става в конструктора). В **CBox** тази нишка е наречена **t**.

Забележете масива **colors**, който номерира всички цветове в класа **Color**. Това се използва в **newColor()** за получаване на случайно избран цвят. Текущия цвят на клетка е **cColor**.

paint() е доста прост – слага цвета **cColor** и запълва цялата основа с него цвят.

В **run()** виждате безкраен цикъл който слага **cColor** да бъде нов случаен цвят и после **repaint()** за да го покаже. После нишката отива да **sleep()** за време зададено в командния ред.

Точно защото този дизайн е гъвкав и нишковостта е налична за всеки **Canvas** елемент може да експериментирате с толкова нишки, колкото желаете. (В реалността има ограничение произтичащо от вашата JVM да може удобно да ги обслужи.)

Тази програма също прави интересен тест за производителност, понеже може да покаже драматичните разлики в скоростта на една реализация на JVM спрямо друга.

ТВЪРДЕ МНОГО НИШКИ

По някое време ще се окаже, че програмата **ColorBoxes** затъва. На моята машина това се случва някъде след 10 x 10 мрежка. Защо става това? Естествено е да предполагате че тук е замесена Swing, така че ето пример за тестване на това предположение чрез създаване на по-малко нишки. Кодът е реорганизиран така, че **ArrayList implements Runnable** и оня **ArrayList**

държе определен брой блокове и избира случајно кой от тях да осъвремени. После се създават няколко такива **ArrayList** обекти, грубо зависещи от блоковете на мрежата. Като резултат имате много по-малко нишки отколкото цветни блокове, така че ако скоростта се покачи ще знаем че е емало твърде много нишки в предишния пример:

```
//: c14:ColorBoxes2.java
// Balancing thread use
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

class CBox2 extends JPanel {
    private static final Color[] colors = {
        Color.black, Color.blue, Color.cyan,
        Color.darkGray, Color.gray, Color.green,
        Color.lightGray, Color.magenta,
        Color.orange, Color.pink, Color.red,
        Color.white, Color.yellow
    };
    private Color cColor = newColor();
    private static final Color newColor() {
        return colors(
            (int)(Math.random() * colors.length)
        );
    }
    void nextColor() {
        cColor = newColor();
        repaint();
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(cColor);
        Dimension s = getSize();
        g.fillRect(0, 0, s.width, s.height);
    }
}

class CBoxVector
    extends ArrayList implements Runnable {
    private Thread t;
    private int pause;
    public CBoxVector(int pause) {
        this.pause = pause;
        t = new Thread(this);
    }
    public void go() { t.start(); }
    public void run() {
        while(true) {
            int i = (int)(Math.random() * size());
            ((CBox2)get(i)).nextColor();
            try {
                t.sleep(pause);
            } catch(InterruptedException e) {}
        }
    }
    public Object last() { return get(size() - 1); }
```

```

}

public class ColorBoxes2 extends JFrame {
    private CBoxVector() v;
    public ColorBoxes2(int pause, int grid) {
        setTitle("ColorBoxes2");
        Container cp = getContentPane();
        cp.setLayout(new GridLayout(grid, grid));
        v = new CBoxVector(grid);
        for(int i = 0; i < grid; i++)
            v(i) = new CBoxVector(pause);
        for (int i = 0; i < grid * grid; i++) {
            v(i % grid).add(new CBox2());
            cp.add((CBox2)v(i % grid).last());
        }
        for(int i = 0; i < grid; i++)
            v(i).go();
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }
    public static void main(String[] args) {
        // Shorter default pause than ColorBoxes:
        int pause = 5;
        int grid = 8;
        if(args.length > 0)
            pause = Integer.parseInt(args(0));
        if(args.length > 1)
            grid = Integer.parseInt(args(1));
        JFrame frame = new ColorBoxes2(pause, grid);
        frame.setSize(500, 400);
        frame.setVisible(true);
    }
} //:~

```

В **ColorBoxes2** се създава масив от **CBoxVector** и се инициализира да държи **grid CBoxVector**и, всеки от които знае колко дълго да спи. Равен брой **Cbox2** обекти после се създава към всеки **CboxVector** и на всеки вектор се казва да **go()**, което стартира нишката му.

CBox2 е подобен на **CBox**: той се боядисва в случаино избран цвят. Но това е *всичко* което прави **CBox2**. Всичкия трединг е преместен в **CBoxVector**.

CBoxVector също би могъл да бъде наследен от **Thread** и да има членове-обекти **ArrayList**. Този дизайн има предимството че **add()** и **get()** методите биха могли после да получат конкретен аргумент и да връщат типовете на стойностите наместо родови **Object**и. (Техните имена също биха могли да се променят на нещо по-късо.) Обаче дизайнът който е тук изглежда на пръв поглед че изисква по-малко код. Освен това той автоматично запазва всичкото останало поведение на **ArrayList**. С всичкия кастинг и скоби необходими за **get()**, това би могло да не бъде случаят при по-голям обем код.

Както преди когато прилагате **Runnable** не получавате всичкото оборудване което идва с **Thread**, така че трябва да създадете **Thread** и да го дадете на конструктора за да има какво да **start()**, както може да видите в **CBoxVector** конструктора и в **go()**. Методът **run()** просто

избира по случаен начин число измежду номерата на векторите и извиква `nextColor()` за да се избере нов случайно избран цвят.

Като пуснете тази програма ще видите че тя разбира се върви по-бързо и отговаря по-бързо (например, като я спрете, тя спира по-бързо), и не затъва толкова бързо. Така се въвежда нов фактор в уравнението за нишките: трябва да следите да няма "твърде много нишки" (каквото се случи да значи това за вашата конкретна система и приложение). Ако това се случи трябва да използвате техники като горната за "балансиране" на броя нишки във вашата програма. Ако срещнете промлеми със скоростта в многонишкова програма трябва да пре-гледате следните неща:

1. Има ли достатъчно извиквания на `sleep()`, `yield()` и/или `wait()`?
2. Достатъчно дълги ли са виканията на `sleep()`?
3. Твърде много нишкш лш сти пуснали?
4. Опитвали ли ста други платформи и JVMи?

Подобни въпроси са причината често програмирането с много нишки да се счита изкуство.

Резюме

Жизнено важно е да се научи кога да се използва мулитрединг и кога да се избягва. Главната причина за използване е ако има няколко задачи които могат да вървят паралелно и така да се използва по-добре компютъра или удобството на потребителя. Класическият пример за балансиране на ресурси е използването на CPUто по време на чакането за I/O. Класическият пример за удобство на потребителя е наблюдаването на "stop" бутона по време на дълги сваляния (от мрежата).

Главните недостатъци на мулитрединга са:

1. Забавяне докато се чака за ресурси
2. Допълнително натоварване на CPU за управление на нишките
3. Неоправдано усложняване, като тъпата идея да има отделна нишка за обновяване на всеки елемент от масив
4. Патологии включващи изтощаване, надбягване и мъртъв блокаж

Допълнително предимство на нишките е че те заменят "леките" превключвания на контекста (от порядъка на 100 инструкции) за "тежки" превключвания на контекста (от порядъка на 1000ди инструкции). Понеже всички нишки в даден процес споделят едно адресно пространство, лекият контекст променя само програмното изпълнение и локалните променливи. От друга страна, смяната на процеса, тежкото превключване, трябва да смени цялото адресно пространство.

Многонишковостта прилича на крачене в изцяло ново пространство и изучаване на нов език, или най-малкото нови концепции. С появата на поддръжка за мулитрединг в повечето микрокомпютърни операционни системи, разширенията за многонишковост се появиха и в много програмни езици и библиотеки. Във всички случаи такова програмиране (1) изглежда мистериозно и изисква промяна в начина на мислене (2) изглежда подобно на поддръжката в други езици, така че като разберете нишките, разбирайте общ език. И макар че поддръжката на нишки може да направи Java да изглежда по-сложен, не обвинявайте Java. Нишките са трудни.

Една от най-големите трудности възниква понеже може повече от една нишка да искат ресурс, като памет на обект, та трябва да се осигури само една нишка да работи с

ресурса едновременно (обикновено критично е писането). Това индуцира използването на ключовата дума **synchronized**, която е полезен инструмент но трябва анапълно да се разбере понеже тихичко може да доведе до ситуации на мъртво блокиране.

Освен това има известно изкуство в прилагането на нишките. Java е проектирана да позволи създаването на колкото обекти трябват за решаването на вашия проблем – поне на теория. (Създаването на милиони обекти за решаване на задачи с крайни елементи, например, може да не е практично с Java.) Обаче изглежда че има горна граница на броя на желаните нишки понеже от известно място нататък работата се забатачва. Тази критична точка не е в обхвата на много хиляди както е с обектите, а някъде в околността на по-малко от 100. Тъй като обикновено се създават шепа нишки за решаване на някакъв проблем, това не е много ограничително, но все пак при по-общо проектиране може да стане преграда.

Значителен неинтуитивен момент във връзка с нишките е че, поради тяхното превключване от планировчика, типично е възможно да направите вашето приложение да работи по-бързо чрез вмъкване на извиквания на **sleep()** вътре в главния цикъл на **run()**. Това определено доказва усещането за изкуство, в частност когато по-големите изчаквания повишават бързината. Разбира се, причината за ускоряването е че честите събуждания - завършвания на **sleep()** довеждат до съответното прекъсване на планировчика, преди нишката да е готова да спи, карайки планировчика да спре каквото върши и после да го продължи за да направи нишката да спи. Изиска се допълнително мислене за да се разбере колко объркано може да бъде всичко това.

Едно нещо което може да ви се стори пропуснато в тази книга е пример с анимация, което е едно от най-популярните за правене с аплет неща. Обаче завършеното решение (със звук) идва с Java JDK (достърен от java.sun.com) в демо секцията. Освен това може да се очаква по-добра поддръжка за анимацията от бъдещи версии на Java, докато напълно различни не-Java, не-програмни решения за анимация във Web се появяват и вероятно ще са по-добри от традиционните подходи. За обяснение как работи анимацията в Java вижте *Core Java* от Cornell & Horstmann, Prentice-Hall 1997. За по-напреднали дискусии за трединга вижте *Concurrent Programming in Java* от Doug Lea, Addison-Wesley 1997, или пък *Java Threads* от Oaks & Wong, O'Reilly 1997.

Упражнения

1. Наследете клас от **Thread** и подтиснете **run()** метода. Вътре в **run()** изведете съобщение, после извикайте **sleep()**. Повторете това три пъти, после завършете **run()**. Сложете стартово съобщение в конструктора и подтиснете **finalize()** да извежда завършващо съобщение. Направете отделен нишков клас който вика **System.gc()** и **System.runFinalization()** вътре в **run()**, извеждайки съобщение че го прави. Направете няколко нишкови обекта от двата типа и ги пуснете да видите какво ще стане.
2. Променете **Counter2.java** така че нишката да е вътрешен клас и да не трябва явно да се помни манипулятор към **Counter2**.
3. Променете **Sharing2.java** като добавите **synchronized** блок вътре в **run()** метода на **TwoCounter** вместо целия **run()** метод да е синхронизиран.
4. Създайте два **Thread** подкласса, един с **run()** който стартира, взема манипулятора на втория **Thread** обект и после вика **wait()**, **run()** на другия клас ще вика **notifyAll()** за първата нишка след определено количество секунди, така че първата нишка да може да изведе съобщение.

5. В **Counter5.java** вътре в **Ticker2** замнете **yield()**-а и обеснете резултатите.
Заменете **yield()** със **sleep()** и обясните резултатите.
6. В **ThreadGroup1.java** заменете викането на **sys.suspend()** с извикване на **wait()** за групата нишки, да чака две секунди. За да работи това коректно трябва да придобиете кючалката на **sys** вътре в **synchronized** блок.
7. Променете **Daemons.java** така че **main()** да има **sleep()** вместо **readLine()**.
Експериментирайте с различни времена на спане да видите какво ще стане.
8. (преходно) В глава 7 намерете примера **GreenhouseControls.java**, който се състои от три файла. В **Event.java** класът **Event** е основан на следенето на времето. Сменете **Event** да бъде **Thread** и променете останалото така че да работи с този **Thread**-базиран **Event**.

15: Разпределена обработка

Исторически погледнато, мрежовото програмиране бе предразполагащо към грешки, трудно и сложно.

Програмистът трябваше да знае много детайли за мрежата и даже за хардуера. Обикновено беше необходимо да се разбираат различни "слоеве" на мрежовия протокол, имаше множество функции във всяка отделна библиотека за мрежите за свързването, пакетирането и разпакетирането на блокове информация; разкарването на тези блокове напред-назад; и протокола по началното свързване. Това беше ужасяваща задача.

Обаче концепцията за мрежите не е толкова трудна. Искате да вземете някаква информация от онази машина ей тук и да я изпратите на онази машина там, или обратно. Това е твърде подобно на четенето и писането на файлове, само дето файловете са на отдалечената машина и освен това тя може да решава какво може и какво не може да правите с тях.

Едно от най-силните места на Java е безболезненото мрежово програмиране. Колкото е възможно подлежащите детайли на мрежата са абстрагирани и взети предвид в JVM и локалната машинна инсталация на Java. Използваният програмен модел е този на файла; фактически обгръщате мрежовата връзка ("socket"-а) със потокови обекти, така че свършвате със същите извиквания на методи както във всеки друг случай на работа с потоци. Освен това вградената многонишковост на Java е извънредно удобна когато се разправяте с друг мрежив въпрос: обслужване на няколко вразки едновременно.

Тази глава запознава с поддръжката в Java за мрежово програмиране с лесни за разбиране примери.

Идентификация на машина

Разбира се за да се говори на друга машина и да е сигурно че тя е точно желаната, трябва да има начин за уникална идентификация на машината в мрежата. Ранните мрежи бяха удовлетворени с единствени имена на машините в локалната мрежа. Обаче Java работи в Internet, което изисква единствена идентификация за всяка машина на света. Това става чрез IP (Internet Protocol) адрес който може да съществува в две форми:

1. Познатата DNS (Domain Name Service) форма. Моето домейново име е **bruceeckel.com**, така че да предположим че имам компютър наречен **Opus** в моя домейн. Неговото домейново име би било **Opus.bruceeckel.com**. това е точно име от вида в който изпращате електронна поща на хората и е често вместено във World-Wide-Web адрес.
2. Алтернативно може да използвате формата "dotted quad", която е четири числа разделени с точки като **123.255.28.120**.

В двета случая IP е представен вътрешно като 32-битово число¹ (така че всяко от четирите числа не може да надмине 255), а може да вземете специален Java обект за представяне на адреса в която и да е форма чрез **static InetAddress.getByName()** метода който е в **java.net**. резултатът е от типа **InetAddress** който може да използвате за направа на "socket" както ще видите по-късно.

Като прост пример на използване на **InetAddress.getByName()** да видим какво става когато имате връзка по комутируема линия с Internet service provider (ISP). Всеки път когато се свържете ви се присвоява временен IP адрес. Но докато сте свързани вашият IP е точно толкова валиден колкото всеки друг IP адрес в Internet. Ако някой друг се свърже с вашата машина чрез този IP тогава може да се свърже с Web и/или FTP сървъра който работи на вашата машина. Разбира се, те трябва да знаят вашия IP адрес, а понеже тай се присвоява всеки път когато наберете, как могат да го знаят?

Следващата програма използва **InetAddress.getByName()** за да произведе вашия IP адрес. За да го използвате трябва да знаете името на вашия компютър. Това е тествано само с Windows 95, тома може да кликнете "Settings," "Control Panel," "Network," а после да изберете "Identification". "Computer name" е името което да слежите на командния ред.

```
//: c15:WhoAmI.java
// Finds out your network address when you're
// connected to the Internet.
package c15;
import java.net.*;

public class WhoAmI {
    public static void main(String[] args)
        throws Exception {
        if(args.length != 1) {
            System.err.println(
                "Usage: WhoAmI MachineName");
            System.exit(1);
        }
        InetAddress a =
            InetAddress.getByName(args[0]);
        System.out.println(a);
    }
} ///:~
```

В моя случай машината е наречена "Colossus" (от филмчето съ същото име, понеже продължавам да слагам все по-големи дискове на няя). Щом веднъж съм се свързал към моя ISP пускам програмата:

```
| java WhoAmI Colossus
```

Получавам съобщение като това (разбира се, адресът е различен всеки път):

```
| Colossus/199.190.87.75
```

Ако кажа на мой приятел този адрес, той може да влезе в моя персонален Web сървър чрез избиране на URL <http://199.190.87.75> (през тази моя сесия само). Това понякога е лесен начин за предаване на информация на някого или за тестване на конфигурацията на Web сайт преди да се изпрати на "реален" сървър.

¹ This means a maximum of just over [4four](#) billion numbers, which is rapidly running out. The new standard for IP addresses will use a 128-bit number, which should produce enough unique IP addresses for the foreseeable future.

Сървъри и клиенти

Работата на мрежата като цяло е да позволи на две машини да се свържат и да си говорят. Щом веднъж машините са свързани те могат да имат приятен двупосочен разговор. Но как да се намерят една друга? Това прилика на загубване в увеселителен парк: едната машина трябва да стои на едно място и да слуша докато другата машина каже, "Ей, къде си?"

Машината която "стои на едно място" се нарича *server*, а тази която търси е *client*. Тази разлика е важна само докато се намерят. Щом веднъж се свържат, започва разговор и няма значение кой (от двамата) е бил сървърът и кой се е случило да е клиентът.

Така че работата на сървъра е да слуша за свързване, а това става със специален сървърски обект който създавате. Работата на клиента е да се опита да се свърже със сървъра, а това става със специален клиентски обект който създавате. Щом веднъж връзката е направена, ще видите че и сървърът и клиентът завършват, връзката магически се превръща в IO потоков обект, и от там нанатък може да третирате връзката като писане и четене във/от файл. Така щом имате връзката ще използвате познатите IO команди от глава 10. Това е една от добrите страни на мрежовото програмиране на Java.

Тестване на програми без мрежа

По много причини може да нямаете клиентска машина, сървърска машина и мрежа за да тествате вашата програма. Бихте могли да правите упражненията в класна стая, или може да правите програми които още не са толкова стабилни, че да ги сложите в мрежа. Създателите на Internet Protocol знаеха за този въпрос и създадоха специален адрес наречен **localhost** за "локална обратна връзка" - IP за тестване без мрежа. Родовия начин да се произведе този адрес в Java е:

```
| InetAddress addr = InetAddress.getByName(null);
```

Ако дадете на **getByName()** стойност **null**, по подразбиране той използва **localhost**. **InetAddress** използвате за позоваване на тази машина и трябва да направите такъв преди каквото и да е друго. Не може да се пипа съдържанието на **InetAddress** (но може да се извежда, както ще видите в следващия пример). Единствения начин да се създаде **InetAddress** е чрез някой от **static** членовете-методи **getByName()** (който обикновено ще използвате), **getAllByName()**, или **getLocalHost()**.

Може също да направите локалния адрес чрез даване на стринга **localhost**:

```
| InetAddress.getByName("localhost");
```

Или чрез използване на другата форма на IP за обратната връзка:

```
| InetAddress.getByName("127.0.0.1");
```

И трите форми дават един и същ резултат.

Port: УНИКАЛНО МЯСТО В МАШИНАТА

IP адресът не е достатъчен за идентификация, понеже много сървъри могат да съществуват на една машина. Всяка IP машина също съдържа port-ове, та когато настройвате клиента трябва да посочите порт на който и той и сървърът са съгласни да говорят; ако вие срещате някого, IP адресът е околността и портът е бара.

Портът не е физическо място на машината, а софтуерна абстракция (главно за счетоводни цели). Клиентската програма знае как да се свърже чрез IP адреса, но как се свързва с желаната услуга (една от многоот на машината)? За това служат номерата на портовете като втори порядък адресиране. Изята е че ако питате за конкретен порт, вие искате услугата която е асоциирана с него. Времето (астр.) е пример за услуга. Типично

всяка услуга се свързва с уникален порт на всяка сървърска машина. Работа на клиента е да знае кой е портът с търсената услуга.

Системните услуги резервираят портове от 1 до 1024, така че няма да ги използвате, както и който и да е друг за който знаете че е в употреба. Първият избор в тази книга ще е порт 8080 (в памет на почтения стар 8-bit Intel 8080 чип в моя първи компютър, една CP/M машина).

Socket-и

Socket-ът е софтуерната абстракция за представяне на "краищата" на връзката между две машини. За дадена връзка има сокет (ще употребяваме това наместо цокъл - б.пр.), може да си представяте "кабел" между двете машини и краищата на "кабела" закачени в сокетите. Разбира се, физическите кабели и пр. на машината са абсолютно неизвестни. Цялата работа с абстракцията е да не трябва да знаете много.

В Java създавате сокет за да се свържете с друга машина, после вземате **InputStream** и **OutputStream** (или, с подходящи преобразуватели, **Reader** и **Writer**) от сокета за да можете да третирате като IO потоков обект. Има два потоково базирани класа: **ServerSocket** който сървърът използва за "слушане" за искания за включване и **Socket** който клиентът използва за започване на връзката. Щом клиентът направи връзката със сокета, **ServerSocket** връща (чрез метода **accept()**) кореспондентния **Socket** чрез който ще става комуникацията. От там нататък имате истинска **Socket** към **Socket** връзка и третирате двата края еднакво понеже те са еднакви. В тази точка използвате методите **getInputStream()** и **getOutputStream()** за получаване на кореспондентните **InputStream** и **OutputStream** обекти за всеки **Socket**. Тези трябва да са обградени в буфери и форматирани класове точно както всички потоци описани в глава 10.

Използването на термина **ServerSocket** би изглеждало като друг пример на смущаващи имена в Java библиотеките. Бихте могли да си помислите че **ServerSocket** щеше да е по-добре да се нарече "ServerConnector" или нещо без "Socket". Бихте могли също да мислите че **ServerSocket** и **Socket** трябва и двата да са наследени от един базов клас. Разбира се, двата класа имат няколко подобни метода, но не достатъчно, че да имат общ базов клас. Напротив, работата на **ServerSocket** е да чака докато другата машина се свърже, после да върне фактическия **Socket**. Затова името **ServerSocket** изглежда малко сбъркано, понеже неговата работа не е да бъде сокет а да направи **Socket** обект когато някой друг се свърже с него.

Обаче **ServerSocket** не създава физически "сървърски" или слушащ сокет на хоста. Този сокет слуша за заявки за връзка и после връща "основан" сокет (с определени крайни точки) чрез метода **accept()**. Смущаващата част е че и двата сокета (слушащия и сега съставения) са асоциирани с един и същ сървърски сокет. Слушащия сокет може да приема само заявки и не може да приема данни. Така че докато **ServerSocket** няма много програмистки смисъл, той има "физически" такъв.

Когато създавате **ServerSocket** му давате само номер на порт. Няма нужда от IP адрес понеже вече е на машината която представя. Когато създавате **Socket**, обаче, трябва да му дадете и IP адрес и номер на порт с който се опитвате да се свържете. (От друга страна, **Socket**-ът който се връща обратно към **ServerSocket.accept()** въобще съдържа тази информация.)

ПРОСТИ СЪРВЪР И КЛИЕНТ

Този пример показва най-простото използване на сървър и клиент използващи сокети. Всичкото което сървърът прави е да чака за връзка, после използва **Socket** създадена от тази връзка за да създаде **InputStream** и **OutputStream**. След това всичкото което чете от **InputStream** дава (прави ехо) на **OutputStream** докато получи реда END, когато затваря връзката.

Клиентът прави връзка със сървъра, после създава **OutputStream**. Редове от текст се изпращат по **OutputStream**. Клиентът също създава **InputStream** за да чуе какво казва сървъра (което в този случай е само ехото).

И сървърът и клиентът използват един и същ номер на порт и клиентът използва локалния адрес за обратна връзка за да се свърже със сървъра на същата машина така че не трябва да се тества през мрежа. (За някои конфигурации обаче може да трябва да сте свързани към мрежа за да работи програмата, макар и да няма връзки през мрежата.)

Ето сървъра:

```
//: c15:JabberServer.java
// Very simple server that just
// echoes whatever the client sends.
import java.io.*;
import java.net.*;

public class JabberServer {
    // Choose a port outside of the range 1-1024:
    public static final int PORT = 8080;
    public static void main(String[] args)
        throws IOException {
        ServerSocket s = new ServerSocket(PORT);
        System.out.println("Started: " + s);
        try {
            // Blocks until a connection occurs:
            Socket socket = s.accept();
            try {
                System.out.println(
                    "Connection accepted: " + socket);
                BufferedReader in =
                    new BufferedReader(
                        new InputStreamReader(
                            socket.getInputStream()));
                // Output is automatically flushed
                // by PrintWriter:
                PrintWriter out =
                    new PrintWriter(
                        new BufferedWriter(
                            new OutputStreamWriter(
                                socket.getOutputStream())),true);
                while (true) {
                    String str = in.readLine();
                    if (str.equals("END")) break;
                    System.out.println("Echoing: " + str);
                    out.println(str);
                }
                // Always close the two sockets...
            } finally {
                System.out.println("closing...");
                socket.close();
            }
        } finally {
            s.close();
        }
    }
} ///:~
```

Може да се види че **ServerSocket** иска само номер на порт, не иска IP адрес (щом работи на тази машина!). Когато извикате **accept()**, методът блокира докато някой клиент се опита да се свърже с него. Тоест чака за връзка, но други процеси могат да вървят (виж глава 14). Когато е направена връзка, **accept()** завършва със **Socket** обект представящ връзката.

Въпросът с почистването на сокетите е много тънък тук. Ако **ServerSocket** конструктора се провали, програмата просто свършва (забележете че трябва да предполагаме че конструкторът за **ServerSocket** не оставя отворени връзки ако се провали). За този случай, **main() throws IOException** така че **try** не е нужен. Ако **ServerSocket** конструкторът е успешен всички други методи трябва да бъдат защитени с **try-finally** блок, за да се осигури че, без значение как е останал сокетът, **ServerSocket** е правилно затворен.

Същата логика се използва за **Socket**-а върнат от **accept()**. Ако **accept()** се провали трябва да смятаме че **Socket** не съществува и не държи ресурси, така че не е необходимо да се почиства. Ако той е успешен, обаче, следните оператори трябва да са в **try-finally** така че ако не са успешни **Socket** все пак да бъде почищен. Тук е нужно внимание понеже сокетите са важни ресурси, различни от паметта, така че трябва интелигентно да се работи с тях (понеже няма деструктори в Java да го направят заради вас).

Както **ServerSocket** така и **Socket** създадени с **accept()** се извеждат към **System.out**. Това значи че техните **toString()** методи автоматично се викат. Получава се:

```
ServerSocket(addr=0.0.0.0,PORT=0,localport=8080)  
Socket(addr=127.0.0.1,PORT=1077,localport=8080)
```

Накратко, вижда се как си подхожда с това което прави клиентът.

Следващата част от програмата изглежда точно като отваряне на файлове за писане и четене само дето **InputStream** и **OutputStream** са създадени от **Socket** обекта. И **InputStream** и **OutputStream** са преобразувани в Java 1.1 **Reader** и **Writer** обекти чрез класове-“конвертори” **InputStreamReader** и **OutputStreamWriter**, респективно. Би могло също да се използват и класовете **InputStream** и **OutputStream** на Java 1.0 направо, но за изхода има значително предимство използването на **Writer** подхода. Това е с **PrintWriter**, който има претоварен конструктор който взема втори аргумент, **boolean** флаг който показва дали автоматично да се изпразва буфера след всеки **println()** (но не **print()**) оператор. Всеки път когато пишете на **out**, неговият буфер трябва да се изпразни та информацията да отиде по мрежата. Изпразването е важно за този конкретен пример понеже сървърът и клиентът се изчакват един друг да изпратят ред за да продължат. Ако не се изпразва буфера, информацията няма да се изпрати докато той се напълни, което причинява множество проблеми в този пример.

Когато пишете мрежови програми трябва да бъдете внимателни с автоматичното изпразване на буфера. Всеки път когато се изпразва буферът трябва да се създаде и изпрати пакет. В този случай това е, което искаме, понеже ако не се изпрати пакет съдържащ реда тогава разговорът назад-напред между клиента и сървъра ще спре. С други думи казано, краят на реда е край на съобщението. Но в много случаи съобщенията не са на редове и е по-добре да се остави вградения механизъм да решава кога да изпразва буфера. По този начин може да се изпращат по-дълги пакети и процесът ще бъде по-бърз.

Забележете че както повечето потоци с които работим и тия са буферираны. В края на главата има упражнение в което ще видите какво ще стане ако не са (нещата се забавят).

Безкрайния **while** цикъл чете редове от **BufferedReader in** и извежда информацията към **System.out** и към **PrintWriter out**. Забележете че това биха могли да бъдат кои да е потоци, просто се е случило да са свързани към мрежа.

Когато клиентът изпрати ред състоящ се от “END” програмата излиза от цикъла и затваря **Socket**-а.

Ето клиента:

```

//: c15:JabberClient.java
// Very simple client that just sends
// lines to the server and reads lines
// that the server sends.
import java.net.*;
import java.io.*;

public class JabberClient {
    public static void main(String[] args)
        throws IOException {
        // Passing null to getByName() produces the
        // special "Local Loopback" IP address, for
        // testing on one machine w/o a network:
        InetAddress addr =
            InetAddress.getByName(null);
        // Alternatively, you can use
        // the address or name:
        // InetAddress addr =
        //     InetAddress.getByName("127.0.0.1");
        // InetAddress addr =
        //     InetAddress.getByName("localhost");
        System.out.println("addr = " + addr);
        Socket socket =
            new Socket(addr, JabberServer.PORT);
        // Guard everything in a try-finally to make
        // sure that the socket is closed:
        try {
            System.out.println("socket = " + socket);
            BufferedReader in =
                new BufferedReader(
                    new InputStreamReader(
                        socket.getInputStream()));
            // Output is automatically flushed
            // by PrintWriter:
            PrintWriter out =
                new PrintWriter(
                    new BufferedWriter(
                        new OutputStreamWriter(
                            socket.getOutputStream())),true);
            for(int i = 0; i < 10; i++) {
                out.println("howdy " + i);
                String str = in.readLine();
                System.out.println(str);
            }
            out.println("END");
        } finally {
            System.out.println("closing...");
            socket.close();
        }
    }
} ///:~

```

В **main()** може да се видят всичките три начина за получаване на **InetAddress** от локалния IP адрес за обратна връзка: чрез **null**, **localhost**, или чрез явния резервиран адрес **127.0.0.1**. Разбира се, ако искате да се свържете с машина през мрежата, слагате IP адреса на нея машина. Когато **InetAddress addr** се изведе (чрез автоматично викане на неговия **toString()** метод) резултатът е:

```
| localhost/127.0.0.1
```

Чрез подаване на **getByName()** на **null**, то взема стойността по подразбиране **localhost**, а това дава специалния адрес **127.0.0.1**.

Забележете че **Socket**-а наречен **socket** е създаден и с **InetAddress** и с номер на порт. За да разберете какво значи това когато извеждате някой от **Socket** обектите, помнете че Internet връзката е напълно определена от тези четири данни: **clientHost**, **clientPortNumber**, **serverHost** и **serverPortNumber**. Когато сървърът се задейства, той взема присвоения му порт (8080) на локалната машина (127.0.0.1). Когато клиентът тръгва, той взема следващия номер на порт, 1077 в този случай, което се случва да бъде на същата машина (127.0.0.1) като сървъра. Сега за да може да се движат данните между клиента и сървъра, всяка страна трябва да знае къде да ги праща. Затова по време на процеса на свързване към "познат" сървър клиентът изпраща "адрес на връщане" така че сървърът знае къде да изпраща данните. Това се вижда в примерния изход от сървърската страна:

```
| Socket(addr=127.0.0.1,port=1077,localport=8080)
```

Това значи че сървърът току-що е приел връзката от 127.0.0.1 на порт 1077 докато е слушал на неговия локален порт (8080). На клиентската страна:

```
| Socket(addr=localhost/127.0.0.1,PORT=8080,localport=1077)
```

Което значи че сървърът се е свързал към 127.0.0.1 на порт 8080 използвайки локален порт 1077.

Ще забележите че всеки път когато наново пускате клиента с единица се увеличава номерът на локалния порт. Той започва от 1025 (единица след резервираните) и продължава да расте докато не се бутстрапира машината наново, когато започва пак от 1025. (На UNIX машини след като се достигне най-големия позволен номер, започва се пак от началото.)

Щом веднъж **Socket** обектът е бил създаден, процесът на превръщане в **BufferedReader** и **PrintWriter** е същият като за сървъра (отново, и в двата случая започвате със **Socket**). Тук клиентът започва разговора с изпращане на "howdy" следвано от номер. Забележете че буферът пак трябва дасе изпразни (което става автоматично чрез втория аргумент на конструктора на **PrintWriter**). Ако буферът не се изпразни, челия разговор зависва понеже "howdy"-то никога няма да се изпрати буферът не е достатъчно пълен, та да се изпразни автоматично). Всеки ред изпратен обратно към сървъра се праща на **System.out** за свидетелство че всичко работи коректно. За да се свърши разговора, предварително определеното "END" се изпраща. Ако клиентът просто затвори, сървърът изхвърля изключение.

Може да се види, че е обрнато същото внимание на освобождаването на ресурсите заети от **Socket**-а чрез **try-finally** блок.

Сокетите дават "посветена" връзка която съществува докато не бъде явно прекъсната. (Посветената връзка може все пак да се прекъсне неявно ако едната страна или пък междинна връзка се скапе.) Това значи че и двете страни са непрекъснато заети и връзката е непрекъснато отворена. Това изглежда като локален подход в мрежите, но слега допълнително натоварване върху мрежата. По-късно в тази глава ще видите алтернативен подход в мрежите, при който връзките са само временни.

Обслужване на много клиенти

JabberServer работи, но може да обслужва само един клиен едновременно. С типичен сървър ще е необходимо да могат да се обслужват много клиенти едновременно. Отговорът е многонишковост и за езиците които не го поддържат директно това означава всички видове усложнения. В глава 14 видяхте че многонишковостта в Java е почти толкова проста колкото е възможно, считайки че мултитредингът е малко сложна тема. Понеже мултитрединга в Java е

достатъчно праволинеен, правенето на сървър който обслужва много клиенти е относително просто.

Основната схема е да се направи единствен **ServerSocket** в сървъра и да се вика **accept()** да чака за нова връзка. Когато **accept()** завърши, вземате създадения **Socket** и го използвате за създаване на нишка чиято задача е да обслужва него конкретен клиент. После викате **accept()** пак да чака за нов клиент.

Може да се види, че следващият сървърски код изглежда подобно на **JabberServer.java** примера освен че всичките операции за обслужване на конкретен клиент са преместени в отделен клас-нишка:

```
//: c15:MultiJabberServer.java
// A server that uses multithreading to handle
// any number of clients.
import java.io.*;
import java.net.*;

class ServeOneJabber extends Thread {
    private Socket socket;
    private BufferedReader in;
    private PrintWriter out;
    public ServeOneJabber(Socket s)
        throws IOException {
        socket = s;
        in =
            new BufferedReader(
                new InputStreamReader(
                    socket.getInputStream()));
        // Enable auto-flush:
        out =
            new PrintWriter(
                new BufferedWriter(
                    new OutputStreamWriter(
                        socket.getOutputStream())), true);
        // If any of the above calls throw an
        // exception, the caller is responsible for
        // closing the socket. Otherwise the thread
        // will close it.
        start(); // Calls run()
    }
    public void run() {
        try {
            while (true) {
                String str = in.readLine();
                if (str.equals("END")) break;
                System.out.println("Echoing: " + str);
                out.println(str);
            }
            System.out.println("closing...");
        } catch (IOException e) {
        } finally {
            try {
                socket.close();
            } catch(IOException e) {}
        }
    }
}
```

```

}

public class MultiJabberServer {
    static final int PORT = 8080;
    public static void main(String[] args)
        throws IOException {
        ServerSocket s = new ServerSocket(PORT);
        System.out.println("Server Started");
        try {
            while(true) {
                // Blocks until a connection occurs:
                Socket socket = s.accept();
                try {
                    new ServeOneJabber(socket);
                } catch(IOException e) {
                    // If it fails, close the socket,
                    // otherwise the thread will close it:
                    socket.close();
                }
            }
        } finally {
            s.close();
        }
    }
} //:~

```

Нишката **ServeOneJabber** взема **Socket** обекта създаден от **accept()** в **main()** всеки път когато се създава връзка. После, както и преди, се създава **BufferedReader** и автоматично изпразван **PrintWriter** обект с използване на **Socket**. Накрая се вика специалния **Thread** метод **start()**, който изпълнява инициализацията на нишката и после вика **run()**. Това прдизвиква същите видове действие както и в предишния пример: четене на нещо от сокета и връщането им като ехо докато дойде сигнал "END".

Отговорността за почистването на сокета трябва отново да бъде грижливо поета. В този случай сокетът е създаден извън **ServeOneJabber** така че отговорността може да бъде споделена. Ако конструкторът на **ServeOneJabber** се провали, той просто ще изхвърли изключение към извикващия, който тогава ще почисти нишката. Ако обаче конструкторът успее, **ServeOneJabber** взема отговорността за почистването, в неговия **run()**.

Забележете простотата на **MultiJabberServer**. Както преди се създава **ServerSocket** и се вика **accept()** за да се позволи нова връзка. Но този път връщаната стойност от **accept()** (един **Socket**) се дава на конструктора за **ServeOneJabber**, който създава нова нишка за обслужване на връзката. Когато връзката се прекрати, нишката просто заминава.

Ако създаването на **ServerSocket** се провали, отново се изхвърля изключение чрез **main()**. Но ако успее, външният **try-finally** гарантира почистването. Вътрешният **try-catch** пази само от проваляне на **ServeOneJabber** конструктора; ако конструкторът успее, нишката **ServeOneJabber** ще затвори асоциирания сокет.

За да пробваме дали сървърът наистина поддържа много клиенти, следващата програма създава няколко (използвайки нишки) които се свързват към същия сървър. Всяка нишка има ограничено време на живот и когато загине, отваря се място за нова нишка. Най-големият позволен брой нишки се определя от **final int maxthreads**. Ще забележите че тази стойност е критична, понеже ако я направите твърде голяма види се нишките почват да не им стигат ресурсите и програмата мистериозно се скапва.

```

//: c15:MultiJabberClient.java
// Client that tests the MultiJabberServer

```

```

// by starting up multiple clients.
import java.net.*;
import java.io.*;

class JabberClientThread extends Thread {
    private Socket socket;
    private BufferedReader in;
    private PrintWriter out;
    private static int counter = 0;
    private int id = counter++;
    private static int threadcount = 0;
    public static int threadCount() {
        return threadcount;
    }
    public JabberClientThread(InetAddress addr) {
        System.out.println("Making client " + id);
        threadcount++;
        try {
            socket =
                new Socket(addr, MultiJabberServer.PORT);
        } catch(IOException e) {
            // If the creation of the socket fails,
            // nothing needs to be cleaned up.
        }
        try {
            in =
                new BufferedReader(
                    new InputStreamReader(
                        socket.getInputStream()));
            // Enable auto-flush:
            out =
                new PrintWriter(
                    new BufferedWriter(
                        new OutputStreamWriter(
                            socket.getOutputStream())), true);
            start();
        } catch(IOException e) {
            // The socket should be closed on any
            // failures other than the socket
            // constructor:
            try {
                socket.close();
            } catch(IOException e2) {}
        }
        // Otherwise the socket will be closed by
        // the run() method of the thread.
    }
    public void run() {
        try {
            for(int i = 0; i < 25; i++) {
                out.println("Client " + id + ":" + i);
                String str = in.readLine();
                System.out.println(str);
            }
            out.println("END");
        } catch(IOException e) {
        } finally {

```

```

// Always close it:
try {
    socket.close();
} catch(IOException e) {}
threadcount--; // Ending this thread
}
}
}

public class MultiJabberClient {
    static final int MAX_THREADS = 40;
    public static void main(String[] args)
        throws IOException, InterruptedException {
        InetAddress addr =
            InetAddress.getByName(null);
        while(true) {
            if(JabberClientThread.threadCount()
                < MAX_THREADS)
                new JabberClientThread(addr);
            Thread.currentThread().sleep(100);
        }
    }
} //:~

```

Конструкторът на **JabberClientThread** взема **InetAddress** и го използва за отваряне на **Socket**. Вероятно започвате да виждате схемата: **Socket**-ът се използва винаги за създаване на някакъв вид **Reader** и/или **Writer** (или **InputStream** и/или **OutputStream**) обект, което е единствения начин по който може да се използва **Socket**. (Може, разбира се, да направите един или два класа за автоматизиране на този процес ако стане досадно.) Пак, **start()** изпълнява инициализацията на нишките и вика **run()**. Тук се изпращат съобщения на сървъра и се прави ехото им на екрана. Обаче нишката има ограничено време на живот и накрая свършва. Забележете че сокетът се почиства ако конструкторът е неуспешен след като съкетът е създаден но преди завършването на конструктора. В останалото време отговорността за викане на **close()** за сокета е делегирана на **run()** метода.

threadcount следи колко **JabberClientThread** обекта съществуват в момента. Той се инкрементира в конструктора и декрементира щом завърши **run()** (което значи че нишката се прекратява). В **MultiJabberClient.main()** може да видите че се тества броят на нишките и ако има твърде много не се създават повече. После методът спи. По този начин когато завършват някои нишки други могат да се създават. Може да експериментирате с **MAX_THREADS** за да видите с колко връзки вашата конкретна машина започва да се затруднява.

Датаграми

Примерите които видяхте до тук използват *Transmission Control Protocol* (TCP, също известен като *stream-based sockets*), който е проектиран за много добра надеждност и гарантира, че данните ще пристигнат. Той позволява препредаване на загубените данни, дава резервни пътища за в случай на откази, а също така байтовете се доставят в реда в който са изпратени. Всичко това има цена: TCP товари значително.

Има втори протокол наречен *User Datagram Protocol* (UDP), който не гарантира че пакетите ще се доставят и че ще дойдат в реда в който са били изпратени. Той се нарича "ненадежден протокол" (TCP е "надежден протокол"), което звучи лошо, но понеже е много по-бърз той може да бъде полезен. Има някои приложения, като например звуков сигнал, където не е много важно дали няколко пакета ще се разсипят по пътя, но бързината е жизнено важна. Или да вземем система за точно време, където наистина няма значение че ще се изгуби едно от

съобщенията. Също някои приложения могат да са в състояние да изпратят UDP съобщение към сървъра и след като нямат отговор в определен интервал от време, да считат че е било изгубено.

Поддръжката на датаграми в Java създава същото усещане като поддръжката на TCP сокети, но има значителни разлики. При датаграмите слагате **DatagramSocket** и на клиента и на сървъра, но няма аналогия с **ServerSocket** който очаква връзка. Така е защото няма "връзка," а датаграма която показва. Друга фундаментална разлика е че с TCP сокетите щом веднъж сте създали връзката няма нужда да се тревожите кой с кого говори; просто изпращате данните насам и нататък по потоците. С датаграмите обаче пакетът трябва да знае откъде идва и къде трябва да отиде. Това значи че трябва да знаете тези неща за всеки пакет който идва или изпращате.

DatagramSocket изпраща и приема пакетите и **DatagramPacket** съдържа информацията. Когато приемате датаграма трябва само да дадете буфер където данните ще се сложат; информацията за Internet адреса и номера на порта от където информацията е дошла ще бъдат автоматично инициализирани когато пакетът пристигне от **DatagramSocket**. Така че конструкторът за **DatagramPacket** за приемане на датаграми е:

| `DatagramPacket(buf, buf.length)`

където **buf** е масив от **byte**. Понеже **buf** е масив може да се зачудите защо конструкторът сам не определи дължината на масива. Аз се чудих и мога само да предположа че това са следи от С-стила на програмиране, където разбира се масивите не могат да ви кажат колко са големи.

Може да използвате повторно приеманата датаграма; не е необходимо да правите нова всеки път. Всеки път когато я използвате данните в буфера се презаписват.

Максималната дължина на буфера се ограничава само от допустимата дължина на пакета на датаграмата, тоест до малко по-малко от 64Kbytes. Но в много приложения ще я искате много по-малка, особено когато изпращате данни. Каква дължина ще се избере зависи от конкретното приложение.

Когато изпращате датаграма **DatagramPacket** трябва да съдържа не само данните, но също Internet и порта към които се изпраща. Така конструкторът за заминаваш **DatagramPacket** е:

| `DatagramPacket(buf, length, inetAddress, port)`

Този път **buf** (който е масив от **byte**) вече съдържа данните които ще изпращате. **length** би могла да бъде дължината на **buf**, но също и по-малка, показваща колко байта искате да се изпратят. Другите два аргумента са Internet адреса където пакетът ще ходи и портът на назначението на далечната машина.²

Може да се помисли че двата конструктора създават два различни обекта: за приемане и един за изпращане на датаграми. Един добър ОО дизайн би се спрял на два отделни класа, а не на един който става различен в зависимост от конструктора си. Това вероятно е истина, но за щастие използването на **DatagramPacket**ите е достатъчно просто и проблемът не е интересен както ще се види в следващия пример. Той е подобен на **MultiJabberServer** и **MultiJabberClient** примера за TCP сокетите. Много клиенти ще изпращат датаграми до сървъра, който ще прави echo на изпращащия клиент.

За опростяване създаването на **DatagramPacket** от **String** и обратно примерът започва със спомагателен клас, **Dgram**, който да върши тази работа:

```
//: c15:Dgram.java
// A utility class to convert back and forth
```

² TCP and UDP ports are considered unique. That is, you can simultaneously run a TCP and UDP server on port 8080 without interference.

```

// Between Strings and DatagramPackets.
import java.net.*;

public class Dgram {
    public static DatagramPacket toDatagram(
        String s, InetAddress destIA, int destPort) {
        // Deprecated in Java 1.1, but it works:
        byte[] buf = new byte(s.length() + 1);
        s.getBytes(0, s.length(), buf, 0);
        // The correct Java 1.1 approach, but it's
        // Broken (it truncates the String):
        // byte[] buf = s.getBytes();
        return new DatagramPacket(buf, buf.length,
            destIA, destPort);
    }
    public static String toString(DatagramPacket p){
        // The Java 1.0 approach:
        // return new String(p.getData(),
        // 0, 0, p.getLength());
        // The Java 1.1 approach:
        return
            new String(p.getData(), 0, p.getLength());
    }
} ///:~

```

Първият метод на **Dgram** взема **String**, **InetAddress** и номер на порт и изгражда **DatagramPacket** чрез копиране на съдържанието на **String** в **byte** буфер и подаването на та буфера на конструктора на **DatagramPacket**. Забележете “+1”-та в алокирането на буфера – това е необходимо за да се предотврати отрязване. Методът **getBytes()** от **String** е специална операция която копира **char**кътири от **String** в **byte** буфер. Този метод сега е остатъл; Java 1.1 има “по-добър” метод да се направи това, но той е поставен на коментар понеже отрязва **Stringa**. Така че ще получите съобщение когато компилирате под Java 1.1, но поведението ще е коректно. (Този бъг може и да е оправен когато четете това.)

Методът **Dgram.toString()** показва подхода и при Java 1.0 и при Java 1.1 (който е различен понеже има нов вид **String** конструктор).

Ето сървъра за демонстрация на датаграмите:

```

//: c15:ChatterServer.java
// A server that echoes datagrams
import java.net.*;
import java.io.*;
import java.util.*;

public class ChatterServer {
    static final int INPORT = 1711;
    private byte[] buf = new byte(1000);
    private DatagramPacket dp =
        new DatagramPacket(buf, buf.length);
    // Can listen & send on the same socket:
    private DatagramSocket socket;

    public ChatterServer() {
        try {
            socket = new DatagramSocket(INPORT);
            System.out.println("Server started");

```

```

while(true) {
    // Block until a datagram appears:
    socket.receive(dp);
    String rcvd = Dgram.toString(dp) +
        ", from address: " + dp.getAddress() +
        ", port: " + dp.getPort();
    System.out.println(rcvd);
    String echoString =
        "Echoed: " + rcvd;
    // Extract the address and port from the
    // received datagram to find out where to
    // send it back:
    DatagramPacket echo =
        Dgram.toDatagram(echoString,
            dp.getAddress(), dp.getPort());
    socket.send(echo);
}
} catch(SocketException e) {
    System.err.println("Can't open socket");
    System.exit(1);
} catch(IOException e) {
    System.err.println("Communication error");
    e.printStackTrace();
}
}

public static void main(String[] args) {
    new ChatterServer();
}
} //:~

```

ChatterServer-ът съдържа единствен **DatagramSocket** за приемане на съобщения, вместо да се създава всеки път когато приемате съобщение. Единственият **DatagramSocket** може да бъде използван пак и пак. Този **DatagramSocket** има номер на порт понеже това е сървърът и клиентът трябва да знае точния адрес където да прати датаграмата. Има номер на порт но не и Internet адрес понеже е резидентен на “тази” машина така че знае нейния Internet адрес (в този случай дефолтния **localhost**). В безкрайния **while** цикъл на **socket** се казва да **receive()**, при което той изчаква докато датаграмата се покаже, а после я дава в назначения приемник, **DatagramPacket dp**. Пакетът се превръща в **String** заедно с информацията за Internet адреса и сокета откъдето пакетът идва. Изобразява се тази информация а после се добавят нови стрингове да покажат че тя е била изведена на сървъра.

Сега има малко затруднение. Както ще видите, има потенциално много Internet адреси и номера на портове откъдето би могло да дойде съобщение – тоест, клиентите могат да са резидентни на всяка машина. (В тази демонстрация те са всичките на **localhost**, но номерът на порт за всеки е различен.) За да се изпрати съобщение обратно на клиента, трябва да се знае неговия Internet адрес и номер на порт. За щастие тази информация е удобно пакетирана в **DatagramPacket** който е изпратил съобщението, така че просто трябва да я извладите чрез **getAddress()** и **getPort()**, които са използвани за построяването на **DatagramPacket echo** което е изпратено през същия сокет през който е получено. Освен това когато сокетът импраща датаграмата, той автоматично добавя Internet адреса и информацията за порта на тази машина, така че когато клиентът получи съобщението, той може да използва **getAddress()** и **getPort()** за да намери откъде идва датаграмата. Фактически единствения път когато **getAddress()** и **getPort()** не казват адреса и порта е когато вие създавате датаграмата и извиквате **getAddress()** и **getPort()** преди да изпратите датаграмата (в който случай се казва адреса и порта на тази машина, от която е била изпратена датаграмата). Това е основно за датаграмите: не е необходимо да знаете откъде е съобщението понеже това винаги е запомнено в датаграмата. Фактически най-добре е да

не се мъчите да следите откъде е съобщението, а да извличате информацията направо от датаграмите (както правим тук).

Ето програма за тестване на този сървър, която създава няколко клиента, всеки от които изпраща датаграма на сървъра и чака за echo.

```
//: c15:ChatterClient.java
// Tests the ChatterServer by starting multiple
// clients, each of which sends datagrams.
import java.net.*;
import java.io.*;

public class ChatterClient extends Thread {
    // Can listen & send on the same socket:
    private DatagramSocket s;
    private InetAddress hostAddress;
    private byte[] buf = new byte(1000);
    private DatagramPacket dp =
        new DatagramPacket(buf, buf.length);
    private int id;

    public ChatterClient(int identifier) {
        id = identifier;
        try {
            // Auto-assign port number:
            s = new DatagramSocket();
            hostAddress =
                InetAddress.getByName("localhost");
        } catch(UnknownHostException e) {
            System.err.println("Cannot find host");
            System.exit(1);
        } catch(SocketException e) {
            System.err.println("Can't open socket");
            e.printStackTrace();
            System.exit(1);
        }
        System.out.println("ChatterClient starting");
    }
    public void run() {
        try {
            for(int i = 0; i < 25; i++) {
                String outMessage = "Client #" +
                    id + ", message #" + i;
                // Make and send a datagram:
                s.send(Dgram.toDatagram(outMessage,
                    hostAddress,
                    ChatterServer.INPORT));
                // Block until it echoes back:
                s.receive(dp);
                // Print out the echoed contents:
                String rcvd = "Client #" + id +
                    ", rcvd from " +
                    dp.getAddress() + ", " +
                    dp.getPort() + ": " +
                    Dgram.toString(dp);
                System.out.println(rcvd);
            }
        }
    }
}
```

```

} catch(IOException e) {
    e.printStackTrace();
    System.exit(1);
}
}

public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
        new ChatterClient(i).start();
}

} //:~

```

ChatterClient е създаден като **Thread** така че могат да бъдат направени много клиенти да досаждат на сървъра. Тук може да се види че получаваният **DatagramPacket** изглежда точно като този за **ChatterServer**. В конструктора, **DatagramSocket** е създаден без аргументи понеже не му трябва да се саморекламира с номер на порт. Internet адресът използва за този сокет ще бъде “тази машина” (за примера - **localhost**) и автоматично ще се присвои номер на порт, както ще видите от изхода. Този **DatagramSocket**, подобно на ония за сървъра, ще се използва и за изпращане и за приемане.

hostAddressът е Internet адреса на хоста на който искате да говорите. Тази част от програмата където трябва да знаете точния Internet адрес и номер на порт е когато правите заминаващ **DatagramPacket**. Както винаги, хостът трябва да е с известен адрес и номер на порт така че клиентите да могат да започват разговори с хоста.

На всяка нишка е даден уникален идентификационен номер (макар че номерът на порта за нишката също е уникален). В **run()** се създава **String** на съобщението който съдържа идентификационния номер и номера на съобщението което в момента нишката изпраща. Този **String** се използва за създаване на датаграма която се изпраща на хоста на неговия адрес; номерът на порт се взема направо от константа в **ChatterServer**. Щом веднъж съобщението се изпрати, **receive()** блокира докато тойде ехото. Информацията изпратена със съобщението дава възможност да се проследи точно от кое съобщение е ехото. В този пример макар че UDP е “ненадежден” протокол ще видите че всички датаграми отиват където трябва. (Това ще е вярно за локалния хост и LAN ситуации, но може да започнете да виждате провали при нелокални връзки.)

Когато пуснете програмата ще видите че нишките се прекратяват, което значи че е дошло съответното ехо и е правилно прието; Ако това не стане, една или повече нишки ще увиснат, блокирани докато техният изход не се покаже.

Може да си помислите че единствения правилен начин да, например, се изпрати файл от една машина на друга е чрез TCP сокети, понеже те са “надеждни.” Поради бързината си обаче датаграмите могат да бъдат по-добро решение. Просто разделяте файла на пакети и номерирайте пакетите. Приемащата машина взема пакетите и възстановява файла; един “заглавен пакет” колко пакета да очаква и всякаква друга важна информация. Ако се загуби пакет, приемащата машина изпраща датаграма за да поиска повторното му изпращане.

RMI (Remote Method Invocation)

Традиционният подход за достъп до код на друга машина е труден, както и досаден и предразполагащ към грешки като се прилага. Най-весело е като си мислим за този проблем че обектът е на някаква друга машина, а може да му пратим съобщение и да получим отговора и да го използваме като чели обектът е на локалната машина. Това опростяване се дава точно от *Remote Method Invocation* (RMI) на Java 1.1. Тази секция минава по стъпките необходими за да създадете собствени RMI обекти.

Отдалечени интерфейси

RMI много използва интерфейси. Когато искате да създадете отдалечен обект вие замаскирате подлежащата машина със задаване на интерфейс. По този начин като получите манипулятор клиентът към отдалечения обект, той получава фактически интерфийс който се случва да бъде свързан към накаво локално парче код което говори през мрежата. Но вие не мислите за това, просто изпращате съобщения през манипулятора на интерфейса.

Когато създавате отдалечен интерфейс трябва да спазвате следните насоки:

1. Отдалеченият интерфейс трябва да бъде **public** (не може да има "пакетен достъп," тоест не може да бъде "приятелски"). Иначе клиентът ще получи съобщение за грешка когато се опита да получи достъп до отдалечения обект който реализира интерфейса.
2. Отдалеченият интерфейс трябва да разширява **java.rmi.Remote**.
3. Всеки метод в отдалечения интерфейс трябва да декларира **java.rmi.RemoteException** в неговата **throws** клуза освен зависимите от приложението изключения.
4. Отдалечен обект подаден като аргумент или връщана стойност (било директно или вграден в локален обект) трябва да бъде деклариран като отдалечен интерфейс, не като реализиращ клас.

Ето прост отдалечен интерфейс който дава точно време:

```
//: c15:ptime:PerfectTimeI.java
// The PerfectTime remote interface
package c15.ptime;
import java.rmi.*;

interface PerfectTimeI extends Remote {
    long getPerfectTime() throws RemoteException;
} ///:~
```

Той изглежда като всеки друг интерфейс само дето разширява **Remote** и всичките му методи изхвърлят **RemoteException**. Помните че **interface** и всичките му методи са автоматично **public**.

Реализация на отдалечен интерфейс

Сървърът трябва да има клас който разширява **UnicastRemoteObject** и реализира отдалечения интерфейс. Този клас също може да има и допълнителни методи, но само методите от отдалечения интерфейс ще са достъпни за клиента, разбира се, понеже клиентът ще получи манипулятор само към интерфейса, не и към класа който го реализира.

Трябва явно да напишете конструктор за отдалечения обект даже и ако само определяте конструктор по подразбиране който вика конструктора на базовия клас. Трябва да се напише понеже изхвърля **RemoteException**.

Ето реализацията на отдалечения интерфейс **PerfectTimeI**:

```
//: c15:ptime:PerfectTimeI.java
// The implementation of the PerfectTime
// remote object
package c15.ptime;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;
```

```

public class PerfectTime
    extends UnicastRemoteObject
    implements PerfectTimeI {
    // Implementation of the interface:
    public long getPerfectTime()
        throws RemoteException {
        return System.currentTimeMillis();
    }
    // Must implement constructor to throw
    // RemoteException:
    public PerfectTime() throws RemoteException {
        // super(); // Called automatically
    }
    // Registration for RMI serving:
    public static void main(String[] args) {
        System.setSecurityManager(
            new RMISecurityManager());
        try {
            PerfectTime pt = new PerfectTime();
            Naming.bind(
                "//colossus:2005/PerfectTime", pt);
            System.out.println("Ready to do time");
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
} ///:~

```

main() се оправя с всички детайли по конституирането на сървъра. Когато обхлужвате RMI обекти в някоя точка на програмата трябва:

1. Да създадете и инсталирате управител на сигурността който поддържа RMI. Единствения достъпен за RMI като част от дистрибуцията на Java е **RMISecurityManager**.
2. Да създадете един или повече екземпляри от отдалечения обект. Тук може да се види създаването на **PerfectTime** обект.
3. Да регистрирате най-малко един отдалечен обект в RMI регистъра за отдалечени обекти с цел бутстрапиране. Един отдалечен обект може да дава манипулатори към други отдалечени обекти. Това позволява организация при която клиентът чете регистъра само веднъж, да вземе първия отдалечен обект.

Установяване на регистъра

Може да видите извикване на **static** метода **Naming.bind()**. Обаче такова извикване изисква регистърът да работи като отделен процес на машината. Името на сървъра на регистъра е **rmiregistry**, а под 32-bit Windows пишете:

```
| start rmiregistry
```

за да го стартирате във фонов режим. С Unix това е:

```
| rmiregistry &
```

Както много мрежови програми **rmiregistry** е разположено на IP адреса на машината на която е стартирано, но също трябва да слуша и порт. Ако сте извикали **rmiregistry** като горе, без аргумент, портът на реджистрито ще стане по подразбиране 1099. Ако го искате на друг порт, слагате аргумент на командния ред за определяне на порта. За този пример портът ще бъде 2005, така че **rmiregistry** ще се стартира така под 32-bit Windows:

```
| start rmiregistry 2005
```

или за Unix:

```
| rmiregistry 2005 &
```

Информацията за порта трябва също да бъде дадена на командата **bind()**, както и IP адреса на машината където е разположено реджистрито. Но това показва възможен проблем ако решите да тествате RMI програми локално по начина показан преди в тази глава. В рилиза JDK 1.1.1 има два проблема:³

1. **localhost** не работи с RMI. Така за да се експериментира с RMI на единствена машина трябва да се даде името на машината. За да намерите името на вашата машина под 32-bit Windows изберете контрол панела и "Network." Селектирайте "Identification" и ще видите името. В моя случай аз нарекох моя компютър "Colossus" (заради многото твърди дискове които съм му закачил за да събера всичките развойни системи). Изглежда че капитализацията се игнорира.
2. RMI няма да работи докато вашият компютър няма активна TCP/IP връзка, даже и всичките компоненти да говорят помежду си само на локалната машина. Това значи че трябва да се свържете към вашия Internet доставчик преди да работите с отдалечени интерфейси или ще виждате някакви скритни съобщения за изключения.

Като имаме пред вид всичко това команда **bind()** става:

```
| Naming.bind("//colossus:2005/PerfectTime", pt);
```

Ако използвате порта по подразбиране 1099, не е необходимо да пишете порт, така че ще е:

```
| Naming.bind("//colossus/PerfectTime", pt);
```

В бъдещи версии на JDK (след 1.1) когато бъгът с **localhost** се оправи ще може да правите тестове оставайки IP адреса и използвайки само идентификатор:

```
| Naming.bind("PerfectTime", pt);
```

Името на услугата е произволно; тук то е **PerfectTime**, точно като името на класа, но може да бъде каквото си искате. Важно е то да е единствено в отдалеченото реджистри за да може да се намери обектът. Ако името вече е в реджистрито, ще получите **AlreadyBoundException**. За да се предотврати това винаги може да използвате **rebind()** вместо **bind()**, понеже **rebind()** или добавя име или замества името (и свързаните с него неща) ако вече го има.

Макар и **main()** да завършва, вашият обект е създаден и регистриран, така че се поддържа жив от регистъра, чакатки да дойде клиент и да го поиска. Доколкото **rmiregistry** работи и не викате **Naming.unbind()** с вашето име, обектът ще бъде налице. Поради тази причина когато разработвате кода си трябва да спрете **rmiregistry** и да го пуснете пак когато сложите нова версия на обекта си.

Не се налага да оформите **rmiregistry** като отделен процес. Ако се знае че вашето приложение само ще използва регистъра, може да го пуснете от вашата програма с реда:

```
| LocateRegistry.createRegistry(2005);
```

Както преди, 2005 е номерът на порта който сме използвали за този пример. Това е еквивалентно на пускане **rmiregistry 2005** от командния ред, но често може да е по-удобно когато разработвате RMI код понеже премахва допълнителните стъпки за спиране и пускане на регистъра. Щом сте изпълнили този код, може да **bind()** чрез **Naming** както преди.

³ Many brain cells died in agony to discover this information.

Създаване на стубове и скелетони

Ако компилирате и пуснете **PerfectTime.java**, няма да работи даже и да имате правилно работещо **rmiregistry**. Така е защото работната рамка на RMI не е всичкото, което е необходимо. Трябва първо да създадете стубове и скелетони които да дадат мрежовите връзки и да позволят да се преструвате че отдалеченият обект е просто още един обект на вашата локална машина.

Това което става зад сцената е много сложно. Всички обекти които давате на или връщате от отдалечения обект трябва да прилагат **implement Serializable** (ако искате да давате отдалечени позовавания наместо цели обекти, аргументите на обекта могат да прилагат **implement Remote**), така че може да си въобразим че стубовете и скелетоните автоматично правят сериализация както си "водят под строй" всички аргументи през мрежата и връщат резултата. За щастие не е необходимо да знаете нищо от това, но трябва да създадете стубовете и скелетоните. Това става просто: викате инструмента **rmic** върху вашия компилиран код, създават се необходимите файлове. Така че единственото изискване е още една стъпка в процеса на компилирането.

Инструментът **rmic** обаче е приидирчив относно пакетирането и пътищата. **PerfectTime.java** е в пакета **c15.PTime**, та даже и да извикате **rmic** в същата директория където е разположен **PerfectTime.class**, **rmic** няма да го намери, понеже търси по "пътя до класовете". Така че трябва да се посочи местоположението ако не на пътя, примерно така:

```
| rmic c15.PTime.PerfectTime
```

Не е необходимо да бъдете в директорията съдържаща **PerfectTime.class** когато изпълнявате тази команда, но резултатът ще бъде сложен в текущата директория.

Когато **rmic** сработи успешно, ще имате два нови класа в директорията:

```
| PerfectTime_Stub.class  
| PerfectTime_Skel.class
```

Съответно за стуба и скелетона. Сега сте готови за разговорите на клиента и сървъра помежду им.

Използване на отдалечения обект

Цялата работа с RMI е да се направи просто използването на отдалечените обекти. Единственото допълнително нещо което трябва да направите е да накарате вашата програма да се огледа, открие и достави интерфейса от сървъра. От там нататък, всичко е просто обикновено програмиране на Java: изпращане на съобщения до обекти. Ето програмата която използва **PerfectTime**:

```
//: c15:ptime:DisplayPerfectTime.java  
// Uses remote object PerfectTime  
package c15.ptime;  
import java.rmi.*;  
import java.rmi.registry.*;  
  
public class DisplayPerfectTime {  
    public static void main(String[] args) {  
        System.setSecurityManager(  
            new RMISecurityManager());  
        try {  
            PerfectTime t =  
                (PerfectTime)Naming.lookup(  
                    "//colossus:2005/PerfectTime");
```

```

for(int i = 0; i < 10; i++)
    System.out.println("Perfect time = " +
        t.getPerfectTime());
} catch(Exception e) {
    e.printStackTrace();
}
}

} //:~
```

Стрингът ID е същият, който е използван за регистрирането на обекта с **Naming**, и първата част представя URL и номер на порт. Понеже използвате URL, може също да посочите машина в Internet.

Това което се връща от **Naming.lookup()** трябва да бъде кастнато до отдалечения интерфейс, не до класа. Ако кастнете към класа, ще получите изключение.

Може да видите в извикването

```
t.getPerfectTime()
```

че щом веднъж имате манипулятор към отдалечения обект, програмирането с него е неразличимо от програмирането с локалните обекти (с една разлика: отдалечените методи изхвърлят **RemoteException**).

Въведение в CORBA

В големи, разпределени приложения, вашите нужди може да не могат да се задоволят от изложените подходи. Например може да искате да комуникирате с традиционни бази данни, или пък да искате обслужване от сървърските обекти независимо от неговото физическо разположение. Такива операции изистват някаква форма на Remote Procedure Call (RPC), може би и независимост от езика. Това е мястото, където може да помогне CORBA.

CORBA е черта на езика; това е интеграционна технология. Това е спецификация която доставчиците могат да следват за да създадат CORBA-съвместими интеграционни продукти. CORBA е част от усилието на OMG за дефиниране на стандартна, независима от езика възможност за съвместна работа на обекти.

CORBA дава възможност да се правят отдалечени извиквания на процедури в Java обекти и не-Java обекти, а също и да се взаимодейства с традиционни системи по прозрачен относно местоположението начин. Java добавя поддръжка на мрежи и чудесен обектно-ориентиран език за създаване на графични и неграфични приложения. Обектовите модели на Java и OMG се проектират идеално един върху друг; например, и Java и CORBA прилагат концепцията за интерфейс и моделът за позоваване на обект.

Основи на CORBA

Спецификацията на OMG за взаимодействие на обекти често се споменава като Object Management Architecture (OMA). OMA дефинира два компонента: Core Object Model и Reference Architecture на OMA. Core Object Model задава основните концепции за обекти, интерфейс, операция и т.н. (CORBA е подобрение на Core Object Model.) OMA Reference Architecture определя подлежащата архитектура от услуги и механизми които позволяват на обектите да взаимодействват. OMA Reference Architecture включва Object Request Broker (ORB), Object Services (също известни като CORBA услуги) и общи помагала.

ORB е комуникационната шина чрез която обекти могат да изискват услуги от други обекти, без значение физическото им разположение. Това значи че нещото което изглежда като извикване на клиентски метод в кода е сложна операция. Първо, трябва да съществува

връзка със сървърския обект, а за да се създаде връзка ORB-то трябва да знае къде е резидентен реализация сървърски код. Щом веднъж се направи връзката, параметрите на метода трябва да се маршалнат, тоест да се превърнат в двоичен поток за изпращане през мрежата. Друга информация която трябва да се изпрати е името на сървърската машина, сървърския процес, идентичността на сървърския обект в него процес. Накрая всичко това се изпраща по жиците с протокол от ниско ниво, информацията се декодира на сървърската страна и се изпълнява извикването. ORB скрива цялата тази сложност от програмиста и прави операцията почти толкова проста като викането на локален обект.

Не е специфицирано как да се реализира ORB Core, но за осигуряване на основна съвместимост между ORB на различните доставчици, OMG определя мрежа от услуги които са достъпни чрез стандартни интерфейси.

Interface Definition Language (IDL) на CORBA

CORBA е проектиран за езикова прозрачност: клиентски обект може да вика методи на сървърски обект от друг клас, без значение на езика с който са били реализирани. Разбира се, клиентският обект трябва да знае имената и сигнатурите на методите на сървърския обект. Тук е мястото на IDL. IDL на CORBA е неутрален към езиците начин да се посочат даннови типове, атрибути, операции, интерфейси и други неща. Синтаксисът на IDL е подобен на този на C++ или Java. Следната таблица показва съответствието между някои от концепциите общи за трите езика които може да се посочат с IDL на CORBA:

CORBA IDL	Java	C++
Модул	Пакет	Пространство на имената
Интерфейс	Интерфейс	(Чисто) абстрактен клас
Метод	Метод	Член-функция

Поддържа се също и концепцията за наследяването, използва се операторът двоеточие както в C++. Програмистът пише описание на атрибутите, методите и интерфейсите на IDL, което ще бъде използвано от сървъра и клиентите. После IDL се компилира с доставян от производителя IDL/Java компилатор, който чете IDL сурса и генерира Java код.

IDL компилаторът е изключително полезен инструмент: той не просто генерира Java сурс еквивалентен на този на IDL, той също генерира код който ще маршалне аргументите и ще направи отдалечените извиквания. Този код наречен стъбов и скелетонен код е организиран в множество Java сорсови файлове и е обикновено част от същия Java пакет.

Услуга за имената

Службата за имената е основна в CORBA. Обект в CORBA се достига чрез позоваване, парченце информация която няма смисъл за човек-четец. Но на позоваванията може да се присвояват стрингови, програмно определени имена. Тази операция е известна като *stringifying the reference* (стрингифиране на позоваването-б.пр.) и един от ОМА компонентите, Naming Service, е посветен на превръщането и проектирането стринг-обект и обект-стринг. Понеже Naming Service работи като телефонен указател който и сървърът и клиентите могат да разглеждат и манипулират, това е в отделен процес. Създаването на обект-към-стринг изображение се назова *binding an object*, а махането на изображението е наречено *unbinding*. Вземането на обектово позоваване (обобщен указател-б.пр.) чрез стринг се нарича *resolving* на името.

Например в началото едно сървърско приложение може да създаде сървърски обект, да свърже обекта в услугата за имената, а после да чака клиентите да дават поръчки. Клиентът първо придобива референс към сървърския обект, резолвира стринговото име, а после може да прави поръчки на сървъра чрез референса.

Отново, Naming Service спецификацията е част от CORBA, но приложението което я реализира се доставя от доставчика на ORB. Начинът да се получи достъп до Naming Service функционалността може да се променя от доставчик към доставчик.

Пример

Кодът по-долу няма да е довършен понеже различните ORB-ти имат различни начини за достъп до услугите на CORBA така че примерите са зависими от доставчика. (Примерът по-долу използва JavaIDL, безплатен продукт на Sun който идва с "лекото" ORB, службата за имената и IDL-Java компилатор.) Освен това, понеже Java е млад и още се развива, не всички черти на CORBA са представени в различните Java/CORBA продукти.

Искаме да реализираме сървър, работещ на някаква машина, който може да бъде питан за точното време. Искаме също да реализираме клиент който пита за точното време. Ще направим и двете програми на Java, но бихме могли също да използваме два различни езика (което често се случва в реални ситуации).

Написване на IDL сурса

Първата стъпка е да се напише IDL описание на даваните услуги. Това обикновено се прави от сървърския програмист, който после е свободен да реализира сървърските програми на всякакъв език, за който има CORBA IDL компилатор. IDL файла се дава на програмистите на клиентската страна и става мост между езиците.

Примерът по-долу показва описанието на IDL за нашия сървър за точно време:

```
//: c15:corba:ExactTime.idl
module RemoteTime {
    interface ExactTime {
        string getTime();
    };
}; //:~
```

Това е декларация на **ExactTime** интерфейса вътре в **RemoteTime** пространството на имена. Интерфейсът се състои от един метод кото връща точно време в **string** формат.

Създаване на стъбове и скелетони

Втората стъпка е да се компилира IDL-то за създаване на Java стъбов и скелетоне код който ще използваме при реализацијата на клиента и сървъра. Инструментът който идва с JavaIDL продукта е **idltojava**:

```
| idltojava -fserver -fclient RemoteTime.idl
```

Двета флагаказват на **idltojava** да генерира код и за стуб и за скелетон. **idltojava** генерира Java наречен на името на IDL модула, **RemoteTime**, а генерираните Java файлове се слагат в **RemoteTime** поддиректорията. **_ExactTimeImplBase.java** е скелетонът който ще използваме за реализацијата на сървърския обект, а **_ExactTimeStub.java** ще е за клиента. Има Java представяния на IDL интерфейса в **ExactTime.java** и в двойка други поддържащи файлове, например за обслужване на операциите на службата за имената.

Реализация на сървъра и клиента

По-долу можете да видите кода за сървърската страна. Реализацията на сървърския обект е класът **ExactTimeServer**. **ExactTimeServer** е приложението което създава сървърския обект, регистрира го в ORB-то, дава име на обектото позоваване, а после седи тихо да чака клиентски поръчки.

```
| //: c15:corba:RemoteTimeServer.java
```

```

import RemoteTime.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.util.*;
import java.text.*;

// Server object implementation
class ExactTimeServer extends _ExactTimeImplBase{
    public String getTime(){
        return DateFormat.
            getTimelInstance(DateFormat.FULL).
            format(new Date(
                System.currentTimeMillis()));
    }
}

// Remote application implementation
public class RemoteTimeServer {
    public static void main(String args) {
        try {
            // ORB creation and initialization:
            ORB orb = ORB.init(args, null);
            // Create the server object and register it:
            ExactTimeServer timeServerObjRef =
                new ExactTimeServer();
            orb.connect(timeServerObjRef);
            // Get the root naming context:
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references(
                    "NameService");
            NamingContext ncRef =
                NamingContextHelper.narrow(objRef);
            // Assign a string name to the
            // object reference (binding):
            NameComponent nc =
                new NameComponent("ExactTime", "");
            NameComponent path() = {nc};
            ncRef.rebind(path, timeServerObjRef);
            // Wait for client requests:
            java.lang.Object sync =
                new java.lang.Object();
            synchronized(sync){
                sync.wait();
            }
        } catch (Exception e) {
            System.out.println(
                "Remote Time server error: " + e);
            e.printStackTrace(System.out);
        }
    }
} ///:~

```

Както може да видите, да се реализира сървърския обект е просто; той е нормален Java клас който наследява от скелетонния код генериран от IDL компилатора. Нещата стават мъничко по-сложни когато се дойде до взаимодействието с ORB и други услуги на CORBA.

Някои услуги на CORBA

Това е кратко описание какво върши JavaIDL-замесения код (най-вече игнорирайки CORBA кода който е зависим от производителя). Първият ред в `main()` стартира ORB, разбира се защото нашият сървърски обект ще иска да взаимодейства с него. Веднага след инициализацията на ORB се създава сървърския обект. Фактически точният термин би бил *transient servant object*: обект който приема заявки от клиентите и чието време на живот е колкото на породилия го процес. Щом транзиентния обслужващ обект е създаден, той се регистрира в ORB-то, което значи че ORB знае за съществуването му и може да препраща заявки към него.

До този момент е налице само `timeServerObjRef`, позоваване на обект което е известно само в текущия сървърски процес. Следващата стъпка ще бъде да се присвои стрингово име на този референс; клиентите ще го използват за намиране на обслужващия обект. Тази операция ще извършим чрез Naming Service. Първо, трябва ни позоваване на обект за Naming Service; викането на `resolve_initial_references()` взема стрингираното позоваване към Naming Service което е "NameService," в JavaIDL, и връща позоваване на обект. Това е каст към специфичен `NamingContext` референс чрез метода `narrow()`. Сега може да използваме услугите за имената.

За да свържем обслужващия обект със стрингованото позоваване на обект първо създаваме един `NameComponent` обект, инициализиран с "ExactTime," стринговото име което искаме да свържем с обекта. После използвайки метода `rebind()` стрингования референс се свързва с обектовия референс. Използваме `rebind()` за присвояване на референса, даже и ако той вече съществува, доколкото `bind()` дава изключение ако той вече съществува. Име се съставя в ORBA от последователност от NameContexts – затова използваме масив за свързване на името с обектовия референс.

Обслужващият обект е накрая готов за използване от клиентите. В тази точка сървърският процес влиза в състояние на изчакване. Отново, това е защото е транзиентен обслужващ, така че времето му на живот е колкото на сървърския процес. JavaIDL в момента не поддържа устойчиви обекти – обекти които надживяват прекратяването на процеса който ги е продил.

Сега като имаме представа какво прави сървърският код, нека да погледнем клиентския код:

```
//: c15:corba:RemoteTimeClient.java
import RemoteTime.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class RemoteTimeClient {
    public static void main(String args) {
        try {
            // ORB creation and initialization:
            ORB orb = ORB.init(args, null);
            // Get the root naming context:
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references(
                    "NameService");
            NamingContext ncRef =
                NamingContextHelper.narrow(objRef);
            // Get (resolve) the stringified object
            // reference for the time server:
            NameComponent nc =
                new NameComponent("ExactTime", "");
            NameComponent path[] = {nc};
            ExactTime timeObjRef =
                ExactTimeHelper.narrow(
```

```

    ncRef.resolve(path));
    // Make requests to the server object:
    String exactTime = timeObjRef.getTime();
    System.out.println(exactTime);
} catch (Exception e) {
    System.out.println(
        "Remote Time server error: " + e);
    e.printStackTrace(System.out);
}
}
} //:~

```

Първите няколко реда правят същото като в сървърския код: ORB се инициализира и се резолвира референса към службата за имената. После ни трябва референс към обслужващия обект, така че даваме стрингования референс към обект на метода **resolve()** и кастваме резултата към референса на интерфейса **ExactTime** чрез метода **narrow()**. Накрая викаме **getTime()**.

Активиране на процеса на службата за имената

Накрая имаме клиентското и сървърското приложение готови за съвместна работа. Видяхте че и двете имат нужда от службата за имената за резолвиране на стрингованите референси. Трябва да стартирате процеса на службата за имената преди да използвате клиента или сървъра. В JavaIDL службата за имената е Java приложение което идва в пакет с продукта, но нещата може да са различни в други продукти. JavaIDL службата за имената се изпълнява в екземпляр на JVM и по подразбиране слуша на порт 900.

Активиране на сървъра и клиента

Сега може да се стартират сървърското и клиентското приложения в този ред, понеже сървърът е временен). Ако всичко е направено коректно, ще получите единствен ред на конзолния прозорец на клиента, показващ текущото време. Разбира се, това може да не е много вълнуващо само по себе си, но ще имате пред вид едно нещо: ако и да са на една физическа машина, клиентското и сървърското приложение работят на различни виртуални машини и комуникират чрез позлежащи структури, ORB-то и Naming Service.

Това е прост пример, проектиран да работи без мрежа, но ORB е обикновено конфигурирано за прозрачност спряма положението. Когато сървърът и клиентът са на различни машини, ORB може да резолвира стринговани обектови позовавания чрез компонент наречен *Implementation Repository*. Макар че Implementation Repository е част от CORBA, почти няма спецификация, така че варира от производител на производител.

Както може да се види, има много повече да се каже за CORBA отколкото разгледаното тук, но трябва да имате основната идея. Ако искате повече информация за CORBA, мястото да започнете е сайта на OMG, на <http://www.omg.org/>. Там ще намерите документация, "бели книги", протоколи и препратки към други CORBA източници и продукти.

Java аплети и CORBA

Java могат да действат като CORBA клиенти. По този начин аплетът получава достъп до ресурси които са представени като отдалечени CORBA обекти. Но аплет може да се свърже само със сървъра от който е сварен, така че всичките CORBA обекти с които работи аплетът трябва да са на него сървър. Това е точно наопаки на целта на CORBA: да дава пълна прозрачност спрямо положението.

Това е въпрос от мрежовата сигурност. Ако не сте в Intranet, едно решение е да отслабите условията за сигурност на браузера. Или, да установите политика на "защитна стена" при връзката с други сървъри.

Някои Java ORB продукти дават собствени решения на този проблем. Например някои прилагат тъй наречения HTTP Tunneling, докато други имат собствени защитни функции.

Това е твърде сложна тема, за да се разгледа в апендиц, но определено трябва да знаете че съществува.

CORBA vs. RMI

Видяхте че една от основните енергии на CORBA е поддръжката на RPC, което позволява вашите локални обекти да викат методи на отдалечени обекти. Вече има собствена за Java черта която прави същото: RMI (виж глава 15). Докато RMI прави RPC възможно между Java обекти, CORBA прави RPC възможно между обекти реализирани на различни езици. Това е голяма разлика.

Обаче RMI може да бъде използвано за викане на услуги на отдалечен, не-Java код. Необходима е само някаква обвивка от Java обект около не-Java кода на сървърската страна. Обгръщащият обект се свързва външно с Java клиентите чрез RMI, а вътрешно се свързва с не-Java кода използвайки техниките по-горе, като JNI или J/Direct.

Този подход изиска да направите някакъв вид интеграционен (свързващ) слой, което точно CORBA прави за вас, но тогава не ви трябва ORB от трети доставчици.

Jini: разпределени услуги

Тази секция⁴ прави преглед на Jini технологията на Sun Microsystems. Тя описва някои неща от Jini и показва как архитектурата на Jini помага да се повиши степента на абстракция в програмирането на разпределени системи, ефективно превръщайки мрежовото програмиране в ОО програмиране.

Jini в контекст

Традиционно операционните системи са създавани като се предполага че компютърът има процесор, памет и диск. Като щракнете копчето на компютъра, първото нещо е да огледа за диск. Ако не намери диск, не може да функционира като компютър. Все по-често обаче компютрите се появяват в друг вид: като вградени устройства които имат процесор, памет и мрежова връзка – но не диск. Първото нещо което целуларния телефон прави като го включите, например, е да погледне за телефонна мрежа. Ако не намери такава, не може да работи като телефон. Тази тенденция в хардуерното оборудване, от центриране около дисковете към центриране около мрежите, ще има влияние върху организацията на нашия софтуер – и това е мястото на Jini.

Jini е опит да се преосмисли компютърната архитектура, давайки по-голяма тежест на мрежата и все по-многото процесори в устройства без диск. Тези устройства, които ще са от много различни производители, ще трябва да работят заедно в мрежата. Самата мрежа ще бъде много динамична – устройства и услуги ще бъдат добавяни и махани често. Jini дава механизми за гладко добавяне, махане и намеране на устройства и услуги в мрежата. Освен това Jini дава програмен модел който улеснява програмистите да накарат устройства да си говорят.

Построено върху Java, обектова сериализация и RMI (което позволява на обектите да се движат в мрежата от виртуална машина към виртуална машина) Jini се опитва да разшири ползите от ООП върху мрежата. Вместо да изиска от доставчиците техните устройства да отговарят на мрежов протокол, Jini дава възможност на устройствата да си говорят чрез обектов протокол.

⁴ This section was contributed by Bill Venners (www.artima.com)

Какво е Jini?

Jini е множество от API-та и мрежови протоколи което може да ви помогне да построите и пуснете разпределени системи които са организирани като *федерации от услуги*. Услуга е всичко което седи на мрежата и може да изпълни полезна функция. Хардуерни устройства, софтуер, комуникационни канали, даже хората-оператори – могат да бъдат услуги. Едно Jini-съгласно дисково устройство, например, би могло да предложи услуга "масова памет". Jini-принтер би могъл да предложи "печаташа" услуга. Федерацията от услуги, тогава, е множество услуги, тъкъто достъпни в мрежата, които клиент (програма, устройство или потребител) може да използва за постигане на някаква цел.

За да изпълни задача, клиентът използва определен брой услуги. Например, един клиент може да натовари образи от дигитална камера в услугата за запомняне на образи, да ги запише в услугата устойчива памет предлагана от дисково устройство и да изпрати за печатане у малени версии на образите в услугата за печатане, предлагана от принтер. В този пример клиентската програма строи разпределена система състояща се от нея самата, услугата за запомняне на образи, услугата с енергонезависима памет и услугата за цветно печатане. Клиентът и услугите от тази разпределена система работят заедно за да изпълнят задачата за изпращането на образите от дигиталния фотоапарат за запомняне и отпечатването на хартия.

Идеята зад думата *федерация* е че гледната точка на Jini за мрежата не включва централна власт, поради което съвкупността от услугите има формата на *Федерация* – група от равноправни членове. Вместо централно ръководство инфраструктурата на Jini по време на изпълнение просто позволява на клиентите и услугите да се намират едни други (чрез услуга за оглед на услугите, която поддържа списък на достъпните в момента услуги). След като услугите се намерят една с друга, те продължават сами. Клиентът и неговите записани услуги работят самостоятелно, без намеса на инфраструктурата на Jini. Ако справочникът за услуги на Jini се скапе, всяка разпределена система заработила чрез въпросната услуга преди скапването може да продължи своята работа. Jini даже включва мрежов протокол който клиентите могат да използват за намиране на услуги при липса на справочника.

Как работи Jini

Jini определя *runtime infrastructure* която е резидентна на мрежата и дава механизми които позволяват да се добавят, мачат, намират и използват услуги. Рънтайн инфраструктурата има на три места: в справочниците които са резидентни на мрежата, в доставчиците на услуги (такива като Jini-работещи устройства) и в клиентите. *Lookup services* са централния организиращ механизъм в Jini-базирани системи. Когато нови услуги станат достъпни в мрежата, те се регистрират в справочника. Когато клиенти искат да намерят услуга да им свърши работа в някаква задача, те се консултират със справочната услуга.

Рънтайн инфраструктурата използва един протокол от мрежово ниво, наречен *discovery* и два от обектово ниво, наречени *join* и *lookup*. Discovery позволява на клиентите и услугите да открият справочната услуга. Join дава възможност на услуга да се регистрира в справочника. Lookup позволява на клиента да пита за услуги според нуждите си.

Процесът discovery

Discovery работи така: Да допуснем, че имате Jini-способно дисково устройство, което предлага услугата "устойчива памет". Щом свържете устройството в мрежата, то издава съобщение за присъствие като пуска мултикастен пакет в мрежата на добре известен порт. Включените в това съобщение IP адрес и номер на порт са където може да се намери устройството от службата-регистър.

Оглеждащите услуги следят споменатия известен порт за появата на съобщения за наличност на нови услуги. Когато се получи такъв пакет, услугата го отваря и преглежда. Пакетът

Съдържа информация за услугата дали тя трябва или не трябва да влезе в контакт с изпращащата. Ако трябва, контактът става направо чрез създаването на TCP връзка на IP адреса и номера на порт извлечени от пакета. Използвайки RMI услугата изпраща обект, наречен **регистратор на услуга** по мрежата до източника на пакета. Целта на регистратора на услуга е да подпомогне по-нататъшната комуникация между оглеждащата услуга и източника на пакета. Чрез викане на методи на този обект изпращащът на анонса може да осъществи присъединяването и оглеждането с оглеждащата услуга. В случая на дисково устройство услугата би направила TCP връзка с дисковото устройство и би отпрамила към него обект-регистратор на услуга, чрез който дисковото устройство би регистрирало неговата услуга "устойчива памет" в процеса на присъединяване.

Процесът на присъединяване

Щом доставчикът на услугата има обекта-регистратор, крайния продукт от `discovery`, той е готов за присъединяване – да стане част от федерацията на услуги които са регистрирани в оглеждащата услуга (регистратора на услуги). За да се присъедини доставчикът вика метода `register()` на регистриращия обект, давайки като параметър обект наречен артикул на услугата, група обекти, които описват услугата. Методът `register()` изпраща копие от артикула на услугата до регистъра, където той се запомня. Щом това стане, процесът на присъединяване е завършил: новата услуга е регистрирана в регистриращата (оглеждащата) услуга.

Артикулът на услугата е контейнер за няколко обекта, включително такъв наречен **обект на услугата**, който клиентите може да използват за работа с услугата. Артикулът може също да включва всяка към брой **атрибути**, които могат да бъдат всеки обект. Някои потенциални атрибути са икони, класове които дават GUI за услугата и обекти които дават повече информация за услугата.

Обектите на услугата обикновено прилагат един или повече интерфейси чрез които клиентите контактуват с услугата. Например регистърът е Jini услуга и нейния обект е регистратора на услуга. Методът `register()` извикан от доставчиците на услуга по време на присъединяването е деклариран в `ServiceRegistrar` интерфейса (член на пакета `net.jini.core.lookup`), който прилагат всичките регистратори на услуги. Клиентите и доставчиците на услуги говорят с регистратора чрез обекта-регистратор на услуги чрез викане на методи декларирани в интерфейса `ServiceRegistrar`. По подобен начин дисковото устройство би дало обект който прилага някакъв добре познат интерфейс на услуга с памет. Клиентите ще оглеждат и работят с устройството чрез този интерфейс.

Процесът на преглеждане

Щом веднъж услугата се е регистрирала в регистъра чрез процес на присъединяване, тя е достъпна за ползване от клиентите. За да се построи разпределена система от услуги които ще работят заедно за изпълнение на определена задача клиентът трябва да намери и разгледа помощта на отделните услуги. За да намери услуга, клиентът пита регистъра в процес, наречен **преглеждане**.

За да огледа, клиентът вика метода `lookup()` на обекта-регистратор на услуга. (Клиентът, както доставчикът на услугата, получава регистратора на услугата чрез описания по-рано процес `discovery`.) Клиентът дава като аргумент на `lookup()` *service template*, обект, който служи като критерий за търсене при заявка. Шаблонът на услугата може да включва масив от **Class** обекти. Тези **Class** обекти показват на регистриращата услуга Java типа (или типовете) на обекта-услуга, желан от клиента. Шаблонът на услугата също може да включва *ID* на услугата, който уникатно бележи услугата, и атрибути, които трябва да са точно като атрибутите натоварени от доставчика в артикула на услугата. Шаблонът на услугата може също да съдържа метасимболи за всякакви полета. Такъв метасимбол ("уайлдкард") в полето *ID* например, ще се удовлетвори от всякакъв *ID*. Методът `lookup()` изпраща шаблона на

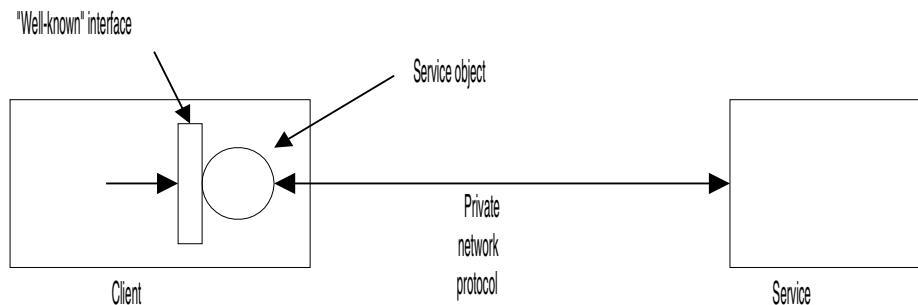
услугата на регистратора, който изпълнява заявката и изпраща нула обратно на всеки отговарящ на условията обект-услуга. Клиентът взема позоваване на отговарящите на условията обекти като връщан резултат от метода `lookup()`.

В общия случай клиентът оглежда услуга по Java тип, обикновено интерфейс. Например ако клиентът трябва да използва принтер, той би композирал шаблон на услугата която включва **Class** обект за добре познат интерфейс към принтерски услуги. Всички принтерски услуги биха прилагали този добре познат интерфейс. Регистриращата (оглеждащата) услуга би върнала обект-услуга (или обекти) които прилагат този интерфейс. Атрибути може да се включват в шаблона на услугата за да се стесни множеството на отговарящите на условията обекти-базирано търсене. Клиентът би използвал принтерската услуга чрез викане на методи от добре познатия интерфейс на принтерски услуги на обекта-услуга.

Разделяне на интерфейса и реализацията

Архитектурата на Jini въвежда ООП в мрежите като дава възможност на услугите да използват едно от големите предимства на обектите: разделянето на интерфейса и приложението. Например един обслужващ обект може да даде услуга на клиентите по много начини. Обектът може фактически да представлява цялата услуга, която се разтоварва при потребителя по време на преглеждането и после се изпълнява локално. Алтернативно, обектът-услуга може да бъде само мост между клиента и сървъра. Когато клиентът му трябват методи от обекта-услуга, той изпраща поръчка по мрежата до сървъра, който върши същинската работа. Третата възможност е част от работата да се върши локално от обекта-услуга, а другата част - от сървъра.

Важна част от архитектурата на Jini е че мрежовия протокол използван за комуникация между мостов обект-услуга и отдалечен сървър не е необходимо да бъде известен на клиента. Както е илюстрирано по-долу, мрежовият протокол е част от реализацията на услугата. Този протокол е частна работа на проектанта на услугата. Клиентът може да комуникира с услугата по този протокол понеже тя инжектира част от собствения си код (обекта-услуга) в адресното пространство на клиента. Инжектирания обект би могъл да комуникира с уклюгата чрез RMI, CORBA, DCOM, някакъв особен протокол построен на основата на сокети и потоци, а и с каквото и да е с друго. Клиентът просто няма нужда да се грижи за мрежовите протоколи, понеже говори на добре познат интерфейс който прилага обекта-услуга. Последният се грижи за всяка необходима комуникация по мрежата.



The client talks to the service through a well-known interface

Различните реализации на една услуга могат да използва напълно различни подходи и мрежови протоколи. Услугата може да използва специален хардуер за да изпълни поръчките, може и да прави всичко софтуерно. Фактически подходът към реализацията на една услуга може да еволюира с времето. Клиентът може да бъде сигурен че разполага с обект-услуга който разбира текущата реализация на услугата, понеже клиентът приема обекта-услуга (чрез оглеждащата услуга) от самия доставчик на услугата. За клиента услугата изглежда като добре познат интерфейс, без значение каква е реализацията.

Абстракция на разпределени системи

Jini се опитва да повиши нивото на абстракция в програмирането на разпределени системи от това на мрежив протокол до това на обектов интерфейс. При все повечето вградени устройства свързани към мрежи много части на разпределената система могат да са от различни производители. Jini прави нещущко споразумяването между производителите относно мрежовите протоколи, по които ще комуникират устройствата им. Вместо това трябва да има съгласие относно Java интерфейсите по които ще комуникират устройствата. Процесът на откриване, присъединяване и преглеждане, извършване от инфраструктурата на Jini позволяват на устройствата ада се намират едно-друго в мрежата. След като се се намерили, устройствата ще могат да комуникират помежду с чрез Java интерфейси.

JavaSpaces

Резюме

Фактически има много повече за мрежите от това, което може да бъде разгледано в това въведение. Java нетъркингът също дава много широка поддръжка за работа с URLobe, включително протоколни обработчици за различни видове съдържание които могат да бъдат отворени в Internet сайт. Може да намерите други Java мрежови черти грижливо и пълно описани в *Java Network Programming* от Elliotte Rusty Harold (O'Reilly, 1997).

Упражнения

1. Компилирайте и пуснете програмите **JabberServer** и **JabberClient** от тази глава. Сега редактирайте файловете за да махнете всякакво буфериране, компилирайте, пуснете и наблюдавайте резултатите.
2. Създайте сървър който питва за парола, после отваря файл и го изпраща по мрежова връзка. Създайте клиент който се свързва с този сървър, дава подходящата парола, после приема и запазва файла. Тествайте двойката програми на вашата машина използвайки **localhost** (IP адреса за локална обратна връзка **127.0.0.1** даван от извикването на **InetAddress.getByName(null)**).
3. Променете сървъра от упр. 2 така че да използва многонишковост за поддръжка на много клиенти.
4. Променете **JabberClient** така че да няма изпразване на буфера и наблюдавайте ефекта.
5. Постройте **ShowHTML.java** да дава аплет, който е защитена с парола врата към конкретна част от вашия Web сайт.
6. (По-голямо предизвикателство) Създайте двойка client/server програми които използват датаграми за изпращане на файл от една машина на друга. (Виж описането на края на секцията за датаграмите на края на тази глава.)
7. (По-голямо предизвикателство) Вземете програмата **VLookup.java** така че когато кликнете в резултиращото име то се взема и изпраща в клипборда (така че просто го пускате в електронната си поща). Ще трябва да

погледнете пак в главата за IO потоците за да си припомните как да използвате клипборда на Java 1.1.

16: Програмиране за фирмата

ЗАБЕЛЕЖКА: Бележки по тази глава да се отправят към dbartlett@pobox.com

Обработките в предприятието са събиране и разпределение на информация.

Това се прави чрез създаване на общи складове (отделни точки за достъп) за тази информация и позволяване на хората да се добират до нея по много начини. Така обработките за предприятието са създаване и манипулиране на споменатите складове и даване на хората възможност да разглеждат и манипулират информацията.

В тази глава ще видите някои начини да се постигне това.

Отваряща фраза

Въвеждащи параграфи

((Дейв: спомена архипримера който ще се използва в тази глава. Бих препоръчал "Community Information System" (CIS?) с която работихме за семинара за сървлетите. Целта ще е да се доставят различни услуги полезни за общината (разбира се това лесно би могло да се адаптира за други нужди, но мисля че ще е лесно просто да помним "община" като проектантски проблем). Например:

- Общински календар/разписание на предстоящи събития (Би могло да се започне със сървleri и текстов файл, после да се добави JDBC, после да се направи с EJB)
- *Указатели на общината (телефонни/за електронна поща)
- *Списъци по интереси за общината (или навсякъде "общността", ако предпочитате - б.пр.)
- Общи обяви/Съобщения/Новини
- Забравено & Намерено
- Продажба на билети за събития с общинско значение (тук ще има въпроси на сигурността)
- Подредени обяви

*Възможен допълнителен пример (може би за EJB) е анти-спаминг система – или парола, или изпраща информация обратно към вас, след като провери дали вашият адрес е на член на общината.

Java Database Connectivity (JDBC)

Пресметнато е, че около половината софтуер е свързан с клиент/сървър операции. Голямо обещание на Java беше да се даде възможност за независими от платформата клиент/сървър операции с бази данни. В Java 1.1 това се прави с Java DataBase Connectivity (JDBC).

Един от главните проблеми с базите данни са били войните между доставчиците. Има "стандартен" език за бази данни, Structured Query Language (SQL-92), но обикновено трябва да си знаете доставчика на СУБД напук на стандарта. JDBC е проектиран да бъде независим от платформата, така че не е необходимо да се притеснявате за това каква база данни използвате когато програмирате. Остава възможността обаче да правите специфични за доставчика извиквания от JDBC така че не сте прекалено ограничени.

JDBC, както много оте APIтата в Java, е проектирано да бъде просто. Виканията на методи кореспондират с логиката на работа с бази данни: свързване с базата, създаване на оператор и изпълнение на заявката разглеждане на резултантното множество.

За да позволи тази независимост от платформата, JDBC има *управител на драйверите* който динамично поддържа всички обекти на драйверите на базите данни с които работите. Така че ако имате да се свързвате с бази данни на трима различни доставчика, ще ви трябват три отделни драйверски обекта. Последните се регистрират в управителя по време на товаренето си и то може да се направи принудително с `Class.forName()`.

За да отворите база данни, трябва да създадене "URL за база данни" който посочва:

1. Че използвате JDBC с "jdbc"
2. "Подпротокол": името на драйвер или името на механизма за свързване с базата данни. Тъй като дизайнът на JDBC бе вдъхновен от ODBC, първият субпротокол е "jdbc-odbc bridge," посочван с "odbc"
3. Идентификатор на базата данни. Това варира с използвания драйвер, но изобщо дава логическо име което се обработва от програмата-администратор на базата данни до физическа директория където са разположени таблиците на базата данни. За да има някакво значение вашият идентификатор на базата данни, трябва да регистрирате името чрез софтуера за администрация на базата данни. (Процесът на регистрация варира от платформа към платформа.)

Всичката тази информация е комбинирана в един стринг, "URL-то на базата данни."

Например за свързване чрез субпротокола ODBC към база данни посочена като "people," той би бил:

```
| String dbUrl = "jdbc:odbc:people";
```

Ако се свързвате през брежка URL също ще съдържа информация посочваща отдалечената машина.

Когато сте готови за свързване с базата данни, викате **static** метода `DriverManager.getConnection()`, давайки му URL на базата, потребителско име и парола за да влезете в базата. Получавате обратно `Connection` обект който после може да използвате за питания и манипулации в базата.

Следващият пример отваря база данни с информация за контакти и гледа за фамилино име дадено на командния ред. Селектира само имената на хората имащи адрес за електронна поща, после извежда тези кито имат споменатата фамилия:

```

//: c16:Lookup.java
// Looks up email addresses in a
// local database using JDBC
import java.sql.*;

public class Lookup {
    public static void main(String[] args) {
        String dbUrl = "jdbc:odbc:people";
        String user = "";
        String password = "";
        try {
            // Load the driver (registers itself)
            Class.forName(
                "sun.jdbc.odbc.JdbcOdbcDriver");
            Connection c = DriverManager.getConnection(
                dbUrl, user, password);
            Statement s = c.createStatement();
            // SQL code:
            ResultSet r =
                s.executeQuery(
                    "SELECT FIRST, LAST, EMAIL " +
                    "FROM people.csv people " +
                    "WHERE " +
                    "(LAST=" + args[0] + ") " +
                    "AND (EMAIL Is Not Null) " +
                    "ORDER BY FIRST");
            while(r.next()) {
                // Capitalization doesn't matter:
                System.out.println(
                    r.getString("Last") + ", "
                    + r.getString("FIRST")
                    + ":" + r.getString("EMAIL"));
            }
            s.close(); // Also closes ResultSet
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
} ///:~

```

Може да се види, че създаването на URL-то на базата данни става както беше вече описано. В този пример няма парола за достъп до базата данни така че потребителското име и паролата са празни стрингове.

Щом веднъж се направи връзка с `DriverManager.getConnection()` може да използвате получения `Connection` за създаване на `Statement` обект използвайки метода `createStatement()`. С получения `Statement` може да извикате `executeQuery()`, подавайки стринг, съдържащ SQL оператор според стандарта SQL-92. (Скоро ще видите как да генерирате оператора автоматично, така че няма нужда да знаете много за SQL.)

Методът `executeQuery()` връща `ResultSet` обект, който малко прилича на итератор: методът `next()` мести итератора на следващия запис на оператора или връща `false` ако е достигнат края на резултантното множество. Винаги ще вземате `ResultSet` връщан от `executeQuery()` даже и ако множеството на резултатите е празно (that is, an exception is not thrown to `est`, не се изхвърля изключение). Забележете че трябва да извикате `next()` веднъж преди всянакъв опит да четете запис данни. Ако резулиращото множество е празно това първо извикване на `next()` ще върне `false`. За всеки запис в резултантното множество може да изберете полетата

(покрай други възможни подходи) като използвате името на полето като стринг. Забележете също че капитализацията на името на полето се игнорира – тя няма значение в една SQL база данни. Типът на връщаното се определя чрез викане на `getInt()`, `getString()`, `getFloat()` и т.н.. В тази точка сте взели вашите данни от базата данни в естествения за Java формати може да ги правите каквото си искате използвайки обикновен Java код.

Каране примера да работи

При JDBC разбирането на кода е относително просто. Смущаващата част от работата е да го накарате да работи на вашата конкретна система. Причината да е смущаваща е че трябва да направите вашият JDBC да се товари правилно и да установите както трябва вашата база данни чрез софтуера за нейната администрация.

Разбира се, този процес може да е драматично различен за различни машини, но процесът който използвах при 32-битов Windows би могъл да ви даде посоки как да атакувате вашата система.

Стъпка 1: Намиране JDBC драйвера

Горната програма съдържа оператора:

```
| Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Това предполага директорна структура, но е измамно. С тази конкретна инсталация на JDK 1.1 няма файл наречен **JdbcOdbcDriver.class**, така че ако сте погледнали примера и тръгнете да търсите ще останете разочаровани. Други публикувани примери използват псевдоиме, такова като "myDriver.ClassName," което е по-малко от полезно. Фактически зареждащият оператор по-горе за jdbc-odbc драйвер (единственият който пристига с JDK 1.1) го има само на няколко места в онлайн документацията (в частност на страницата с етикет "JDBC-ODBC Bridge Driver"). Ако горният оператор не работи, името може да е било променено като част от промени във версията на Java, така че ще огледате документацията пак.

Ако товарещия оператор не е на ред, ще получите изключение. За да видите дали товарещия оператор работи правилно, коментирайте кода след него до **catch** клаузата; ако програмата не изхвърля изключения значи операторът работи правилно.

Стъпка 2: Конфигуриране на базата данни

Отново, специфично за 32-битов Windows; Може да трябва някакво изследване за вашата конкретна система.

Първо отворете control panel. Може да намерите две икони "ODBC." Трябва да използвате тази която е "32bit ODBC," понеже другата е за съвместимост назад със 16-битов ODBC софтуер и няма да даде резултат с JDBC. Като отворите иконата "32bit ODBC" ще видите диалог с няколко ушенца, включително "User DSN," "System DSN," "File DSN" и т.н. където "DSN" значи "Data Source Name." Излиза че за JDBC-ODBC моста, единственото място за настройка на вашата база данни е "System DSN," но вие също ще искате да тествате нещата си и да задавате въпроси, а за това ще трябва да поставите базата си и във "File DSN." Това ще позволи на Microsoft Query tool-а (който идва с Microsoft Office) да намери базата данни. Забележете че и други query tools могат да се намерят от други доставчици.

Най-интересната база данни е тази, която вече използвате. Стандартния ODBC поддържа много различни файлови формати включително такива почтени работяги като DBase. Обаче също включва и "разделен със запетая ASCII" формат, който практически всеки инструмент в базите данни може да пише. В моя случай аз просто казах моята база данни "people" която бах строил години с различни инструменти и я експортирах като ASCII файл с разделител запетая (типичното разширение за такива е **.csv**). Във "File DSN" секцията избрах "Add,"

избрах текстовия драйвер за моя ASCII файл, а после махнах отметката на "use current directory" за да мис се позволи да посоча директорията където беше експортирания файл.

Забележете, че не се посочва файл, само директория. Така е защото обикновено базата данни се състои от множество от файлове в една директория (макар и да може да е по друг начин, също така). Всеки файл обикновено съдържа една таблица и SQL операторите могат да дадат резултат, който е изсмукан от много таблици в базата данни (това се нарича *join*). База данни която съдържа само една таблица (както тази) обикновено се нарича *flat-file база данни*. Повечето проблеми които надхвърлят простото добавяне и четене на данни довеждат до много таблици които изискват "присъединявания" за да се получат желаните резултати, тези се наричат *релационни бази данни*.

Стъпка 3: Проба на конфигурацията

За да се тества конфигурацията трябва начин да се види дали базата данни е видима от програмата, която ще задава въпроси. Разбира се, може да пуснете горния пример на JDBC включвайки оператора:

```
Connection c = DriverManager.getConnection(  
    dbUrl, user, password);
```

Ако се изхвърли изключение, конфигурацията не е коректна.

Полезно е обаче да се намеси в тази точка генератор на отчети. Аз използвах Microsoft Query което е от Microsoft Office, но вие може да предпочетете някакъв друг. Генераторът трябва да знае къде е базата данни, Microsoft Query изисква да отида във "File DSN" паното на ODBC администратора и да добавя нова позиция там, отново посочвайки драйвера за текст и директорията където е базата данни. Може да я наречете както искате, но е полезно да използвате същото име като "System DSN."

След като направите това, ще видите че вашата база данни е видима когато използвате генератор на отчети.

Стъпка 4: Генерация на SQL поръчка

Заявката която направих с Microsoft Query не само показва че базата ми данни е на мястото си и в добро състояние, но също и автоматично създаде SQL който ми беше необходим да вмъкна в моята Java програма. Аз исках заявка за търсене по фамилия зададена на командния ред когато започнах Java програмата. Така като за отправна точка търсих специфично име, 'Eckel'. Исках също да се изведат онези имена които имат адреси на електронна поща асоциирани с тях. Стъпките за създаване на тази заявка бяха:

1. Започнете ново използвайки Query Wizard. Изберете база данни "people". (Това е еквивалентно на отваряне на базата данни чрез подходящ URL на база данни.)
2. Изберете таблицата "people" в базата данни. Извътрите на таблицата изберете колоните FIRST, LAST и EMAIL.
3. За "Filter Data," изберете LAST и "equals" с аргумент: Eckel. Кликнете "And" радио бутона.
4. Изберете EMAIL и определете да бъде "Is not Null."
5. В "Sort By," изберете FIRST.

Резултатът от това запитване ще покаже дали ще получите каквото искате.

Сега може да натиснете SQL бутона и без всякакво усилие от ваша страна ще изскочи SQL кода, готов за копиране в паметта и от там - в програмата ви. За това запитване изглежда така:

```
| SELECT people.FIRST, people.LAST, people.EMAIL
```

```
FROM people.csv people
WHERE (people.LAST='Eckel') AND
(people.EMAIL Is Not Null)
ORDER BY people.FIRST
```

С по-сложни запитвания е лесно да се събърка, но с този инструмент може интерактивно да правите и тествате кода си. Трудно е защитима гледната точка, че това трябва да се прави ръчно.

Стъпка 5: Променете и вмъкнете запитването си

Ще забележите, че кодът по-горе изглежда различно от този в програмата. Това е защото генераторът на отчети използва пълно квалифицирани имена навсякъде, даже и когато се работи с една таблица. (Когато се работи с повече от една таблица, квалификацията предотвратява колизии с колони които имат същите имена но са от различни таблици.) Понеже тук табличата е една, може ако искате да махнете квалификатора "people" от повечето имена, ето така:

```
SELECT FIRST, LAST, EMAIL
FROM people.csv people
WHERE (LAST='Eckel') AND
(EMAIL Is Not Null)
ORDER BY FIRST
```

Освен това не искате програмата да бъде твърдо кодирана да търси едно единствено име. Вместо това тя трябва да търси име зададено като аргумент на командния ред. Правенето на тези промени и превръщането на SQL оператора в динамично създаван **String** дава:

```
"SELECT FIRST, LAST, EMAIL " +
"FROM people.csv people " +
"WHERE " +
"(LAST='" + args(0) + "') " +
"AND (EMAIL Is Not Null) " +
"ORDER BY FIRST";
```

SQL има друг начин да вмъква имена в запитванията наречен *stored procedures*, който е полезен за скоростта. Но за пръв опит и експерименти с базата данни, много си е добре вграждането на стринговете-запитвания в Java.

Може да видите от този пример че чрез използването на текущо достъпни инструменти – в частност генератора на отчети – програмирането на бази данни със SQL и JDBC може да бъзе доста праволинейно.

GUI версия на преглеждащата програма

По-полезно е да се остави преглеждащата програма постоянно пусната и когато трябва да се превключваме към нея като посочваме името което да се търси. Следната програма създава търсещата програма като приложение/аплет и също добавя изваждане на предполагаемата останала част от името за да не ви кара да го пишете цялото:

```
//: c16:VLookup.java
// GUI version of Lookup.java
// <applet code=VLookup
// width=500 height=200> </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;
import java.sql.*;
```

```

public class VLookup extends JApplet {
    String dbUrl = "jdbc:odbc:people";
    String user = "";
    String password = "";
    Statement s;
    JTextField searchFor = new JTextField(20);
    JLabel completion =
        new JLabel("          ");
    JTextArea results = new JTextArea(40, 20);
    public void init() {
        searchFor.getDocument().addDocumentListener(
            new SearchL());
        JPanel p = new JPanel();
        p.add(new Label("Last name to search for:"));
        p.add(searchFor);
        p.add(completion);
        Container cp = getContentPane();
        cp.setLayout(new BorderLayout());
        cp.add(p, BorderLayout.NORTH);
        cp.add(results, BorderLayout.CENTER);
        try {
            // Load the driver (registers itself)
            Class.forName(
                "sun.jdbc.odbc.JdbcOdbcDriver");
            Connection c = DriverManager.getConnection(
                dbUrl, user, password);
            s = c.createStatement();
        } catch(Exception e) {
            results.setText(e.getMessage());
        }
    }
    class SearchL implements DocumentListener {
        public void changedUpdate(DocumentEvent e){}
        public void insertUpdate(DocumentEvent e){
            textValueChanged();
        }
        public void removeUpdate(DocumentEvent e){
            textValueChanged();
        }
    }
    public void textValueChanged() {
        ResultSet r;
        if(searchFor.getText().length() == 0) {
            completion.setText("");
            results.setText("");
            return;
        }
        try {
            // Name completion:
            r = s.executeQuery(
                "SELECT LAST FROM people.csv people " +
                "WHERE (LAST Like " + +
                searchFor.getText() + +
                "%') ORDER BY LAST");
            if(r.next())
                completion.setText(

```

```

        r.getString("last"));
r = s.executeQuery(
"SELECT FIRST, LAST, EMAIL " +
"FROM people.csv people " +
"WHERE (LAST=('" +
completion.getText() +
") AND (EMAIL Is Not Null) " +
"ORDER BY FIRST");
} catch(Exception e) {
results.setText(
    searchFor.getText() + "\n");
results.append(e.getMessage());
return;
}
results.setText("");
try {
while(r.next()) {
results.append(
    r.getString("Last") + ", "
    + r.getString("FIRST") +
    ":" + r.getString("EMAIL") + "\n");
}
} catch(Exception e) {
results.setText(e.getMessage());
}
}
public static void main(String[] args) {
JApplet applet = new VLookup();
JFrame frame = new JFrame("Email lookup");
frame.addWindowListener(
new WindowAdapter() {
    public void windowClosing(WindowEvent e){
        System.exit(0);
    }
});
frame.add(applet);
frame.setSize(500, 200);
applet.init();
applet.start();
frame.setVisible(true);
}
} //:~

```

Повечето логика относно базата данни е същата, но може да видите че е добавен **TextListener** за да слуша **TextField**, така че винаги когато въведете нов знак той първо се опитва да завърши името гледайки последното име в базата данни използвайки първото което се покаже. (Слага го в **completion Label** и го използва като текст за оглеждане.) По този начин, щом сте въввели достатъчно знаци за еднозначно определяне на името може да спрете.

Защо API-то на JDBC изглежда така сложно

Като прегледате онлайн документацията за JDBC може да се уплашите. В частност, в интерфейса **DatabaseMetaData** – той е просто огромен, за разлика от кратките интерфейси които сте виждали в Java – има методи като **dataDefinitionCausesTransactionCommit()**,

`getMaxColumnNameLength()`, `getMaxStatementLength()`, `storesMixedCaseQuotedIdentifiers()`, `supportsANSI92IntermediateSQL()`, `supportsLimitedOuterJoins()` и така нататък. За какво е всичко това?

Както бе споменато вече, базите данни от самото си начало са били в постоянно завихряне, най-вече защото търсенето на СУБД и с това - на съответните инструменти, е толкова голямо. Чак напоследък има някаква сходимост към общи признати езици като SQL (и има множество други езици за бази данни в употреба). Но даже с SQL "стандарта" има толкова много вариации на темата, че JDBC трябва да даде големия **DatabaseMetaData** интерфейс така че вашият код да може да открие всичките възможности на "стандартна" SQL база данни към която е текущо закачен. Накъсо, може да пишете късо, транспортируемо SQL, но ако искате да оптимизирате бързината вашият код стремително ще се умножава като използвате повече и повече специални възможности давани от конкретния производител.

Това, разбира се, не е неуспех на Java. Несъответствията между продуктите за бази данни са просто нещо, за което JDBC се опитва да помогне да се справите с него. Но помнете че животът ви ще е по-лесен ако правите родови запитвания и не се тревожите много за бързината или, ако трябва да настроите бързината, знайте платформата на която това ще се прави така че да не трябва да пишете всички онзи изследващ код.

Повече информация за JDBC има в електронните документи които идват с Java 1.1 дистрибуиран от Sun. Може да намерите повече в книгата *JDBC Database Access with Java* (Hamilton, Cattel, and Fisher, Addison-Wesley 1997). Други JDBC книги се появяват редовно.

СървлеТИ

Традиционния начин да се отработи такъв проблем е да се напише HTML с текстово поле и "submit" бутон. Потребителят може да напише каквото си иска в текстовото поле и то ще бъде изпратено на сървъра без въпроси. Като изпрати данните, Web страницата казва също на сървъра какво да прави с тях чрез Common Gateway Interface (CGI) програма която сървърът ще пусне след получаването на данните. Тази CGI програма типично е написана или на Perl или C (или понякога на C++, ако сървърът го поддържа) и тя трябва да обработи всичко. Първо поглежда данните и решава дали са в правилен формат. Ако не, CGI програмата трябва да създаде HTML страница за да опише проблема; тази страница се обработва от сървъра, който я изпраща обратно на потребителя. Потребителят постле трябва да се съобрази с нея и- отначало. Ако данните са коректни, CGI програмата отваря данновия файл и или добавя електронния адрес към файла или открива че вече го има. И в двата случая трябва да оформи подходяща HTML страница на сървъра за да се изпрати на потребителя.

Като Java програмисти това за нас е тромав начин да се реши проблема и естествено ще искаме всичко да стане на Java. Първо ще използваме Java аплет да се грижи за валидността на данните на клиентската страна, без този целия досаден Web трафик и форматиране на страници. После нека да пропуснем Perl CGI скрипта в полза на Java приложение работещо на сървъра. Фактически нека да пропуснем целия Web сървър в комплект и да направим нашата връзка между клиентската страна и Java приложението на сървъра!

Както ще видите, има няколко въпроса които правят проблема по-сложен, отколкото изглежда. Идеално би било да напишете аплета чрез Java 1.1 но едва ли е практично. По времето когато се пише това, Java 1.1-способни браузъри се използват от малко хора и макар и такива да са широко достъпни вече, ще трябва да имате предвид, че вероятно много хора няма да бързат да минат на такива. Така че за по-сигурно аплетът ще се направи само на Java 1.0. С това наум няма да има JAR файлове за комбиниране на **.class** файловете в аплета, така че аплетът ще бъде проектиран да има минимален брой **.class** доколкото е възможно за да се намали времето за разтоварване.

ОСНОВНИЯ СЪРВЛЕТ

Въвеждащи примери за Java семинар които може да използвате или променяте

Работи с GET & POST:

```
//: c16:ServletsRule.java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class ServletsRule extends HttpServlet {
    int i = 0; // Servlet "persistence"
    public void service(HttpServletRequest req,
        HttpServletResponse res) throws IOException {
        PrintWriter out = res.getWriter();
        out.print("<HEAD><TITLE>");
        out.print("A server-side strategy");
        out.print("</TITLE></HEAD><BODY>");
        out.print("<h1>Servlets Rule! " + i++);
        out.print("</h1></BODY>");
        out.close();
    }
} ///:~
```

Лесно се получава HTML от данни:

```
//: c16:EchoForm.java
// Dumps the name-value pairs of any HTML form
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class EchoForm extends HttpServlet {
    public void service(HttpServletRequest req,
        HttpServletResponse res) throws IOException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.print("<h1>Your form contained:</h1>");
        Enumeration flds = req.getParameterNames();
        while(flds.hasMoreElements()) {
            String field = (String)flds.nextElement();
            String value = req.getParameter(field);
            out.print(field + " = " + value + "<br>");
        }
        out.close();
    }
} ///:~
```

Един сървлет, нова нишка при всяка заявка:

```
//: c16:ThreadServlet.java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class ThreadServlet extends HttpServlet {
    int i;
    public void service(HttpServletRequest req,
```

```

HttpServletResponse res) throws IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    synchronized(this) {
        try {
            Thread.currentThread().sleep(5000);
        } catch(InterruptedException e) {}
    }
    out.print("<h1>Finished " + i++ + "</h1>");
    out.close();
}
} ///:~

```

Java Server Pages

Ето един максимално прост JSP който използва стандартна библиотечно извикване на Java за да вземе текущото време в милисекунди, което после се дели на 1000 за да се получи времето в секунди. Понеже единин JSP израз (виж `<%=`) е използван, резултатът от изчислението е усмирен в **String** така че може да бъде изведен към **out** обект (който го слага в web страницата):

```

//:! c16:jsp>ShowSeconds.jsp
<html><body>
<H1>The time in seconds is:
<%= System.currentTimeMillis()/1000 %></H1>
</body></html>
///:~

```

Основни операции

```

//:! c16:jsp>Hello.jsp
<%-- This JSP comment will not appear in the
generated html --%>
<%-- This is a JSP directive: --%>
<%@ page import="java.util.*" %>
<%-- These are declarations: --%>
<%! long loadTime= System.currentTimeMillis(); %>
<%! Date loadDate = new Date(); %>
<%! int hitCount = 0; %>
<html><body>
<!-- This HTML comment will appear as a comment
on the generated html page -->
<%-- The following comment has the result of a
JSP expression inserted in the generated html;
the '=' indicates a JSP expression --%>
<!-- This page was loaded at <%= loadDate %> -->
<H1>Hello, world! It's <%= new Date() %></H1>
<H2>Here's an object: <%= new Object() %></H2>
<H2>This page has been up
<%= (System.currentTimeMillis()-loadTime)/1000 %>
seconds</H2>
<H3>Page has been accessed <%= ++hitCount %>

```

```

times since <%= loadDate %></H3>
<%-- A "scriptlet" which writes to the server
console. Note that a ';' is required: --%>
<% System.out.println("Goodbye"); %>
</body></html>
///:~

```

Не може да вмествате JSP коментари ('`<%--`' и '`--%>`') вътре в други JSP директиви, но може да използвате коментари извън JSP директивите за да се предотврати опита за компилирането им.

Вграждане на блокове код

```

//:! c16:jsp:PageContext.jsp
<%--Viewing the attributes in the pageContext--%>
<%-- Note that you can include any amount of code
inside the scriptlet tags --%>
<%@ page import="java.util.*" %>
<html><body>
<%
Enumeration e =
pageContext.getAttributeNamesInScope(1);
while(e.hasMoreElements()) {
out.println("\t<li>" +
e.nextElement() + "</li>");
}
%>
<H4>End of list</H4>
</body></html>
///:~

```

Извличане на полета и стойности

Този пример първо гледа дали има някакви параметри в изпратената форма; ако няма знае се че JSP се вика за пръв път и че трябва да генерира формата (това е удобна техника която позволява да се сложи генераторът на формите и отговора в един сорсов файл). После се извеждат полетата и техните стойности (тази втора секция няма да направи нищо при първото викане на JSP).

```

//:! c16:jsp:DisplayFormData.jsp
<%-- Fetching the data from an HTML form --%>
<%-- Also generates the form --%>
<%@ page import="java.util.*" %>
<html><body>
<H1>DisplayFormData</H1><H3>
<%
Enumeration f = request.getParameterNames();
if(f.hasMoreElements()) { // No fields
%>
<form method="POST" action="DisplayFormData.jsp">
<%
for(int i = 0; i < 10; i++) {

```

```

%>
Field<%=i%>:
<input type="text" size="20"
      name="Field<%=i%>" value="Value<%=i%>"><br>
<% } %>
<INPUT TYPE=submit name=submit Value="Submit">
</form>
<% } %>

<%
Enumeration flds = request.getParameterNames();
while(flds.hasMoreElements()) {
  String field = (String)flds.nextElement();
  String value = request.getParameter(field);
%>
<li><%= field %> = <%= value %></li>
<%
}
%>
</H3></body></html>
///:~

```

Манипулиране на сесиите в JSP

```

//:! c16:jsp:SessionObject.jsp
<%--Setting and getting session object values--%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H1>This session was created at
<%= session.getCreationTime() %></H1>
<H3>MaxInactiveInterval=
<%= session.getMaxInactiveInterval() %></H3>
<% session.setMaxInactiveInterval(5); %>
<H3>MaxInactiveInterval=
<%= session.getMaxInactiveInterval() %></H3>
<H1>Session value for "My dog"
<%= session.getValue("My dog") %></H1>
<% session.putValue("My dog",
  new String("Ralph")); %>
<H1>My dog's name is
<%= session.getValue("My dog") %></H1>
<FORM TYPE=POST ACTION=SessionObject2.jsp>
<INPUT TYPE=submit name=submit Value="Submit">
</FORM>
</body></html>
///:~

```

```

//:! c16:jsp:SessionObject2.jsp
<%--The session object carries through--%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H1>Session value for "My dog"
<%= session.getValue("My dog") %></H1>
<% session.invalidate(); %>

```

```

</body></html>
///:~

//:! c16:PlayWithCookies.jsp
<%--This program has different behaviors under
different browsers! --%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<%
Cookie() cookies = request.getCookies();
for(int i = 0; i < cookies.length; i++) { %>
Cookie name: <%= cookies(i).getName() %> <br>
value: <%= cookies(i).getValue() %><br>
Max age in seconds:
<%= cookies(i).getMaxAge() %><br>
<% cookies(i).setMaxAge(3); %>
Max age in seconds:
<%= cookies(i).getMaxAge() %><br>
<% response.addCookie(cookies(i)); %>
<% } %>
<%-- <% response.addCookie(
    new Cookie("Bob", "Car salesman")); %> --%>
</body></html>
///:~

```

Предаване управлението на други сървлети

```

//:! c16:RequestDispatcher1.jsp
<%--Using the request dispatcher forward()
to pass control to another servlet --%>
<html><body><H1>In RequestDispatcher 1</H1>
<% response.setContentType("text/html");
out.println(
    "Printing to response object from RD 1");
RequestDispatcher rd =
    application.getRequestDispatcher(
        "/jsp/RequestDispatcher2.jsp");
rd.forward(request, response);
out.println("<H1>Hello</H1>");
%>
</body></html>
///:~

```

```

//:! c16:RequestDispatcher2.jsp
<%--Recieves a dispatched request --%>
<html><body>
<% response.setContentType("text/html");
out.println(
    "Printing to response object from RD 2");
%>

```

```

<H1>In RequestDispatcher 2</H1>
</body></html>
///:~


//:! c16:RequestDispatcher3.jsp
<%--Using a RequestDispatcher.include() --%>
<html><body>
<H1>In RequestDispatcher 3</H1>
<% response.setContentType("text/html");
out.println(
    "Printing to response object from RD 3");
out.println("<H1>In RD 3</H1>");
RequestDispatcher rd =
    application.getRequestDispatcher(
        "/Jsp/RequestDispatcher2.jsp");
rd.include(request, response);
out.println("<H1>Out RD 3</H1>");
%>
<H1>Hello</H1>
</body></html>
///:~

```

RMI във фирмата

Corba във фирмата

Enterprise Java Beans (EJB)¹

Спецификацията Enterprise JavaBeans (**EJB**) определя компонентен модел и среда за разгръщане за да опрости жизнения цикъл на многоредови разпределени системи. Enterprise JavaBeans е спецификация дадена от JavaSoft, утвърдена и използвана от много доставчици. Това не е реален продукт. Един построен съгласно EJB продукт е такъв, който прилага спецификацията.

Целта на EJB спецификацията е да "направи лесно писането на (на сървърската страна) приложения". Спецификацията EJB достига тази цел чрез определяне на множество от шаблони за проектиране за сървърската страна и също така ролите, които различните разработчици, администратори и системни компоненти играят в различните фази от цикъла на живот на разпределеното приложение.

Чрез използването на определено множество от шаблони и роли - които развързват работната логика от подлежащата инфраструктура - продуктите които отговарят на спецификацията EJB правят възможно да се пишат продукти за цяло предприятие без да е необходимо да се разбираат всичките тези въпроси на инфраструктурата като транзакции, механизми за кеширане, многонишковост и сигурност.

Определянето на тези роли и компоненти също дава на разработчика независимост от доставчиците. EJB са се обединили около една спецификация, поради което Enterprise Java Bean компонентите са лесно транспортируеми между инструментите от различни

¹ This section was contributed by Robert Castaneda (robertc@altavista.net)

разработчици и платформи и отговарят на обещанието на Java за "Write Once, Run Anywhere™" ("Пиши Веднъж, Пускай Навсякъде™" - б.пр.) философия.

Макар че EJB спецификацията е спецификация на Java основана на и реализирана с езика Java, тя също отределя възможности за опериране с не-Java системи. Също и с CORBA спецификацията.

EJB е може би най-видимата от многото API която излага Java 2, Enterprise Edition спецификацията. Класовете и интерфейсите които съставят Enterprise Java Beans API са определени в стандартния пакет за разширение **javax.ejb**. Последната версия на EJB спецификацията е 1.1 и е достъпна от <http://java.sun.com/products/ejb>.

Защо ни трябва EJB?

Когато се конструира многоредова система с използване на RMI, CORBA или друг мидълуеър, системните архитекти и разработчици се срещат с много предизвикателства като:

- Как да се осигури че приложението ще работи?
- Как да се осигури скалируемост на приложението?
- Как да се осигури управляемост на приложението, дава ли то 24x7 достъпност?
- Как да се прави достъп до различни източници на данни в единна форма?
- Как ще се обработват разпределените транзакции?
- Как ще се осигури идентификацията на потребителите и сигурността?
- Как да се създадат повторно използваеми компоненти и даже да се използва код, написан от други?
- Как ще се управлява жизнения цикъл на създаваните компоненти?

Всички тези въпроси съзникнали даже преди да обхванем проблема, който има да се решава! Както се вижда, има много неща за които да мисли разработчикът на разпределени системи. Спецификацията Enterprise JavaBeans признава тези проблеми и определя как всичките или част от тях да се заобиколят или решат.

И RMI и CORBA позволяват разработката на разпределено приложение, без да навлизаме в разгорещени дебати коя технология е по-добра (каквото може да се намерят в други книги). Разпределени протоколи и API-та като RMI и CORBA позволяват на разработчика да създаде свои собствени компоненти. Обаче проектирането на тези компоненти е оставено на въпросния разработчик. Което значи че компонентите създавани от един тим за конкретен сървър не бяха повторно използваеми и тези които бяха, обикновено се основаваха на работни рамки, зависещи от конкретните механизми на кеширане, система за сигурност и т.н. на сървъра.

Както все повече и повече системи ставаха многоредови (или многосвързани, ако предпочитате - б.пр.) и различните доставчици създаваха свои решения както се описа по-горе, това започна да приковава разработчиците към приложениета на един конкретен доставчик. Понеже беше много тредна миграцията към друг.

EJB още повече рафинира нещата чрез определяне как да се построи средния ред. Целта е разработчика да не трябва да се беспокои и за механизмите от ниско ниво като Разпределени транзакции, На сигурността и Кеширащи. И понеже всички създен код отговаря на стандарта, код написан за един сървър трябва да може да работи на друг сървър, който отговаря на EJB, без прекомпилиране.

EJB позволи създаването на пазар за компоненти които се създават и дистрибутират от експерти в даден домейн.

EJB решава проблема чрез създаване на стандартна RUNTIME среда, това е EJB Container. EJB Container-а обработва и скрива проблемите от бийн разработчика така че последният може да разчита че услугите са налице и може да се съсредоточи върху основния проблем, а също не е обвързан с архитектурата на конкретен разработчик.

Как да сложа 'Е'-то на съществуващите мои JavaBeans?

Има доста недоразумение относно отношението между JavaBeans компонентния модел и Enterprise JavaBeans спецификацията. Докато и JavaBeans и Enterprise JavaBeans спецификациите споделят общи цели за постигане на повторна използваемост на Java код при следите за настройка и пускане чрез използване на подходящи шаблони за проектиране, мотивите зад всяка от спецификациите са насочени към различни видове проблеми.

Стандартите определени в компонентния модел на JavaBeans са проектирани за създаване на повторно използваеми компоненти които типично се използват в IDE развойни среди и са общо взето, макар и не изключително, визуални компоненти.

Спецификацията Enterprise JavaBeans определя компонентен модел за програмиране на сървърската страна с Java код. Enterprise JavaBean може да използва JavaBeans в своята реализация.

Компоненти в Enterprise Java Beans

Какви компоненти определя EJB? Какви други J2EE компоненти се изискват?

EJB Сървър

EJB Server е определен като Application Server който съдържа и пуска 1 или повече EJB Containeri. Дефиницията на интерфейс между EJB Server и EJB Container в момента не е налична в EJB спецификацията. Спецификацията съветва и Container и Server да са от един доставчик.

EJB Container

EJB спецификацията въвежда концепцията за EJB Container (container). Containerът е среда за работа която съдържа компоненти (Enterprise JavaBeans) и доставя стандартни услуги за компонентите. Отговорностите на EJB Container са тясно определени от спецификацията за да се осигури независимост от доставчика. EJB container е отговорен за доставката на някои технически ползи от EJB, включително разпределени транзакции, сигурност, управление на жизнения цикъл на бийновете, кеширане, трдинг и управление на сесиите.

EJB Container постига тези задачи като действа като "лепило" между EJB и клиента който използва услугите на Beans. Поради това ниво на опосредствяване контейнерът може прозрачно да кешира бийновете и да разпространява транзакциите и информация свързана със сигурността без потребителския код да има нещо общо с ниското ниво.

Enterprise JavaBeans

Enterprise Java Beans са повторно използваеми компоненти които са разработени в съгласие с шаблоните за проектиране изложени в спецификацията EJB. EJBBeans могат да бъдат от 2 различни типа, Entity Beans и Session Beans. Версия 1.0 на EJB спецификацията не изискваше Entity Beans да бъдат приложени, това е променено в 1.1, която прави задължително

прилагането на Entity Beans. EJB 1.1 спецификацията също още поизчисти някои двусмислени въпроси около Session Beans. EJB Container е отговорен да даде кеширането, управлението на жизнения цикъл и сесиите на всичките EJBове така че достъвчикът на бийнове да може да се съсредоточи на основния проблем.

EJB определя 2 шаблона или типове Bean, Session Bean-ове и Entity Bean-ове. Фиг. x. показва йерархията на EJB типовете

< author note >

покажи прост чертеж на йерархията EJB -> Entity Bean/Session Bean -> BMP/CMP,
Stateful/Stateless

</author note>

Session Beans

Session Beans са компоненти които работят в полза на клиента и доставят тези услуги. Session Beans могат да бъдат или Статусни или Безстатусни и обслужват само един клиент. Session Beans представят операциите върху устойчивите данни, Stateful Session Bean държи данните на конкретен клиент между извикванията на методи. Stateless Session е безстатусен и може да бъде използван повторно от различни клиенти между заявките тъй като не се води статус в бийна

Entity Beans

Entity Beans са компоненти, които представляват устойчиви данни например такива от база данни, и поведението им. Устойчивостта в Entity Beans може да бъде управлявана от разработчика, това е известно като "Bean Managed Persistence" или тя може да бъде управлявана от EJB Container, възможно със съдействието на инструмента Object to Relational Mapping. Този подход е известен като "Container Managed Persistence". В много клиенти могат да споделят бийновете-същности, контейнерът се занимава с въпросите на многонишковостта. Понеже Entity представлява устойчиви данни, Entity Bean надживява контейнера.

Bean Managed Persistence

Bean Managed Persistence или BMP е когато доставчикът на бийна е отговорен за цялата логика на създаването на EJB, осъвременяване на порета на Entity Bean, изтриване на Entity Bean и намиране на Entity Bean в устойчиво хранилище. Това обикновено довежда до написването на JDBC код за взаимодействие с база данни. С BMP, разработчикът има пълен контрол върху устойчивостта на Entity Bean.

BMP също дава гъвкавост където CMP реализация може да не е налична, например ако желаете да създадете EJB който обхваща код на съществуваща голяма машина, бихте могли да осигурите устойчивостта използвайки CORBA. Това е нещо, което CMP би имал много грижи ако се опита да го направи!

Container Managed Persistence

Container Managed Persistence (CMP) е когато EJB Container поддържа обработката на Entity Beans във ваша полза. CMP намалява времето за проектиране на Entity Beans тъй като разработчикът може да се фокусира totally върху логиката на компонента вместо върху това как се запазва и извлича.

Bean Managed vs. Container Managed Persistence

Има няколко интересни момента които трябва да се имат предвид като се избира механизма за устойчивост на вашето EJB приложение. Ако изберете да използвате CMP разчитате на

инструмента да осигури устойчивост заради вас. Макар и да изглежда логично да се прави за прости бийнове, това може да няма способността да прави сложни устойчивости както не-jdbc устойчивост. Също липсата на утвърдили се стандарти за Entity Bean да се проектира в релационни бази данни би могла да означава че вашият Bean е принуден да работи в някакъв Container или да използва някакъв инструмент за проектиране. Долната таблица кратко сравнява и дава предимствата и недостатъците на всеки от двата механизма

Фигура x. - Сравнение на механизмите за устойчивост.

Показател	Bean-управлявана устойчивост	Container-управлявана устойчивост	Notes
Обем на кода за писане	BMP изисква ръчно кодиране на 4-те методи за устойчивостта, което значи много кодиране на викания към JDBC или друг код за устойчивост	EJB Container прави всичко това в услуга на потребителя и обикновено създава множество таблици за релационни бази за вашите същностни бинове.	
Изпълнение	Разработчикът има фин контрол върху устойчивостта, например да осъвременява само отделни понлета или да изпълнява запомнена процедура. Разработчикът лесно може да оптимизира виканията за достъп до базата данни	Някои CMP инструменти са много развити и могат да правят оптимизации и кеширане. Някои CMP инструменти са примитивни и могат само	
Поддръжка за различни места за запазване	BMP ще позволи на потребителя да отиде към SQL/J, когато има драйвер, заместо да чака CMP доставчикът да реализира машина използвайки SQL/J. Също BMP може да се приложи и чрез използване на CORBA		
Преместваемост/независимост от доставчика			

Неща за запомняне -> Ето пример идеален в аспекта на проектиране на EJB. Понеже устойчивостта е изолирана от отдалечения интерфейс, може да мените рт на вашия бийн без проблеми за никое приложение. Това значи че за ранното разработване на прототипа и проектирането на EJB може да се използва CMP а после да се прехвърли към bmp и оптимизира JDBC кода с DBA или специалист да подобри изпълнението.

Описатели на разгръщането

Компонентният модел на EJB позволява поставянето на параметри по време на разгръщането, чрез използване на описател на разгръщането. Описателите на разгръщането позволяват промени на параметрите на окръжението, атрибутите на транзакциите и ролите в сигурността да бъдат определени лед като е разработен един Bean. Това позволява на разработчика на Enterprise JavaBean да се съсредоточи само на основната логика в работата си по бийна.

Такъв дескриптор се използва за описание на Enterprise JavaBean-ове които се съдържат в EJB-jar файл. Разликата между EJB-jar файл и нормален jar файл е описателят на разгръщането. Дескрипторът на разгръщането трябва да бъде сложен в META-INF директорията на jar файла, заедно с манифестния файл. Също той трябва да има име EJB-jar.xml

Преди EJB 1.1 спецификацията описателят на разгръщането беше сериализиран обект определен във вашето код, компилиран и вкаран в jar файл. Това го правеше много труден за променяне, понеже изискваше десериализация на обектите, товаренето им в паметта, промяна на атрибутите, прекомпиляция и нова десериализация. Това беше досаден процес, ако разработчикът искаше да промени само един атрибут.

Спецификацията EJB 1.1 промени дескриптора и премахна сериализираните обекти в полза на XML. Минаването на XML има много предимства. Описателят на разгръщането може да бъде редактиран и четен от нормален текстов редактор или инструмент за XML и неизисква прекомпиляция и десериализация и сериализация на обекти.

Повечето инструменти за EJB ще съдържат ютилити за преобразуване от стария формат в новия XML 1.1 формат.

JNDI

Java Naming and Directory Interface (JNDI) е използвана в Enterprise JavaBeans като услуга за имената. Използва се за определяне местоположението на Enterprise JavaBeans и Home обекти в мрежата. JNDI прави преглеждането на обекти по-лесно отколкото оглеждането за име в телефонен указател. JNDI се проектира много близко до други стандарти като LDAP и CORBA CosNaming. За повече подробности за JNDI виж....

JTA/JTS

Java Transaction API / Java Transaction Service. EJB Container-ът използва JTA/JTS като API за транзакциите. Разработчик на Bean може да използва JTS за да получи достъп до транзакция, но по-често транзакциите се дефинират по време на разгръщането и разработчикът на Bean може да кодира неговите Enterprise JavaBean-ове без да трябва да е определил границите на транзакцията. EJB Container е отговорен да поддържа транзакциите били те локални или разпределени. Спецификацията JTS е проекцията на Java към CORBA OTS (Object Transaction Service). За повече подробности виж....

CORBA

CORBA все-повече се налага като стандарт в програмирането на системи за предприятието поради нейната независимост от доставчика, платформата и езиците. Има много прилики между услугите давани от CORBA, познати като Common Object Services (COS) и услугите давани от Enterprise JavaBeans. Понеже Enterprise JavaBeans е спецификация от по-високо ниво – свързващият протокол за EJB е RMI, макар че RMI може да бъде приложен отгоре на

множество протоколи вкл. Java Remote Method Protocol (JRMP) и Internet Interoperable ORB Protocol (IIOP) на CORBA. Поради качеството на CORBA често може да се види EJB Container доставчик да разработва EJB Container върху CORBA. Съвместимостта с CORBA спецификацията и IIOP протокола е определена в EJB спецификацията и RMI/IIOP спецификацията. Спецификацията 1.1 казва “*The Enterprise JavaBeans architecture will be compatible with the CORBA protocols.*”

Реализацията на EJB Container върху CORBA също позволява достъп до наследени системи и приложения написани на различни езици и отново спасява от зависимост от протоколи и доставчици.

Ролите на Enterprise Java Beans

Спецификацията EJB ролята на различните разработчици в процеса на създаване на EJB приложение. Тя определя ниши в които са изолирани различните области на проектирането. Това позволява разработчици специализирани в различни ниши да работят съвместно ефективно. EJB спецификацията определя 6 роли за разработването/разгръщането и управлението на EJB приложение.

Доставчик на Enterprise Bean-овете

Enterprise Bean Provider е разработчик който спазва EJB шаблоните за проектиране на Entity и Session Beans при разработването на бийнове, които заедно решават основния проблем. Enterprise JavaBeans се пакетират в jar файлове, jar файлът също трябва да съдържа Deployment Descriptor за да опише Enterprise JavaBeans съдържани в jar файла.

Монтажник на приложениета

Application Assembler-ът има ролята да събере приложението от колекция EJB-jar файлове. EJB-jar може да изват от доставчик или да са разработени от нулата, например компания специализирана в счетоводството може да произведе, разпространява и продава бийнове за счетоводството. Обаче тези Bean-ове може да не могат да извършват несчетоводните задачи. Enterprise JavaBeans са разработени за да взаимодействат с и да ръспособяват Beans в полезно приложение. Ролята на разработчика на приложения е да събере всичките необходими Enterprise JavaBeans и да ги монтира в практично, полезно приложение.

Deployer

Ролята му е да вземе колекцията от EJB-jar файлове от Assembler-а и/или Bean доставчика и да ги разгръне в EJB Container. Деплоерът е отговорен за генерацията на всякакви специфични за доставчика субове и скелетони които могат да са нужни както и за определянето на границите на транзакциите между Enterprise JavaBean-овете и ролите по сигурността. Deployer постига това с помощта на инструменти давани от EJB Server/Container доставчик.

EJB Server/Container доставчик

Сегашната версия на EJB спецификацията не определя ясно отношенията между Container доставчика и сървър доставчика и предполага че един и същ доставчик играе и двете роли. Ролята на Server/Container доставчика е да даде EJB Container който е с поведение съгласно EJB спецификацията.

Системен администратор

Ролята на Systems Administrator-а е да надзира най-важната цел на системата. Тук още кипи. Управлението и администрацията на EJB и разпределени системи са критични и все пак са предизвикателство. Това е поради факта че “приложение” не е определено като процес

работещ на една машина. Едно разпределено приложение може да се състои от много компоненти и услуги на различни места всичките конфигурирани и работещи съвместно.

Разработване с Enterprise Java Beans

<authors note>

I noticed that you guys have some sort of community theme going. If you send me some details of what we should tailor to EJB, then I'll work off that. I'm still undecided whether to show both a BMP and CMP bean, as support for CMP is fairly fresh right now. Is 1 Entity bean example enough?

</authors note> (не се превежда, отнася се за бъдещо оформление - б.пр.)

Разработка на прост Stateless Session Bean

Bean Provider - разработва бийна, домашния и отдалечения интерфейси

Bean Provider - създава и добавя входни точки в описателя на разгръщането

Bean Provider - създава EJB-jar файл

Deployer – генерира стубовете на доставчика

Deployer – създава монтажни и за разгръщането входни точки в EJB-jar файла

Deployer – добавя бийна в контейнер

Systems Admin – пуска го!

Разработка на прост Stateful Session Bean

Разработка на BMP Entity Bean

Разработка на CMP Entity Bean

Дискусионни въпроси, интересни проблеми и литература

Защо толкова много правила?

Понеже EJB контейнерът е отговорен за много и сложни задачи, изискват се Enterprise Java Bean компоненти които са проектирани така че да отговарят на стандартно множество правила. Чрез налагане на много правила също се постига независимост от доставчика и между средите за разработка, средите за разгръщане и EJB Servers/Containers.

Какво не е дефинирано в EJB спецификацията

Събития – текущо няма специфициран модел на събития в EJB. Това ще стане в EJB 2.0

Singleton шаблон – Entity и Session Beans се генерират текущо за приложения за бази данни и e-commerce и затова не предлагат възможности за всичките възможни сценарии. Пример за това е Singleton Pattern. Singleton Pattern се използва когато е необходим един единствен екземпляр от даден обект. Понеже EJB Container скрива екземплярите от потребителя, няма гаранция че последният ще работи винаги с един и същ екземпляр. Ето пример където CORBA или RMI биха могли да бъдат интегрирани в EJB приложение понеже позволяват създаване на Singleton обекти.

Други позовавания

Enterprise Java Beans Home Page - <http://java.sun.com/products/ejb/>

спецификация, разтоварвания

Enterprise Java Beans Special Interest Group - <http://www.mgm-edv.de/ejbsig/ejbsig.html>

Книги за по-нататъшно четене

Andreas Vogel, Tom Valesky, Richard Monson-Hafel

ДОСТЪП ДО ЦЯЛОСТНИ УСЛУГИ

Преглед на последните теми в главата и техните цели.

Java Naming and Directory Interface (JNDI)

Java Messaging Service (JMS)

JavaMail

Сигурност във фирмата

Резюме

Текст на резюмето

Препоръчани книги

Упражнения

8. Упражнение

17: Шаблони за проектиране

Тази глава запознава с важния и все още нетрадиционен “шаблонен” подход към проектирането на програми.

Може би най-важната стъпка напред в проектирането на ОО програми е “шаблони в проектирането”, хронилана в *Design Patterns*, от Gamma, Helm, Johnson & Vlissides (Addison-Wesley 1995).¹ Книгата показва 23 различни решения на конкрете клас проблеми. В тази глава ще се изложи основната концепция на подхода заедно с някои примери. Това трябва да изостри вашия апетит към читането на *Design Patterns* (източник който сега става основен, почти задължителен, речник за ООР програмисти).

Втората част на тази глава съдържа пример за процеса на еволюция на дизайн, започвайки с начално решение и преминавайки през логиката и процеса на еволоване на решението към по-успешен дизайн. Посочената програма (симулация на “боклуджийско кошче”) е еволовала с времето и вие можете да погледнете на тази еволюция като на прототип аналогично на който ваш собствен дизайн може да започне като адекватно решение на конкретен проблем и да еволовира в гъвкав подход към клас от проблеми.

Концепцията за шаблон

Първоначално може да мислите за шаблона като за някакъв особено сръчен и знатен начин за решаване на определен ъръг проблеми. Тоест сякаш много хора са разработили заедно всичките гяволии на проблема и са излезли с най-общото, гъвкаво решение за него. Проблемът би могъл да бъде виждан и решен от вас преди, но вашето решение вероятно няма да има завършеността, която ще видите в шаблона.

Макар че са наречени “шаблони за проектиране,” те не са само за сферата на проектирането. Шаблонът изглежда да стои на страна от традиционния начин на мислене при анализа, проектирането и реализацията. Вместо тях шаблонът вгражда цялостна идея в една програма, а така може да се появи във фазата на анализ или в проектирането на високо ниво. Това е интересно понеже шаблонът има пряка реализация в код и така вие можете да не очаквате да се покаже преди проектирането на ниско нива или реализацията (и фактически вие бихте могли да не разбирате че ви трябва конкретен шаблон преди да минтете през тези фази).

Основната концепция за шаблон би могла също да бъде видяна като основна концепция на проектирането на програми: добавяне на слой абстракция. Винаги когато абстрагирате нещо вие изолирате конкретни детайли, а един от най-важните мотиви за това е да се разделят *нещата* които се менят от *нещата*, които остават *същите*. Друг начин да се каже това е че когато веднъж намерите някаква част от вашата програма вероятно ще се променя по една или друга причина, ще поискате да предотвратите щото тези промени да разпространят (предизвикат - б.пр.) други промени във вашия код. Това не само прави кода много по-лесен за поддържане, но излиза че в повечето случаи е по-лесно за разбиране (което резултира в по-ниски разходи).

¹ But be warned: the examples are in C++.

Често най-трудната част от разработването на елегантен и лесен за поддръжка дизайн е откриването на това, което аз наричам “вектора на промяната.” (Тук “вектор” се отнася за максимален градиент, а не за клас-колекция.) Това значи намиране на най-важното нещо което се променя във вашата система, или с други думи, намиране къде ви е най-голямата цена. Веднъж като откриете вектора на промяната, вече имате фокусна точка около която да структурирате вашия дизайн.

Така целта на шаблоните в проектирането е да изолират промените във вашия код. Ако гледате на нещата по този начин, вече сте видели някои шаблони в тази книга. Например наследяването може да се нарече шаблон за проектиране (но прилаган от компилатора). То ви позволява да изразите разликата в поведението (това е нещото което се мени) в обекти които имат всичките един и същ интерфейс (това е което остава същото). Композицията също може да бъде смятана за шаблон, понеже ви позволява да мените – динамично или статично – обектите които прилага вашия клас, а с това начина по който работи класът.

Вече сте виждали и друг шаблон който се появява в *Design Patterns: iterator* (Java 1.0 и 1.1 кипризно го нарича **Enumeration**; Java 2 колекциите използват “iterator”). Това скрива конкретна реализация на колекцията като минавате по нея и гледате елементите един по един. Итераторът ви позволява да пишете родов код който провежда операция над всичките елементи последователно без значение как последователността е била построена. Така вашият родов код може да се използва с всякакви колекции които могат да дадат итератор.

Синглетонът

Може би най-простият шаблон за проектиране е *singleton*-ът, който е начин да се даде един и само един екземпляр от обект. Това е използвано в библиотеките на Java, но има по-директен пример:

```
//: c17.SingletonPattern.java
// The Singleton design pattern: you can
// never instantiate more than one.
package c17;

// Since this isn't inherited from a Cloneable
// base class and cloneability isn't added,
// making it final prevents cloneability from
// being added in any derived classes:
final class Singleton {
    private static Singleton s = new Singleton(47);
    private int i;
    private Singleton(int x) { i = x; }
    public static Singleton getHandle() {
        return s;
    }
    public int getValue() { return i; }
    public void setValue(int x) { i = x; }
}

public class SingletonPattern {
    public static void main(String[] args) {
        Singleton s = Singleton.getHandle();
        System.out.println(s.getValue());
        Singleton s2 = Singleton.getHandle();
        s2.setValue(9);
        System.out.println(s.getValue());
        try {
            // Can't do this: compile-time error.
```

```

    // Singleton s3 = (Singleton)s2.clone();
} catch(Exception e) {}
}
} //:~

```

Ключът към създаването на синглетона е да се предотврати създаването на обект по какъвто и да е начин от потребителя освен дадения от вас. Трябва да направите всичките конструктори **private**, трябва да създадете най-малко един конструктор за да предотвратите създаването на конструктор по подразбиране от компилатора (който ще се създае като "приятелски").

В тази точка вече трябва да решите как ще създавате вашия обект. Тук кой е създаден статично, но също може да чакате клиентския програмист да поиска създаването му и да го създадете по поръчка. Във всички случаи обектът трябва да бъде запазен частно. Достъп се дава чрез публични методи. Тук **getHandle()** дава манипулятор към **Singleton** обекта. Останалата част от интерфейса (**getValue()** и **setValue()**) е обикновен интерфейс на клас.

Java също позволява създаването на обект чрез клониране. В този пример правенето на класа **final** премахва възможността за клониране. Понеже **Singleton** е наследен направо от **Object**, методът **clone()** остава **protected** така че не може да бъде използван (при опит се получава грешка по време на компилиация). Обаче ако наследявате от йерархия която има вече подтиснат **clone()** като **public** и прилагащ **Cloneable**, начинът да се предотврати клонирането е да се подтисне **clone()** и да се изхвърли **CloneNotSupportedException** както е описано в глава 12. (Също бихте могли да подтиснете **clone()** и просто да върнете **this**, но това ще бъде измамно, понеже клиент-програмистът би помислил че клонирането е станало, докато същевременно би имал работа с оригинала.)

Забележете че не сте ограничени да създадете само един обект. Това също е техника за създаване на ограничен брой обекти. В подобна ситуация, обаче, ще се изправите срещу въпроса за съвместното използване на обектите. Ако това е важно, ще трябва да създадете механизъм за следене на влизането и излизането от обектите.

Класификация на шаблоните

Книгата *Design Patterns* дискутира 23 различни шаблона, класифицирайки ги по три цели (всички цели се въртят около конкретен аспект който може да варира). The three purposes are:

- Създаване:** как може да бъде създаван обект. Това често включва изолирането на детайлите по създаването на обектите така че вашият код не зависи от това какви типове обекти има и затова няма нужда да бъде променян когато въвеждате нов тип обект. Поменатият преди *Singleton* е класифициран като шаблон за създаване, а по-късно в тази глава ще видите примери на *Factory Method* и *Prototype*.
- Структурни:** проектиране на обектите да удовлетворяват конкретни ограничения. Тук се работи с начините по които обектите се свързват с други обекти за да се осигури промените в обектите да не довеждат до промени на останалата част от системата.
- Поведенчески:** Обекти, които обслужват определени типове действия в програма. Тук се капсулират процесите, които искате да проведете, такива като интерпретация на език, изпълняване на поръчка, пробягване на последователност (като в итератор), или прилагане на алгоритъм. Тази глава съдържа примери на *Observer* и *Visitor* шаблони.

Книгата *Design Patterns* има секция за всеки от нейните 23 шаблона заедно с един или повече примери, типично на C++ но понякога на Smalltalk. (Ще намерите, че това няма особено значение защото лесно ще транслирате концепцията от другите езици на Java.) Тази книга няма да повтаря шаблоните от *Design Patterns* понеже книгата стои сама по себеси и трябва да се изучава отделно. Вместо това тази глава ще даде някои примери които ще създават прилично усещане за какво са шаблоните и защо те са толкова важни.

Фабрики: капсулиране създаването на обекти

Когато откриете, че е необходимо да добавите нови типове към системата, най-смислената първа стъпка е да използвате полиморфизъм за създаването на общ интерфейс за тези типове. Това отделя останалата част от кода във вашата система от знанието за конкретни типове които добавяте. Нови типове могат да се създават без да се засяга съществуващ код ... или така изглежда. Отначало ще ни се струва че единственото място което трябва да промените в кода на такъв дизайн е мястото където наследявате нов тип, но това не е точно така. Още трябва да се създаде обект от новия тип, а в точката на създаването трябва да зададете точно кой конструктор да се използва. Така ако кодът който създава обектите е разпределен по вашето приложение, имате същия проблем когато добавяте нови типове – трябва да изловите всичките места, където променяте неща. Излиза, че това което е същественото в случая е създаването на типа а не използването на типа (за което се е погрижил полиморфизъм), но ефектът е същият: добавянето на нов тип може да причини проблеми.

Решението е да бъде наложено всякакви създавания на обекти да минават през обща фабрика наместо да се позволи кодът за създаването да е разхвърлян из цялото приложение. Ако всичкият код във вашата програма трябва да минава през тази фабрика щом трябва да се създаде обект, тогава всичко което трябва да се направи е да се добави обектът към фабриката, тя да се промени.

Понеже всяка ОО програма създава обекти, и понеже е твърде вероятно да поискате да разширите ваши програми с нови типове, аз подозирам че фабриките ще са най-използваните от шаблоните за проектиране.

Като пример нека да ревизираме **Shape** системата. Един подход е да направим фабриката **static** метод на базовия клас:

```
//: c17:ShapeFactory1.java
// A simple static factory method
import java.util.*;

class BadShapeCreation extends Exception {
    BadShapeCreation(String msg) {
        super(msg);
    }
}

abstract class Shape {
    public abstract void draw();
    public abstract void erase();
    static Shape factory(String type)
        throws BadShapeCreation {
        if(type == "Circle") return new Circle();
        if(type == "Square") return new Square();
        throw new BadShapeCreation(type);
    }
}

class Circle extends Shape {
    Circle() {} // Friendly constructor
    public void draw() {
        System.out.println("Circle.draw");
    }
}
```

```

}
public void erase() {
    System.out.println("Circle.erase");
}
}

class Square extends Shape {
    Square() {} // Friendly constructor
    public void draw() {
        System.out.println("Square.draw");
    }
    public void erase() {
        System.out.println("Square.erase");
    }
}

public class ShapeFactory1 {
    public static void main(String args[]) {
        String shlist[] = { "Circle", "Square",
            "Square", "Circle", "Circle", "Square" };
        ArrayList shapes = new ArrayList();
        try {
            for(int i = 0; i < shlist.length; i++)
                shapes.add(Shape.factory(shlist[i]));
        } catch(BadShapeCreation e) {
            e.printStackTrace();
            return;
        }
        Iterator i = shapes.iterator();
        while(i.hasNext()) {
            Shape s = (Shape)i.next();
            s.draw();
            s.erase();
        }
    }
} ///:~

```

factory() взема аргумент който позволява да се определи томния вид **Shape** който да създаде; в този случай това е **String** но би могъл да бъде всякакво множество данни. **factory()** е единствения друг код в системата който трябва да бъде променен когато се добавя нов тип **Shape** (долните нужни за инициализацията предполагаемо ще дойдат някъде отвън за системата и няма да бъдат твърдо кодирани в масив както в горния пример).

За да се окуражи създаването само във **factory()**, конструкторите за специфичните видове **Shape** са направени “friendly,” така че **factory()** има достъп до конструкторите но те не са достъпни извън пакета.

ПОЛИМОРФНИ ФАБРИКИ

Методът **static factory()** в предишния пример принуждаова цялото създаване да бъде фокусирано в една точка, която да е единственото място където ще се изисква промяна на кода. Това сигурно е достатъчно решение, като слага кутия около създаването на обекти. Обаче книгата *Design Patterns* подчертава че основанието за шаблона *Factory Method* е това че различни видове фабрики могат да бъдат подкласове от основната фабрика (горният дизайн е споменат като специален случай). Обаче книгата не дава пример, а наместо това просто повтаря примера използван за *Abstract Factory* (ще видите такъв в следващата секция). Ето **ShapeFactory1.cpp** променен така че методите на фабрикат аса в отделен клас като

виртуални функции. Забележете също че специфични **Shape** се зареждат динамично при нужда:

```
//: c17:ShapeFactory2.java
// Polymorphic factory methods
import java.util.*;

class BadShapeCreation extends Exception {
    BadShapeCreation(String msg) {
        super(msg);
    }
}

interface Shape {
    void draw();
    void erase();
}

abstract class ShapeFactory {
    protected abstract Shape create();
    static Map factories = new HashMap();
    static Shape createShape(String id)
        throws BadShapeCreation {
        if(!factories.containsKey(id)) {
            try {
                Class.forName(id); // Load dynamically
            } catch(ClassNotFoundException e) {
                throw new BadShapeCreation(id);
            }
            // See if it was put in:
            if(!factories.containsKey(id))
                throw new BadShapeCreation(id);
        }
        return
            ((ShapeFactory)factories.get(id)).create();
    }
}

class Circle implements Shape {
    private Circle() {}
    public void draw() {
        System.out.println("Circle.draw");
    }
    public void erase() {
        System.out.println("Circle.erase");
    }
    static class Factory extends ShapeFactory {
        protected Shape create() {
            return new Circle();
        }
    }
    static {
        ShapeFactory.factories.put(
            "Circle", new Circle.Factory());
    }
}
```

```

class Square implements Shape {
    private Square() {}
    public void draw() {
        System.out.println("Square.draw");
    }
    public void erase() {
        System.out.println("Square.erase");
    }
    static class Factory extends ShapeFactory {
        protected Shape create() {
            return new Square();
        }
    }
    static {
        ShapeFactory.factories.put(
            "Square", new Square.Factory());
    }
}

public class ShapeFactory2 {
    public static void main(String args) {
        String shlist[] = { "Circle", "Square",
            "Square", "Circle", "Circle", "Square" };
        ArrayList shapes = new ArrayList();
        try {
            for(int i = 0; i < shlist.length; i++)
                shapes.add(
                    ShapeFactory.createShape(shlist(i)));
        } catch(BadShapeCreation e) {
            e.printStackTrace();
            return;
        }
        Iterator i = shapes.iterator();
        while(i.hasNext()) {
            Shape s = (Shape)i.next();
            s.draw();
            s.erase();
        }
    }
}
} //:~
```

Сега "фабричният" метод се появява в негов собствен клас, **ShapeFactory**, във вид на метода **create()**. Това е **protected** метод което значи че не може да бъде викан директно, но може да бъде подписан. Подкласовете на **Shape** трябва всеки да създава собствен подklass на **ShapeFactory** и да подписка метода **create()** за да създава обект от собствения си тип. Фактическото създаване на фигури става чрез викане на **ShapeFactory.createShape()**, който е статичен метод използващ **Map** в **ShapeFactory** за намиране на подходящ фабричен обект като използва подадения от вас идентификатор. Фабриката веднага се използва за създаване на обект-фигура, но бихте могли да си въобразите по-сложна ситуация където подходящия фабричен обект се връща и после използва от викащия код за създаване на обект по по-усложнен начин. Обаче изглежда че повечето пъти няма да ви трябват хитрините на полиморфния фабричен метод и единствен статичен метод в базовия клас (както е показано в **ShapeFactory1.cpp**) ще работи чудесно.

Забележете че **ShapeFactory** трябва да бъде инициализиран чрез натоварване на нейния **Map** с обекти-фабрики, което става в статична инициализационна клауза във всяка от реализациите на **Shape**. Така че за да се добави нов туп в този дизайн е необходимо да се

наследи типа, да се създаде фабрика и да се добави статичната инициализационна кляуза за товарене на **Map**. Тази допълнителна сложност не окуражава използването на **static** фабричен метод ако не е нужно да създавате индивидуални фабрични обекти.

Абстрактни фабрики

Шаблонът *Abstract Factory* прилича на видяните преди фабрични обекти, но не е с един а с няколко методи-фабрики. Всеки от тях създава различен вид обекти. Идеята е че в момента на създаване на обекта-фабрика решавате как ще се използват всичките създавани от нея обекти. Примерът даден в *Design Patterns* реализира преместваемост през различни графични интерфейси (GUIта): създавате обект-фабрика подходящ за GUI с който работите, а после питате за меню, бутон и т.н. и се създава автоматично подходяща за този GUI версия. Така сте способни да изолирате, на едно място, ефекта от преминаване от един GUI към друг.

As another example suppose you are creating a general-purpose gaming environment and you want to be able to support different types of games. Here's how it might look using an abstract factory:

```
//: c17:GameEnvironment.java
// An example of the Abstract Factory pattern

interface Obstacle {
    void action();
}

interface Player {
    void interactWith(Obstacle o);
}

class Kitty implements Player {
    public void interactWith(Obstacle ob) {
        System.out.print("Kitty has encountered a ");
        ob.action();
    }
}

class KungFuGuy implements Player {
    public void interactWith(Obstacle ob) {
        System.out.print("KungFuGuy now battles a ");
        ob.action();
    }
}

class Puzzle implements Obstacle {
    public void action() {
        System.out.println("Puzzle");
    }
}

class NastyWeapon implements Obstacle {
    public void action() {
        System.out.println("NastyWeapon");
    }
}

// The Abstract Factory:
```

```

interface GameElementFactory {
    Player makePlayer();
    Obstacle makeObstacle();
}

// Concrete factories:
class KittiesAndPuzzles
implements GameElementFactory {
    public Player makePlayer() {
        return new Kitty();
    }
    public Obstacle makeObstacle() {
        return new Puzzle();
    }
}

class KillAndDismember
implements GameElementFactory {
    public Player makePlayer() {
        return new KungFuGuy();
    }
    public Obstacle makeObstacle() {
        return new NastyWeapon();
    }
}

public class GameEnvironment {
    private GameElementFactory gef;
    private Player p;
    private Obstacle ob;
    public GameEnvironment(
        GameElementFactory factory) {
        gef = factory;
        p = factory.makePlayer();
        ob = factory.makeObstacle();
    }
    public void play() {
        p.interactWith(ob);
    }
    public static void main(String args) {
        GameElementFactory
            kp = new KittiesAndPuzzles(),
            kd = new KillAndDismember();
        GameEnvironment
            g1 = new GameEnvironment(kp),
            g2 = new GameEnvironment(kd);
        g1.play();
        g2.play();
    }
} //:~

```

В това обкъръжение обектите **Player** взаимодействат с **Obstacle** обекти, но има различни видове играчи и препятствия в зависимост от играта която се играе. Определяте вида на играта чрез избиране на конкретен **GameElementFactory**, а после **GameEnvironment** управлява постановката и игрането на самата игра. В този пример те са много прости, но като активности (*initial conditions* и *state change*) могат да определят голяма част от това,

което играта изглежда и дава. Тук **GameEnvironment** не е проектиран за да бъде наследяван, макар и вероятно да би ичало много смисъл да се направи това.

Това също съдържа примери за *Double Dispatching* и *Factory Method*, и двата ще бъдат обяснени по-късно.

Шаблонът "наблюдател"

Шаблонът наблюдател решава доста познат проблем: Какво да се прави ако група обекти трябва да се осъвременят когато даден обект си промени състоянието? Това може да бъде видяно в "model-view" аспекта на MVC (model-view-controller) на Smalltalk, или в почти еквивалентната "Document-View Architecture." Да предположим че има някакви данни ("документ") и повече от един изглед, да кажем чертеж и текстов изход. Когато промените данните, двата изгледа трябва да се осъвречат, а именно това е което подпомага наблюдателя. Това е достатъчно общ и срещан проблем така че решението му бе включено в библиотеката **java.util**.

Има два типа обекти използвани за реализация на наблюдателския шаблон в Java. Класът **Observable** пази сведения за всеки който иска да бъде информиран когато нещата се променят, дали "състоянието" се е променило или не. Когато някой казва "OK, всеки ще проверява и осъвременява себе си," класът **Observable** изпълнява тази задача чрез метода **notifyObservers()** за всеки от списъка. Методът **notifyObservers()** е част от базовия клас **Observable**.

Има фактически две "неща които се променят" в наблюдателския шаблон: количеството на наблюдаваните обекти и начинът на осъвременяването. Тоест, този шаблон ви дава начин да мените тези две неща без да засягате окръжаващия код.

Следващия пример прилика на примера **ColorBoxes** от глава 14. Кутийки се слагат на мрежа на екрана и всяка се инициализира със случаен цвят. Освен това всяка кутийка **implements** интерфейса **Observer** и е регистрирана в **Observable** обект. Когато кликнете на кутийка, всички останали кутийки се уведомяват че има промяна понеже обектът **Observable** автоматично вика метода **update()** на всеки **Observer** обект. В този метод кутийката проверява да види дали не е нагласена според това което се кликва и ако е така си слага цвета според кликнатия цвят.

```
//: c17:BoxObserver.java
// Demonstration of Observer pattern using
// Java's built-in observer classes.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

// You must inherit a new type of Observable:
class BoxObservable extends Observable {
    public void notifyObservers(Object b) {
        // Otherwise it won't propagate changes:
        setChanged();
        super.notifyObservers(b);
    }
}

public class BoxObserver extends JFrame {
    Observable notifier = new BoxObservable();
    public BoxObserver(int grid) {
```

```

setTitle("Demonstrates Observer pattern");
Container cp = getContentPane();
cp.setLayout(new GridLayout(grid, grid));
for(int x = 0; x < grid; x++)
    for(int y = 0; y < grid; y++)
        cp.add(new OCBox(x, y, notifier));
}

public static void main(String[] args) {
    int grid = 8;
    if(args.length > 0)
        grid = Integer.parseInt(args[0]);
    JFrame f = new BoxObserver(grid);
    f.setSize(500, 400);
    f.setVisible(true);
    f.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
}
}

class OCBox extends JPanel implements Observer {
    Observable notifier;
    int x, y; // Locations in grid
    Color cColor = newColor();
    static final Color[] colors = {
        Color.black, Color.blue, Color.cyan,
        Color.darkGray, Color.gray, Color.green,
        Color.lightGray, Color.magenta,
        Color.orange, Color.pink, Color.red,
        Color.white, Color.yellow
    };
    static final Color newColor() {
        return colors(
            (int)(Math.random() * colors.length)
        );
    }
    OCBox(int x, int y, Observable notifier) {
        this.x = x;
        this.y = y;
        notifier.addObserver(this);
        this.notifier = notifier;
        addMouseListener(new ML());
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(cColor);
        Dimension s = getSize();
        g.fillRect(0, 0, s.width, s.height);
    }
}

class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        notifier.notifyObservers(OCBox.this);
    }
}

```

```

public void update(Observable o, Object arg) {
    OCBox clicked = (OCBox)arg;
    if(nextTo(clicked)) {
        cColor = clicked.cColor;
        repaint();
    }
}
private final boolean nextTo(OCBox b) {
    return Math.abs(x - b.x) <= 1 &&
        Math.abs(y - b.y) <= 1;
}
} ///:~

```

Когато за пръв път разглеждате онлайн документацията на **Observable**, тя е малко смущаваща понеже изглежда че може да използвате единичен **Observable** обект за управление на осъвременяването. Но това не работи; опитайте го – вътре в **BoxObserver** създайте **Observable** обект вместо **BoxObservable** обект и вижте какво става: нищо. За да получите ефекта трябва да наследите от **Observable** и някъде в кода на наследения обект да извикате **setChanged()**. Това е метода който слага флага за “промяна”, което значи че когато извикате **notifyObservers()** всичките наблюдатели ще бъдат, наистина, уведомени. В примера по-горе **setChanged()** просто се вика в **notifyObservers()**, но бихте могли да използвате всякакъв критерий за да определите къде да се вика **setChanged()**.

BoxObserver съдържа единствен **Observable** обект наречен **notifier** и всеки път когато се създава **OCBox** обект той се връзва с **notifier**. В **OCBox** винаги когато кликнете мишката методът **notifyObservers()** се вика, давайки кликнатия обект като аргумент така че всички обекти които приемат съобщението (в техния метод **update()**) знаят кой е бил кликнат и дали да се променят или не. Използвайки комбинация на кода от **notifyObservers()** и **update()** бихте могли да се справите с някои много сложни схеми.

Може да изглежда че начинът по който се уведомяват обектите остава замразен по време на компилация в метода **notifyObservers()**. Обаче ако погледнете по-отблизо кода по-горе ще видите че единственото място в **BoxObserver** или **OCBox** където знаете че работите с **BoxObservable** е точката на създаване на **Observable** обекта – от там нататък всичко използва **Observable** интерфейса. Това значи че бихте могли да наследите други **Observable** класове и да ги сменяте по време на изпълнение ако искате да промените маниера на сигнализация.

Симулиране на рециклиатор

Естеството на този проблем е че боклукът се хвърля некласифициран в единствено място, така че информацията за конкретния тип се губи. Но по-късно информацията за конкретния тип трябва да бъде възстановена за да може да се сортира рециклиаторът. В началното решение се използва RTTI (описан в глава 11).

Това не е тривиално понеже има добавено ограничение. Това го прави и интересно – повече прилича на обърканите проблеми които срещате в ежедневната си работа. Допълнителното ограничение е че боклукът отива в мястото за обработка размесен. Програмата трябва да моделира сортирането на боклука. Тук влиза в действие RTTI: имате куп анонимни порчета боклук, а програмата определя точно какъв тип е всяко от тях.

```

//: c17:recyclea:RecycleA.java
// Recycling with RTTI
package c17.recyclea;
import java.util.*;
import java.io.*;

```

```

abstract class Trash {
    private double weight;
    Trash(double wt) { weight = wt; }
    abstract double value();
    double weight() { return weight; }
    // Sums the value of Trash in a bin:
    static void sumValue(ArrayList bin) {
        Iterator e = bin.iterator();
        double val = 0.0f;
        while(e.hasNext()) {
            // One kind of RTTI:
            // A dynamically-checked cast
            Trash t = (Trash)e.next();
            // Polymorphism in action:
            val += t.weight() * t.value();
            System.out.println(
                "weight of " +
                // Using RTTI to get type
                // information about the class:
                t.getClass().getName() +
                " = " + t.weight());
        }
        System.out.println("Total value = " + val);
    }
}

class Aluminum extends Trash {
    static double val = 1.67f;
    Aluminum(double wt) { super(wt); }
    double value() { return val; }
    static void value(double newval) {
        val = newval;
    }
}

class Paper extends Trash {
    static double val = 0.10f;
    Paper(double wt) { super(wt); }
    double value() { return val; }
    static void value(double newval) {
        val = newval;
    }
}

class Glass extends Trash {
    static double val = 0.23f;
    Glass(double wt) { super(wt); }
    double value() { return val; }
    static void value(double newval) {
        val = newval;
    }
}

public class RecycleA {
    public static void main(String[] args) {
        ArrayList bin = new ArrayList();
        // Fill up the Trash bin:

```

```

for(int i = 0; i < 30; i++) {
    switch((int)(Math.random() * 3)) {
        case 0 :
            bin.add(new
                Aluminum(Math.random() * 100));
            break;
        case 1 :
            bin.add(new
                Paper(Math.random() * 100));
            break;
        case 2 :
            bin.add(new
                Glass(Math.random() * 100));
    }
}
ArrayList
glassBin = new ArrayList(),
paperBin = new ArrayList(),
alBin = new ArrayList();
Iterator sorter = bin.iterator();
// Sort the Trash:
while(sorter.hasNext()) {
    Object t = sorter.next();
    // RTTI to show class membership:
    if(t instanceof Aluminum)
        alBin.add(t);
    if(t instanceof Paper)
        paperBin.add(t);
    if(t instanceof Glass)
        glassBin.add(t);
}
Trash.sumValue(alBin);
Trash.sumValue(paperBin);
Trash.sumValue(glassBin);
Trash.sumValue(bin);
}
} ///:~

```

Първото нещо което ще забележите е оператора **package**:

```
| package c16.recyclea;
```

Това значи че в листингите на сорса налични за книгата този файл ще бъде сложен в поддиректорията **recyclea** която е поддиректория на **c16** (за глава 16). Разпакетирация инструмент от глава 17 се грижи за разполагането в правилната поддиректория. Причината да се прави това е че в тази глава се преписва този конкретен пример няколко пъти и слагането в **package** предотвратява колизии на имената на класовете.

Няколко обекта **ArrayList** са създадени за да съдържат **Trash** манипулаторите. Разбира се, **ArrayLists** фактически съдържа **Object**и така че ще съдържа каквото и да е (понеже то произхожда от **Object** - б.пр.). Причината да съдържат **Trash** (или нещо наследено от **Trash**) е само защото сте били грижливи да не слагате вътре нищо освен **Trash**. Ако сложите нещо "нередно" в **ArrayLista**, няма да получите предупреждения или грешки при компилация – ще го откриете само при изключението по време на изпълнение.

Когато **Trash** манипулаторите са добавени, те губят своята специфична идентичност и стават просто **Object** манипулатори (те са **ълкастнати**). Обаче поради полиморфизма правилното поведение вде още е налице когато се викат от **Iterator sorter** динамично свързани методи,

щом резултиращият **Object** е кастнат обратно към **Trash.sumValue()** също използва **Iterator** за да изпълни операции върху всеки обект в **ArrayList**.

Изглежда тъпо да се ъпкастват типовете от **Trash** в колекция съдържаща манипулатори към базовите типове, а после да се обръща и да се даункаства. Защо просто да не се сложат в подходящ резервояр на първото място? (Разбира се, това е цялата енigma на рециклирането). В тази програма би било лесно да се поправи, но понякога системата архитектура и гъвкавостта може много да спечелят от даункастинга.

Тази програма удовлетворява условията на проектирането: тя работи. Това би могло да бъде "файн" доколкото е еднократно решение. Обаче една полезна програма има тенденция да еволюира с течение на времето, така че трябва да се запитаме "Какво ще стане, ако се промени ситуацията?" Например мукавата е ценна стока за рециклиране сега, а как ще интегрираме това в системата? (особено ако програмата е голяма и сложна). Тонеже горния **switch** би могъл да бъде разпръснат из програмата във вид на няколко такива, трябва да ги намерите всичките винаги щом се добави нов тип и да промените кода, а ако пропуснете някое място компилаторът няма да ви помогне с никакви съобщения за грешка.

Кълчът към неправилното използване на RTTI тук е че се проверява всеки тип. Ако следите само за едно подмножество от типове понеже то изиска специално третиране, вероятно ще е добре. Но ако ловувате за всеки тип в оператора **switch**, значи вероятно пропускате нещо важно и определено правите вашия код по-труден за поддръжка. В следващата секция ще видим как тази програма еволюира с течение на времето в няколко фази за да стане много по-гъвкава. Това трябва да даде ценен пример за проектиране на програма.

Подобряване на дизайна

Решенията в *Design Patterns* са организирани около въпроса "Какво ще се промени при еволюцията на тази програма?" Това обикновено е най-важният въпрос който може да се зададе относно един дизайн. Ако можете да построите вашата система около отговора, резултатът ще бъде двуостър: не само че вашата система ще позволи лесно (и не скъпо) поддържане, но също бихте могли да произведете компоненти които са използвани повторно, така че други системи биха могли да бъдат произведени по-евтино. Това е обещанието на ООП, но то не се случва автоматично; то изиска мисъл и вникване от ваша страна. В тази секция ще видим как този процес може да протече по време на подобряването на системата.

Отговорът на въпроса "Какво ще се промени?" за рециклиращата система е често срещан: ще се добавят още типове към системата. Целта на дизайна, тогава, е да се направи добавянето на типове колкото може по-безболезнено. В рециклиращата програма бихме желали да капсулираме всички места където се споменава специфична информация за типа, така че (ако не по друга причина) всички промени да могат да бъдат локализирани в тези места. Излиза, че този процес също почиства значително останалата част от кода.

"Прави още обекти"

Това извежда на преден план общия принцип на ООП който чух за пръв път от Grady Booch: "Ако дизайнът е твърде сложен, прави още обекти." Това е едновременно антиинтуитивно и зашеметяващо просто, а все пак е най-полезната насока която съм чувал. (Може да забележите че "правенето на повече обекти" често е еквивалентно на "добавяне на ново ниво на опосредственост.") Изобщо, ако намерите място с объркан код, вижте какъв вид клас би го оправил. Често страничен ефект от изчистването на кода е система която е по-гъвкава и с по-добра структура.

Да видим първото място където се създават **Trash** обекти, което е операторът **switch** вътре в **main()**:

```

for(int i = 0; i < 30; i++) {
    switch((int)(Math.random() * 3)) {
        case 0 :
            bin.add(new
                Aluminum(Math.random() * 100));
            break;
        case 1 :
            bin.add(new
                Paper(Math.random() * 100));
            break;
        case 2 :
            bin.add(new
                Glass(Math.random() * 100));
    }
}

```

Това определено е объркано, а също е място, където трябва да променяте кода всеки път, когато се добавя нов тип. Ако често се добавят нови типове, по-добро решение е един метод, който получава всичката необходима информация и връща манипулятор към обект от коректния тип, вече ъпкастнат към `trash` обект. В *Design Patterns* това се споменава широко като *creational pattern* (има няколко вида). Клещицният шаблон който ще се употреби тук е вариант на *Factory Method*. Тука фабричният метод е **static** член на `Trash`, но по-често е метод който ще бъде подписан в извлечения клас.

Идеята на фабричният метод е че му давате важната информация от която той се нуждае за да създаде вашия обект, а после чакате за манипулятор (вече ъпкастнат към базовия тип) да изскочи като върната стойност. От там нататък третирате обекта полиморфично. Така даже няма нужда да знаете точния тип на създадения обект. Фактически фабричният метод го скрива от вас за да предотврати неправилно използване. Ако искате да използвате обекта без полиморфизъм, трябва явно да използвате RTTI и кастинг.

Но има малък проблем, специално когато използвате по-сложния подход (не е показван тук) с правене на фабричният метод в базовия клас и подтискането му в извлечения клас. Какво ако извлечения клас иска повече информация във вид на различни или повече аргументи? "Създаване на повече обекти" решава този проблем. За реализация на фабричният метод класът `Trash` взима нов метод наречен **factory**. За да се скрият данните по създаването, има нов клас наречен `Info` който съдържа всичката необходима информация за **factory** метода за създаване на съответния `Trash` обект. Ето проста реализация на `Info`:

```

class Info {
    int type;
    // Must change this to add another type:
    static final int MAX_NUM = 4;
    double data;
    Info(int typeNum, double dat) {
        type = typeNum % MAX_NUM;
        data = dat;
    }
}

```

Единствената задача на обекта `Info` е да държи информация за метода `factory()`. Сега ако имаме ситуация в която `factory()` иска повече или различна информация за създаване на нов `Trash` обект, интерфейсът `factory()` няма нужда да бъде промянен. Класът `Info` може да бъде променен чрез добавяне на нови данни или нови конструктори, или чрез по-типичния за ООП начин със създаване на подкласове.

Методът `factory()` за този прост пример изглежда така:

```

static Trash factory(Info i) {

```

```

switch(i.type) {
    default: // To quiet the compiler
    case 0:
        return new Aluminum(i.data);
    case 1:
        return new Paper(i.data);
    case 2:
        return new Glass(i.data);
    // Two lines here:
    case 3:
        return new Cardboard(i.data);
}
}

```

Тук определянето на точния тип на обекта е просто, но може да си въобразим по-усложнена система където **factory()** използва сложен алгоритъм. Важното е че сега той е скрит в едно място, а вие знаете да отидете на това място когато добавяте типове.

Създаването на нов тип е сега много по-просто в **main()**:

```

for(int i = 0; i < 30; i++)
    bin.add(
        Trash.factory(
            new Info(
                (int)(Math.random() * Info.MAX_NUM),
                Math.random() * 100)));

```

Създаден е **Info** обект за даване на данните във **factory()**, който на свой ред произвежда някакъв вид **Trash** обект на хийпа и връща манипулатора който е добавен към **ArrayList bin**. Разбира се, ако промените количеството и типа на аргументите, този оператор ще трябва все пак да бъде променен, но това може да се избегне ако създаването на **Info** обекта се автоматизира. Например **ArrayList** от аргументи може да бъде подаден на конструктора на **Info** обект (или направо на викането на **factory()**, ако е въпросът). Това изисква разбор и проверка на аргументите да се направи по време на изпълнение, но дава най-голяма гъвкавост.

Може да видите от кода че проблемът “вектор на промяната” е който решава фабриката: ако добавите нови типове към системата (промяната), единствения код който трябва да се промени е във фабриката, така че фабриката изолира ефекта от тази промяна.

Шаблон за създаване на прототипи

Проблем с горния дизайн е, че все пак трябва да има място където всички възможни типова трябва да са известни: в метода **factory()**. Ако редовно се добавят нови типове към системата, методът **factory()** трябва да бъде променян за всеки нов тип. Когато откриете нещо такова, полезно е да опитате да направите една стъпка напред и да придвижите **всичката** информация за типа – включително създаването му – в класа представящ този тип. По този начин единственото нещо което трябва да направите при добавянето на нов тип към системата е да наследите един клас.

За да преместите информацията засягаща типа във всеки специфичен вид **Trash**, шаблонът “прототип” (от книгата *Design Patterns*) ще се използва. Общата идея е че имате главна редица от обекти, всеки един от тип който ви интересува да правите. Обектите от тази редица са използвани само за правене на нови обекти, използвайки операция която не е много различна от схемата на **clone()** вградена в кореновия за Java клас **Object**. В този случай ще наречем клониранция метод **tClone()**. Когато сте готови да направите нов обект, предполага се че имате някакъв вид информация която определя типа на обекта който искате да създавате, тогава преминавате по главната редица от обекти сравнявайки вашата информация с тази в

прототипните обекти в редицата. Когато намерите някой, който ви удовлетворява, клонирате го.

В тази схема няма твърдо кодирана информация за създаването. Всеки обект знае как да покаже подходящата информация и как да се самоклонира. Така методът **factory()** не е необходимо да бъде променян когато се добавя нов тип към системата.

Един подход към проблема за прототипирането е да се добавят множество методи за поддръжка на създаването на нови обекти. Обаче в Java 1.1 вече има поддръжка за създаването на нови обекти ако имате манипулатор към **Class** обект. С рефлексията в Java 1.1 (въведена в глава 11) може да викате конструктор даже ако имате само манипулатор към **Class** обект. Това е перфектното решение на проблема с прототипите.

Списъкът на прототипите ще бъде представен индиректно чрез списък на манипулатори към всичките **Class** обекти които искате да създадете. Свен това ако прототипирането се провали, методът **factory()** ще предполага че това е защото конкретен **Class** обект не е бил в списъка, та ще се опита да го натовари. Чрез товаренето на прототипи диномично по този начин класът **Trash** не е необходимо да знае с какви типове работи, така че не трябва да се променя въобще когато добавяте нови типове. Това позволява лесно да бъде преизползван на много места в главата.

```
//: c17:trash:Trash.java
// Base class for Trash recycling examples
package c17.trash;
import java.util.*;
import java.lang.reflect.*;

public abstract class Trash {
    private double weight;
    Trash(double wt) { weight = wt; }
    Trash() {}
    public abstract double value();
    public double weight() { return weight; }
    // Sums the value of Trash in a bin:
    public static void sumValue(ArrayList bin) {
        Iterator e = bin.iterator();
        double val = 0.0f;
        while(e.hasNext()) {
            // One kind of RTTI:
            // A dynamically-checked cast
            Trash t = (Trash)e.next();
            val += t.weight() * t.value();
            System.out.println(
                "weight of " +
                // Using RTTI to get type
                // information about the class:
                t.getClass().getName() +
                " = " + t.weight());
        }
        System.out.println("Total value = " + val);
    }
    // Remainder of class provides support for
    // prototyping:
    public static class PrototypeNotFoundException
        extends Exception {}
    public static class CannotCreateTrashException
        extends Exception {}
}
```

```

private static ArrayList trashTypes =
    new ArrayList();
public static Trash factory(Info info)
    throws PrototypeNotFoundException,
    CannotCreateTrashException {
    for(int i = 0; i < trashTypes.size(); i++) {
        // Somehow determine the new type
        // to create, and create one:
        Class tc =
            (Class)trashTypes.get(i);
        if (tc.getName().indexOf(info.id) != -1) {
            try {
                // Get the dynamic constructor method
                // that takes a double argument:
                Constructor ctor =
                    tc.getConstructor(
                        new Class[] {double.class});
                // Call the constructor to create a
                // new object:
                return (Trash)ctor.newInstance(
                    new Object[]{new Double(info.data)});
            } catch(Exception ex) {
                ex.printStackTrace();
                throw new CannotCreateTrashException();
            }
        }
    }
    // Class was not in the list. Try to load it,
    // but it must be in your class path!
    try {
        System.out.println("Loading " + info.id);
        trashTypes.add(
            Class.forName(info.id));
    } catch(Exception e) {
        e.printStackTrace();
        throw new PrototypeNotFoundException();
    }
    // Loaded successfully. Recursive call
    // should work this time:
    return factory(info);
}
public static class Info {
    public String id;
    public double data;
    public Info(String name, double data) {
        id = name;
        this.data = data;
    }
}
} ///:~

```

Основния **Trash** клас и **sumValue()** си остават както преди. Останалата част от класа поддържа прототипния шаблон. Първо виждате два вътрешни класа (които са направени **static**, така че са направени вътрешни само за целите на организацията на кода) описвайщи изключенията които могат да се случат. Това е последвано от **ArrayList trashTypes**, който се използва да държи **Class** манипулатори.

В `Trash.factory()`, `String`ът вътре в `Info` обекта `id` (друга версия на класа `Info` от тази в предишната дискусия) съдържа името на типа на `Trash` който ще се създава; този `String` е сравнен с имената на `Class` в списъка. Ако има съвпадение, това е обектът който ще се създава. Разбира се, има много начини да се определи кокъв обект искате да правите. Този е използван така че информацията прочетена от файл да може да се превърне в обекти.

Щом веднъж сте открили какъв тип `Trash` да създадете, влизат в играта методите на рефлексията. Методът `getConstructor()` взема аргумент който е масив от `Class` манипулатори. Този масив представя аргументите, в техния правилен ред, за конструктора който търсите. Тука масивът се създава динамично чрез синтаксиса на Java 1.1 за създаване на масиви:

```
| new Class[] {double.class}
```

Този код предполага че всеки `Trash` тип има конструктор който взема `double` (и забележете че `double.class` е различен от `Double.class`). Възможно е също, за по-гъвкаво решение, да извикате `getConstructors()`, който връща масив от възможни конструктори.

`getConstructor()` връща манипулатор към `Constructor` обект (част от `java.lang.reflect`). Викате конструктора динамично с метода `newInstance()`, който взима масив от `Object` съдържащ фактическите аргументи. Този масив също е дъзданен със синтаксиса на Java 1.1:

```
| new Object[] {new Double(info.data)}
```

В този случай, обаче, `double` трябва да бъде сложено вътре в обгръщащ клас така че да може да бъде част от този масив от обекти. Процесът на викане на `newInstance()` извлича това `double`, но може да ви се стори малко смущаващо – аргументът може да бъде `double` или `Double`, но когато правите извикването трябва винаги да подадете `Double`. За щастие такъв момент съществува само за примитивните типове.

Щом веднъж разберете как се прави, процесът на създаване на обект само с даден `Class` манипулатор е забележително прост. Рефлексията също позволява да се викат методи по този динамичен начин.

Разбира се, подходящият `Class` манипулатор може и да не присъства в списъка `trashTypes`. В този случай `return`ът във вътрешния цикъл никога не се изпълнява и никога не стигате края. В такъв случай програмата се опитва да оправи нещата товарейки `Class` обект динамично и прибавяйки го към списъка `trashTypes`. Ако и сега не стане нещо е наистина сбъркано, но ако товаренето е успешно `factory` методът се вика рекурсивно за да опита пак.

Както ще видите, красотата на този дизайн е че този код няма нужда да бъде променян, без значение в какви ситуации се използва (предполагайки че `Trash` класовете имат конструктор който приема един `double` аргумент).

Trash подкласове

За да съответстват на схемата на прототипирането, единственото нещо което се изисква от новите подкласове на `Trash` е да съдържат конструктор който взема `double` аргумент и рефлексията в Java 1.1 се справя с всичко останало.

Ето различните типове `Trash`, всеки в свой собствен файл но част от пакета `Trash` (отново за да се подпомогне повторното им използване пак в тази глава):

```
//: c17:trash:Aluminum.java
// The Aluminum class with prototyping
package c17.trash;

public class Aluminum extends Trash {
    private static double val = 1.67f;
    public Aluminum(double wt) { super(wt); }
```

```

public double value() { return val; }
public static void value(double newVal) {
    val = newVal;
}
} //:~

//: c17:trash:Paper.java
// The Paper class with prototyping
package c17.trash;

public class Paper extends Trash {
    private static double val = 0.10f;
    public Paper(double wt) { super(wt); }
    public double value() { return val; }
    public static void value(double newVal) {
        val = newVal;
    }
} //:~

//: c17:trash:Glass.java
// The Glass class with prototyping
package c17.trash;

public class Glass extends Trash {
    private static double val = 0.23f;
    public Glass(double wt) { super(wt); }
    public double value() { return val; }
    public static void value(double newVal) {
        val = newVal;
    }
} //:~

```

А ето нов тип **Trash**:

```

//: c17:trash:Cardboard.java
// The Cardboard class with prototyping
package c17.trash;

public class Cardboard extends Trash {
    private static double val = 0.23f;
    public Cardboard(double wt) { super(wt); }
    public double value() { return val; }
    public static void value(double newVal) {
        val = newVal;
    }
} //:~

```

Може да се види че, освен конструктора, нищо специално няма в тези класове.

Информация за **Trash** от външен файл

Информацията за **Trash** обекти ще се чете от външен файл. Файльт има всичката необходима информация за всяко късче боклук на един ред **Trash:weight**, ето така:

```

c16.Trash.Glass:54
c16.Trash.Paper:22
c16.Trash.Paper:11
c16.Trash.Glass:17
c16.Trash.Aluminum:89

```

```
c16.Trash.Paper:88
c16.Trash.Aluminum:76
c16.Trash.Cardboard:96
c16.Trash.Aluminum:25
c16.Trash.Aluminum:34
c16.Trash.Glass:11
c16.Trash.Glass:68
c16.Trash.Glass:43
c16.Trash.Aluminum:27
c16.Trash.Cardboard:44
c16.Trash.Aluminum:18
c16.Trash.Paper:91
c16.Trash.Glass:63
c16.Trash.Glass:50
c16.Trash.Glass:80
c16.Trash.Aluminum:81
c16.Trash.Cardboard:12
c16.Trash.Glass:12
c16.Trash.Glass:54
c16.Trash.Aluminum:36
c16.Trash.Aluminum:93
c16.Trash.Glass:93
c16.Trash.Paper:80
c16.Trash.Glass:36
c16.Trash.Glass:12
c16.Trash.Glass:60
c16.Trash.Paper:66
c16.Trash.Aluminum:36
c16.Trash.Cardboard:22
```

Забележете че пътя до класа трябва да е даден с името, иначе класът няма да бъде намерен.

За да се направи разбор на реда той се прочита и **indexOf()** метода на **String** дава индекса на `:`. Това първо се използва в метода на **String substring()** за извличането на името на типа на боклука, а после да се вземе теглото превърнато в **double** със **static Double.valueOf()** метода. Методът **trim()** маха шпациите в краищата на стринга.

Парсерът на **Trash** е сложен в отделен файл тъй като пак ще се използва в тази глава:

```
//: c17:trash:ParseTrash.java
// Open a file and parse its contents into
// Trash objects, placing each into a ArrayList
package c17.trash;
import java.util.*;
import java.io.*;

public class ParseTrash {
    public static void
    fillBin(String filename, Fillable bin) {
        try {
            BufferedReader data =
                new BufferedReader(
                    new FileReader(filename));
            String buf;
            while((buf = data.readLine())!= null) {
                String type = buf.substring(0,
                    buf.indexOf(':')).trim();
```

```

        double weight = Double.valueOf(
            buf.substring(buf.indexOf(':') + 1)
            .trim()).doubleValue();
        bin.addTrash(
            Trash.factory(
                new Trash.Info(type, weight)));
    }
    data.close();
} catch(IOException e) {
    e.printStackTrace();
} catch(Exception e) {
    e.printStackTrace();
}
}

// Special case to handle ArrayList:
public static void
fillBin(String filename, ArrayList bin) {
    fillBin(filename, new FillableArrayList(bin));
}
} //:~
```

В **RecycleA.java** се използва **ArrayList** да съдържа **Trash** обекти. Могат също да бъдат използвани и други типове колекции. За да позволи това първата версия на **fillBin()** взема манипулатор към **Fillable**, което е просто **interface** който поддържа метод наречен **addTrash()**:

```

//: c17:trash:Fillable.java
// Any object that can be filled with Trash
package c17.trash;

public interface Fillable {
    void addTrash(Trash t);
}
} //:~
```

Всичко, що употребява този интерфейс може да се използва с **fillBin**. Разбира се, **ArrayList** не прилага **Fillable**, така че няма да работи. Понеже **ArrayList** се използва в повечето примери, има смисъл да се добави втори претоварен метод **fillBin()** който взема **ArrayList**. **ArrayList** може да се използва като **Fillable** обект чрез използване на адаптиращ клас:

```

//: c17:trash:FillableArrayList.java
// Adapter that makes a ArrayList Fillable
package c17.trash;
import java.util.*;

public class FillableArrayList
    implements Fillable {
    private ArrayList v;
    public FillableArrayList(ArrayList vv) { v = vv; }
    public void addTrash(Trash t) {
        v.add(t);
    }
}
} //:~
```

Може да се види че единствената задача на този клас е да свърже метода **addTrash()** на **Fillable** към **add()** на **ArrayList**. С този метод поддръка претовареният метод **fillBin()** може да се използва с **ArrayList** в **ParseTrash.java**:

```

public static void
fillBin(String filename, ArrayList bin) {
```

```
    fillBin(filename, new FillableArrayList(bin));
}
```

Този подход работи с всеки клас-колекция който се използва често. Алтернативно, колекцията може да има собствен адаптер към **Fillable**. (Ще видите това по-късно, в **DynaTrash.java**.)

Рециклиране с прототипиране

Сега може да видите ревизираната версия на **RecycleA.java** с използвана техниката на прототипиране:

```
//: c17:recycleap:RecycleAP.java
// Recycling with RTTI and Prototypes
package c17.recycleap;
import c17.trash.*;
import java.util.*;

public class RecycleAP {
    public static void main(String[] args) {
        ArrayList bin = new ArrayList();
        // Fill up the Trash bin:
        ParseTrash.fillBin("Trash.dat", bin);
        ArrayList
        glassBin = new ArrayList(),
        paperBin = new ArrayList(),
        alBin = new ArrayList();
        Iterator sorter = bin.iterator();
        // Sort the Trash:
        while(sorter.hasNext()) {
            Object t = sorter.next();
            // RTTI to show class membership:
            if(t instanceof Aluminum)
                alBin.add(t);
            if(t instanceof Paper)
                paperBin.add(t);
            if(t instanceof Glass)
                glassBin.add(t);
        }
        Trash.sumValue(alBin);
        Trash.sumValue(paperBin);
        Trash.sumValue(glassBin);
        Trash.sumValue(bin);
    }
} ///:~
```

Всички **Trash** обекти както и **ParseTrash** и поддържащите класове сега са част от пакета **c16.trash** така че лесно се импортират.

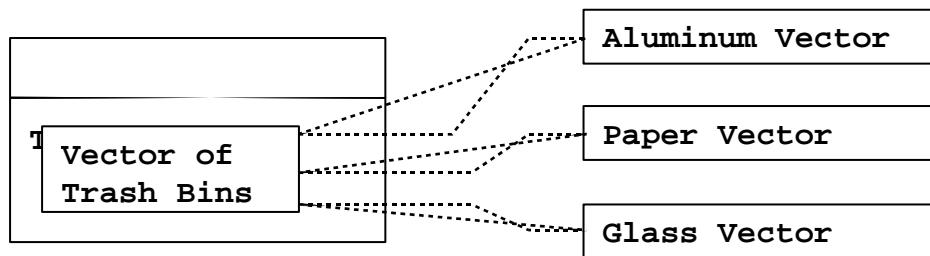
Процесът на отваряне на данновия файл съдържащ описанията на **Trash** и анализа на този файл са обгърнати в **static** метод **ParseTrash.fillBin()**, така че това сега не е част от нашата дизайнърска фокусировка. Ще видите че по протежение на тази глава, без значение какъв клас се добавя, **ParseTrash.fillBin()** ще продължи да работи без промяна, което показва добър дизайн.

В термините на създаване на обекти този дизайн действително твърде много свива местата където трябва ада се правят промени. Обаче има голям проблем с използването на RTTI който добре личи тук. Програмата изглежда да си работи чудесно и все пак не открива никаква мукава, даже ако има мусава в списъка! Това се случва поради употребата на RTTI, което

гледа само за типовете за които сте му казали да гледа. Доказателство че RTTI не е използвано както трябва е че **всеки тип в системата** бива проверяван, наместо един тип или подмножество типове. Както ще видите по-късно, има начини да се използва полиморфизъм когато проверявате за всички типове. Но ако използвате RTTI многоократно по този начин, а и добавяте типове към системата си, лесно е да забравите да направите необходимите промени и ще произведете труден за откриване бъг. Така че си заслужава писането за елиминиране на RTTI в този случай, не само по естетически причини – това дава код по-лесен за поддръжка.

Абстракция на използването

Време е да видим останалата част от дизайна: къде се използват класовете. Понеже слагането в отделенията е особено грубоват акт, защо да не го сложим в клас? Това е принципа че “Ако трябва ада правите нещо грозничко, най-малкото го скрийте в клас.” Това изглежда така:



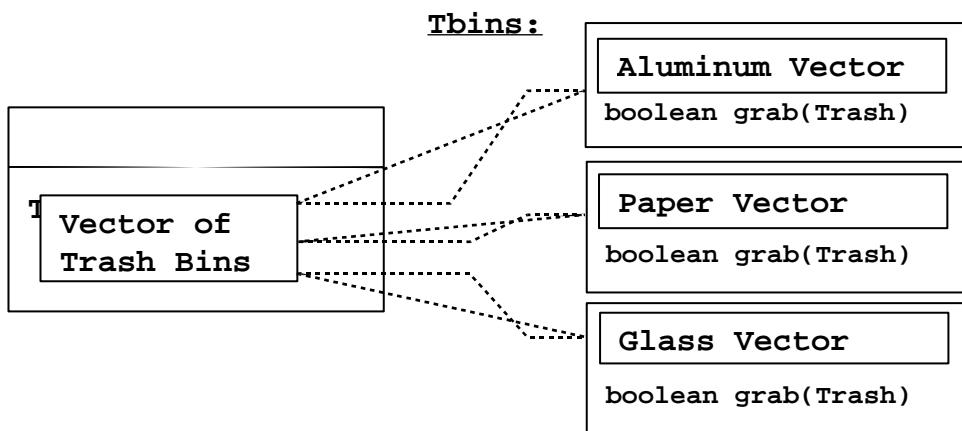
Инициализацията на обекта **TrashSorter** сега трябва да се променя винаги когато се добавя нов тип **Trash** към модела. Може да си представим че класът **TrashSorter** може да изглежда някакси така:

```
class TrashSorter extends ArrayList {  
    void sort(Trash t) { /* ... */ }  
}
```

Тоест, **TrashSorter** е **ArrayList** от манипулатори към **ArrayList**ове от **Trash** манипулатори, а с **add()** може да инсталirate още един, подобно на това:

```
TrashSorter ts = new TrashSorter();  
ts.add(new ArrayList());
```

Сега обаче **sort()** става проблематичен. Как статично кодиран метод ще се справи с факта, че е бил добавен нов тип? За да се реши това, информацията за типа трябва да се махне от **sort()** така че да е нужно само да се вика родов метод, който да се оправя с детайлите за типа. Това, разбира се, е друг начин да се опише динамично свързан метод. Така **sort()** просто щесе придвижи по последователността и ще извика динамично свързан метод за всеки **ArrayList**. Понеже работата на този метод е да награбва парчетата боклук от които се интересува, той е нарече **grab(Trash)**. Структурата сега прилича на тази:



TrashSorter трябва да вика всеки **grab()** метод и да получава различен резултат в зависимост от това какъв вид **Trash** съдържа текущия **ArrayList**. Тоест, всеки **ArrayList** трябва да знае типовете които съдържа. Класическият подход към този проблем е да се създава базов “**Trash bin**” клас и да се наследява за всеки нов тип боклук който искате да съдържа. Ако Java имаше механизъм за параметризиранi типове това вероятно би било най-праволинейния подход. Но вместо твърдото кодиране на всеки клас което би ни докарал този подход, по-нататъшното разглеждане ще покаже по-гъвкав начин.

Основен принцип в ООП е “Използвай даннови членове във вариациите на състоянието, използвай полиморфизъм във вариациите на поведението.” Първо може да си помислите че методът **grab()** непременно има различно поведение за **ArrayList** който съдържа **Paper** от такъв който съдържа **Glass**. Но каквото той прави стриктно зависи от типа, нищо повече. Това може да се интерпретира като различно състояние, а доколкото Java има клас да представи типа (**Class**), това може да бъде използвано за определяне на типа **Trash** който конкретен **Tbin** ще съдържа.

Конструкторът за този **Tbin** изисква да му подадете **Class** по ваш избор. Това казва на **ArrayList** какъв тип ще да съдържа. Тогава методът **grab()** използва **Class BinType** и RTTI да види дали обектът **Trash** който сте дали съответства на типа който трябва да грабни.

Ето цялата програма. Изкоментираните номера (напр. (*1*)) бележат секции които ще се обясняват като се следва кода.

```

//: c17:recycleb:RecycleB.java
// Adding more objects to the recycling problem
package c17.recycleb;
import c17.trash.*;
import java.util.*;

// A vector that admits only the right type:
class Tbin extends ArrayList {
    Class binType;
    Tbin(Class binType) {
        this.binType = binType;
    }
    boolean grab(Trash t) {
        // Comparing class types:
        if(t.getClass().equals(binType)) {
            add(t);
            return true; // Object grabbed
        }
        return false; // Object not grabbed
    }
}

```

```

class TbinList extends ArrayList { //(*1*)
    boolean sort(Trash t) {
        Iterator e = iterator();
        while(e.hasNext()) {
            Tbin bin = (Tbin)e.next();
            if(bin.grab(t)) return true;
        }
        return false; // bin not found for t
    }
    void sortBin(Tbin bin) { // (*2*)
        Iterator e = bin.iterator();
        while(e.hasNext())
            if(!sort((Trash)e.next()))
                System.out.println("Bin not found");
    }
}

public class RecycleB {
    static Tbin bin = new Tbin(Trash.class);
    public static void main(String[] args) {
        // Fill up the Trash bin:
        ParseTrash.fillBin("Trash.dat", bin);

        TbinList trashBins = new TbinList();
        trashBins.add(
            new Tbin(Aluminum.class));
        trashBins.add(
            new Tbin(Paper.class));
        trashBins.add(
            new Tbin(Glass.class));
        // add one line here: (*3*)
        trashBins.add(
            new Tbin(Cardboard.class));

        trashBins.sortBin(bin); // (*4*)

        Iterator e = trashBins.iterator();
        while(e.hasNext()) {
            Tbin b = (Tbin)e.next();
            Trash.sumValue(b);
        }
        Trash.sumValue(bin);
    }
} //:~

```

1. **TbinList** съдържа множество от **Tbin** манипулатори, така че **sort()** може да итерира през **Tbin**овете когато гледа за съвпадение на **Trash** обекта който сте му дали.
2. **sortBin()** позволява да се подаде цял **Tbin** и той минава по целия **Tbin**, взема всяко парче **Trash** и го сортира в съответния конкретен **Tbin**. Задележкете родовостта на този код: той не се променя въобще когато се добавят нови типове. Ако част от вашия код не се нуждае от промяна когато се добави нов тип (или се случи някаква друга промяна) значи имате лесно разширяема система.

3. Сега може да видите колко е лесно да добавите нов тип. Няколко реда трябва да бъдат променени за да се поддържа и добавката. Ако е наистина важно, може да ги намалите чрез по-нататъшна преработка на дизайна.
4. Едно извикване на метод причинява сортирането на **bin** в съответните отделения.

Многократно диспетчиране

Горният дизайн определено удовлетворява. Добавянето на нови типове се свежда до създаването или модифицирането на класове без промените да се разпространяват по останалата част на кода. Освен това RTTI не е "неправилно използвано" като в **RecycleA.java**. Възможно е обаче да се направи една по-нататъшна стъпка и да се възприеме най-чистата гледна точка към RTTI, а именно да се махне въобще от сортирането на боклуците.

За да се направи това, първо трябва да приемем че всички активности които зависят от типа – такива като определяне на типа на боклука и отделянето му в отделно място – ще трябва да се определя от полиморфизма и динамичното свързване.

Предишният пример първо сортираше по тип, последователност от елементи от дадения тип. Но щом се видите да работите с отделни типове спрете и се замислете. Цялата идея на полиморфизма (викането на динамично свързани методи) е да се оправя със зависимата от типа информация вместо вас. Така че защо да се занимаваме с типове?

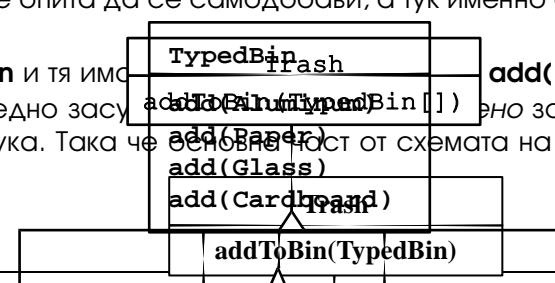
Отговорът е нещо, за което вероятно не сте мислили: Java прави само единично диспетчиране. Тоест ако правите нещо с повече от един обект чийто тип не е известен, Java ще извика механизма на динамичното свързване само за един от типовете. Това не решава проблема, така че трябва да го направите ръчно и фактически да определите собствено поведение в динамичното свързване.

Решението се нарича *multiple dispatching*, което значи съставяне на конфигурация при която едно извикване на метод може да направи повече от едно извикване на динамично свързан метод и по този начин да се определи повече от един тип. За да се получи този ефект, трябва да работите с йерархия на повече от един тип: нуждаете се от йерархия на типовете за всяко диспетчиране. Следния пример работи с две йерархии: съществуващото семейство **Trash** и йерархията на типовете на отделните кошчета където се слагат различните видове боклук. Тази втората йерархия не винаги е очевидна и трябва да бъде създадена в случая за да се направи двойното диспетчиране (в случая ще има две диспетчирания само и това се нарича *double dispatching*).

Реализация на двойното диспетчиране

Помнете, че полиморфизът се случва само чрез викане на методи, така че ако искате да стане двойно диспетчиране, трябва да има две викания на методи: по един за определяне на типа във всяка от йерархията. В йерархията **Trash** ще има нов метод наречен **addToBin()**, който взема за аргумент масив от **TypedBin**. Той използва този масив за да пробяга биновете (отделните кошчета) и да се опита да се самодобави, а тук именно е двойното диспетчиране.

Новата йерархия е **TypedBin** и тя има **addBin(TypedBin[])** който също се използва полиморфно. Но има още едно засущение за да приема аргументи от различните типове на боклука. Така че основната част от схемата на двойното диспетчиране е също претоварването.



Aluminum	Paper	Glass	Cardboard
addToBin()	addToBin()	addToBin()	addToBin()

AluminumBin	PaperBin	GlassBin	CardboardBin
add(Aluminum)	add(Paper)	add(Glass)	add(Cardboard)

Преформулирането на програмата довежда до дилема: сега за базовия клас **Trash** е необходимо да съдържа метод **addToBin()**. Единият подход е да се копира всички код и да се промени базовия клас. Другия подход, който се налага когато не разполагате със сурса, е да се сложи метода **addToBin()** в **interface**, да се остави **Trash** без да се пипа и да се наследят нови конкретни типове **Aluminum**, **Paper**, **Glass** и **Cardboard**. Тук ще се приложи този подход.

Повечето класове в този дизайн трябва да бъдат **public**, така че са сложени в техни собствени класове. Ето интерфейса:

```
//: c17:doubledispatch:TypedBinMember.java
// An interface for adding the double dispatching
// method to the trash hierarchy without
// modifying the original hierarchy.
package c17.doubledispatch;

interface TypedBinMember {
    // The new method:
    boolean addToBin(TypedBin() tb);
} ///:~
```

Във всеки конкретен подтип на **Aluminum**, **Paper**, **Glass** и **Cardboard** методът **addToBin()** който е в **interface TypedBinMember** се прилага, но изглежда сякаш кодът е един и същ във всеки от случаите:

```
//: c17:doubledispatch:DDAluminum.java
// Aluminum for double dispatching
package c17.doubledispatch;
import c17.trash.*;

public class DDAluminum extends Aluminum
    implements TypedBinMember {
    public DDAluminum(double wt) { super(wt); }
    public boolean addToBin(TypedBin() tb) {
        for(int i = 0; i < tb.length; i++)
            if(tb(i).add(this))
                return true;
        return false;
    }
} ///:~

//: c17:doubledispatch:DDPaper.java
// Paper for double dispatching
package c17.doubledispatch;
import c17.trash.*;

public class DDPaper extends Paper
    implements TypedBinMember {
    public DDPaper(double wt) { super(wt); }
    public boolean addToBin(TypedBin() tb) {
        for(int i = 0; i < tb.length; i++)
            if(tb(i).add(this))
                return true;
        return false;
    }
} ///:~

//: c17:doubledispatch:DDGlass.java
```

```

// Glass for double dispatching
package c17.doubledispatch;
import c17.trash.*;

public class DDGlass extends Glass
    implements TypedBinMember {
    public DDGlass(double wt) { super(wt); }
    public boolean addToBin(TypedBin() tb) {
        for(int i = 0; i < tb.length; i++)
            if(tb(i).add(this))
                return true;
        return false;
    }
} ///:~

//: c17:doubledispatch:DDCardboard.java
// Cardboard for double dispatching
package c17.doubledispatch;
import c17.trash.*;

public class DDCardboard extends Cardboard
    implements TypedBinMember {
    public DDCardboard(double wt) { super(wt); }
    public boolean addToBin(TypedBin() tb) {
        for(int i = 0; i < tb.length; i++)
            if(tb(i).add(this))
                return true;
        return false;
    }
} ///:~

```

Кодът във всеки **addToBin()** вика **add()** за всеки **TypedBin** обект в масива. Но забележете аргумента: **this**. Типа на **this** е различен за всеки подклас на **Trash**, така че кодът е различен. (Макар че този код би спечелил ако никога се добавят параметризираны типове в Java.) Така че това е първата част от двойното диспетчирание, понеже щом се озовете в този метод знаете дали е **Aluminum**, или **Paper** и т.н. По време на извикването на **add()** тази информация се подава чрез типа на **this**. Компилаторът решава извикването към правилната версия на претоварения **add()**. Но понеже **tb(i)** дава манипулятор към базовия тип **TypedBin**, това извикване ще завърши извиквайки различен метод в зависимост от това кой **TypedBin** е текущо избран. Това е второто диспетчирание.

Ето базовия клас за **TypedBin**:

```

//: c17:doubledispatch:TypedBin.java
// ArrayList that knows how to grab the right type
package c17.doubledispatch;
import c17.trash.*;
import java.util.*;

public abstract class TypedBin {
    ArrayList v = new ArrayList();
    protected boolean addIt(Trash t) {
        v.add(t);
        return true;
    }
    public Iterator elements() {
        return v.iterator();
    }
}

```

```

public boolean add(DDAluminum a) {
    return false;
}
public boolean add(DDPaper a) {
    return false;
}
public boolean add(DDGlass a) {
    return false;
}
public boolean add(DDCardboard a) {
    return false;
}
} //:~

```

Може да се види че всички претоварени методи **add()** ще връщат **false**. Ако методът не е претоварен в извлечен клас, той ще продължи да връща **false**, а извикващият (**addToBin()** в този случай) ще предполага че текущият **Trash** обект не е бил успешно добавен към колекцията и ще продължи да търси правилната колекция.

Във всеки от подкласовете на **TypedBin** само един претоварен метод е подтиснат, съответно на типа на боклуците които ще се слагат в кошчето което се създава. Например **CardboardBin** подтиска **add(DDCardboard)**. Подтиснатия метод добавя обекта-боклук към колекцията си и връща **true**, докато с другите **add()** методи в **CardboardBin** продължават да връщат **false**, понеже не са били подтиснати. Това е друг пример където параметризиран тип в Java би позволил автоматична генерация на код. (С **template** в C++ нямаше да е необходимо явно да се пишат подкласове или да се слага метод **addToBin()** в **Trash**.)

Понеже н този пример боклуците бяха приспособени и поставени в отделна директория, ще ви трябва и нов даннов файл с боклуци за да накарате примера да работи. Ето един възможен **DDTrash.dat**:

```

c17.DoubleDispatch.DDGlass:54
c17.DoubleDispatch.DDPaper:22
c17.DoubleDispatch.DDPaper:11
c17.DoubleDispatch.DDGlass:17
c17.DoubleDispatch.DDAluminum:89
c17.DoubleDispatch.DDPaper:88
c17.DoubleDispatch.DDAluminum:76
c17.DoubleDispatch.DDCardboard:96
c17.DoubleDispatch.DDAluminum:25
c17.DoubleDispatch.DDAluminum:34
c17.DoubleDispatch.DDGlass:11
c17.DoubleDispatch.DDGlass:68
c17.DoubleDispatch.DDGlass:43
c17.DoubleDispatch.DDAluminum:27
c17.DoubleDispatch.DDCardboard:44
c17.DoubleDispatch.DDAluminum:18
c17.DoubleDispatch.DDPaper:91
c17.DoubleDispatch.DDGlass:63
c17.DoubleDispatch.DDGlass:50
c17.DoubleDispatch.DDGlass:80
c17.DoubleDispatch.DDAluminum:81
c17.DoubleDispatch.DDCardboard:12
c17.DoubleDispatch.DDGlass:12
c17.DoubleDispatch.DDGlass:54
c17.DoubleDispatch.DDAluminum:36
c17.DoubleDispatch.DDAluminum:93

```

```
c17.DoubleDispatch.DDGlass:93
c17.DoubleDispatch.DDPaper:80
c17.DoubleDispatch.DDGlass:36
c17.DoubleDispatch.DDGlass:12
c17.DoubleDispatch.DDGlass:60
c17.DoubleDispatch.DDPaper:66
c17.DoubleDispatch.DDAluminum:36
c16.DoubleDispatch.DDCardboard:22
```

Ето останалата част на програмата:

```
//: c17:doubledispatch:DoubleDispatch.java
// Using multiple dispatching to handle more
// than one unknown type during a method call.
package c17.doubledispatch;
import c17.trash.*;
import java.util.*;

class AluminumBin extends TypedBin {
    public boolean add(DDAluminum a) {
        return addIt(a);
    }
}

class PaperBin extends TypedBin {
    public boolean add(DDPaper a) {
        return addIt(a);
    }
}

class GlassBin extends TypedBin {
    public boolean add(DDGlass a) {
        return addIt(a);
    }
}

class CardboardBin extends TypedBin {
    public boolean add(DDCardboard a) {
        return addIt(a);
    }
}

class TrashBinSet {
    private TypedBin() binSet = {
        new AluminumBin(),
        new PaperBin(),
        new GlassBin(),
        new CardboardBin()
    };
    public void sortIntoBins(ArrayList bin) {
        Iterator e = bin.iterator();
        while(e.hasNext()) {
            TypedBinMember t =
                (TypedBinMember)e.next();
            if(!t.addToBin(binSet))
                System.err.println("Couldn't add " + t);
        }
    }
}
```

```

    }
    public TypedBin() binSet() { return binSet; }
}

public class DoubleDispatch {
    public static void main(String[] args) {
        ArrayList bin = new ArrayList();
        TrashBinSet bins = new TrashBinSet();
        // ParseTrash still works, without changes:
        ParseTrash.fillBin("DDTrash.dat", bin);
        // Sort from the master bin into the
        // individually-typed bins:
        bins.sortIntoBins(bin);
        TypedBin() tb = bins.binSet();
        // Perform sumValue for each bin...
        for(int i = 0; i < tb.length; i++)
            Trash.sumValue(tb(i).v);
        // ... and for the master bin
        Trash.sumValue(bin);
    }
} //:~

```

TrashBinSet капсулира всичките различни типове **TypedBin**ове, заедно с метода **sortIntoBins()**, в който става цялото двойно диспечиране. Може да се види че щом структурата веднъж е установена, сортирането в различните **TypedBin**ове е забележително просто. Освен това ефективността на двета динамично свързани метода е вероятно по-голяма отколкото на какъвто и да е друг начин на сортиране.

Забележете леснотата на използване на тази система в **main()**, както и независимостта от каквато и да било специфична за типа информация в **main()**. Всички други методи които говорят само на интерфейса на базовия клас **Trash** ще бъдат еднакво нечувствителни към промени в типовете **Trash**.

Промените необходими за добавянето на нов тип са относително изолирани: наследявате новия тип от **Trash** с неговия си метод **addToBin()**, после наследявате новия **TypedBin** (това е наистина само копиране и просто редактиране), а накрая добавяте новия тип към агрегатната инициализация на **TrashBinSet**.

Шаблонът “посетител”

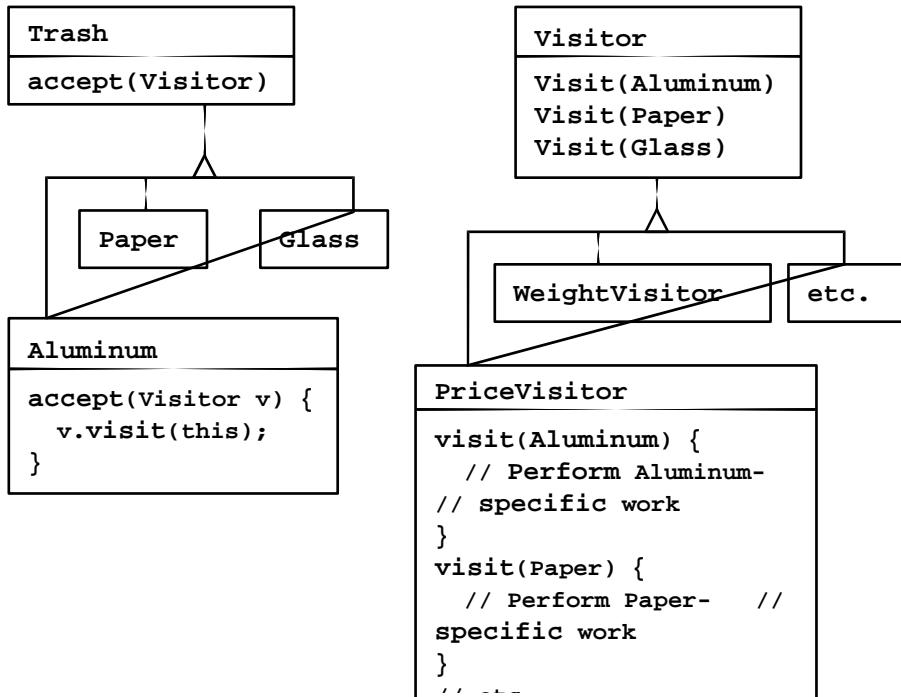
Сега да видим приложението на шаблон със съвсем различна цел към проблема за сортирането на отпадъци.

При този шаблон вече не ни интересува оптимизирането на добавянето на нови типове **Trash** към системата. Разбира се, този шаблон прави добавянето на нови типове към **Trash** по-сложено. Предположението е че имате йерархия на първичен клас която е фиксирана; да кажем че е от друг доставчик и не може да правите промени в нея. Обаче вие искате да добавите нови полиморфни методи към нея йерархия, което значи че нормално ще трябва да добавите нещо към интерфейса на базовия клас. Така че дилемата е че трябва да добавите методи към базовия клас, но не може да пипате базовия клас. Как да се заобиколи това?

Шаблонът за проектиране, който решава този проблем се нарича “посетител” (последния в книгата *Design Patterns*) и той се строи върху схемата на двойното диспечиране която разгледахме.

Наблюдателският шаблон позволява да се разшири интерфейса на базовия клас чрез създаване на нова йерархия от типа **Visitor** за да се реализират операциите върху първичния

типове. Обектите от първичния тип просто "приемат" посетителя, после викат динамично свързан метод на посетителя. Това изглежда така:



Сега ако **v** е **Visitable** манипулятор към **Aluminum** обект, кодът:

```

PriceVisitor pv = new PriceVisitor();
v.accept(pv);
  
```

Предизвика две полиморфни викания на методи: първото за извикване на версията на **accept()** от **Aluminum** и следващото вътре в **accept()** когато специфичната версия на **visit()** се вика динамично чрез манипулатора на базовия клас **Visitor v**.

Тази конфигурация означава че нови свойства могат да се добавят към системата във формата на подкласове на **Visitor**. Йерархията **Trash** не е необходимо да се засяга. Това е пряката полза от шаблона-наблюдател: може да се добавя нова полиморфна функционалност към класова йерархия без да се засяга самата тя (щом веднъж методите **accept()** са били инсталирани). Забележете че ползата тук е добре дошла но не е точно това, което искахме да направим като започвахме, така че на пръв поглед може да решите че това не е желаното решение.

Но да погледнем нещо, което е станало: решението с посетителя избягва сортирането на главната **Trash** последователност в индивидуални последователности с конкретни типове. Така може да оставите всичко в единствена главна последователност и просто да я пробягате с подходящ посетител за да постигнете целта. Макар и това поведение да изглежда като страничен ефект от посетителя, то ни дава каквото искаме (избягвайки RTTI).

Двойното диспечиране в шаблона-посетител се грижи за определянето и на типа на **Trash** и типа на **Visitor**. В следния пример има две реализации на посетител **Visitor**: **PriceVisitor** за определяне и сумиране на цената и **WeightVisitor** за теглата.

Може да видите всичко това претворено в нова, подобрена версия на програмата за рециклирането. Както при **DoubleDispatch.java**, класът **Trash** си стои и нов интерфейс е добавен заради метода **accept()**:

```

//: c17:trashvisitor:Visitable.java
// An interface to add visitor functionality to
// the Trash hierarchy without modifying the
// base class.
  
```

```

package c17.trashvisitor;
import c17.trash.*;

interface Visitable {
    // The new method:
    void accept(Visitor v);
} //:~

```

Подтиповете **Aluminum**, **Paper**, **Glass** и **Cardboard** реализират **accept()** метода:

```

//: c17:trashvisitor:VAluminum.java
// Aluminum for the visitor pattern
package c17.trashvisitor;
import c17.trash.*;

public class VAluminum extends Aluminum
    implements Visitable {
    public VAluminum(double wt) { super(wt); }
    public void accept(Visitor v) {
        v.visit(this);
    }
} //:~

//: c17:trashvisitor:VPaper.java
// Paper for the visitor pattern
package c17.trashvisitor;
import c17.trash.*;

public class VPaper extends Paper
    implements Visitable {
    public VPaper(double wt) { super(wt); }
    public void accept(Visitor v) {
        v.visit(this);
    }
} //:~

//: c17:trashvisitor:VGlass.java
// Glass for the visitor pattern
package c17.trashvisitor;
import c17.trash.*;

public class VGlass extends Glass
    implements Visitable {
    public VGlass(double wt) { super(wt); }
    public void accept(Visitor v) {
        v.visit(this);
    }
} //:~

//: c17:trashvisitor:VCardboard.java
// Cardboard for the visitor pattern
package c17.trashvisitor;
import c17.trash.*;

public class VCardboard extends Cardboard
    implements Visitable {
    public VCardboard(double wt) { super(wt); }
    public void accept(Visitor v) {

```

```
    v.visit(this);
}
} //:~
```

Понеже няма нищо конкретно в базовия клас на **Visitor**, той може да бъде създаден като **interface**:

```
//: c17:trashvisitor:Visitor.java
// The base interface for visitors
package c17.trashvisitor;
import c17.trash.*;

interface Visitor {
    void visit(VAluminum a);
    void visit(VPaper p);
    void visit(VGlass g);
    void visit(VCardboard c);
} //:~
```

Пак новите **Trash** типове са създадени в отделна поддиректория. Новия **Trash** даннов файл е **VTrash.dat** и изглежда така:

```
c17.TrashVisitor.VGlass:54
c17.TrashVisitor.VPaper:22
c17.TrashVisitor.VPaper:11
c17.TrashVisitor.VGlass:17
c17.TrashVisitor.VAluminum:89
c17.TrashVisitor.VPaper:88
c17.TrashVisitor.VAluminum:76
c17.TrashVisitor.VCardboard:96
c17.TrashVisitor.VAluminum:25
c17.TrashVisitor.VAluminum:34
c17.TrashVisitor.VGlass:11
c17.TrashVisitor.VGlass:68
c17.TrashVisitor.VGlass:43
c17.TrashVisitor.VAluminum:27
c17.TrashVisitor.VCardboard:44
c17.TrashVisitor.VAluminum:18
c17.TrashVisitor.VPaper:91
c17.TrashVisitor.VGlass:63
c17.TrashVisitor.VGlass:50
c17.TrashVisitor.VGlass:80
c17.TrashVisitor.VAluminum:81
c17.TrashVisitor.VCardboard:12
c17.TrashVisitor.VGlass:12
c17.TrashVisitor.VGlass:54
c17.TrashVisitor.VAluminum:36
c17.TrashVisitor.VAluminum:93
c17.TrashVisitor.VGlass:93
c17.TrashVisitor.VPaper:80
c17.TrashVisitor.VGlass:36
c17.TrashVisitor.VGlass:12
c17.TrashVisitor.VGlass:60
c17.TrashVisitor.VPaper:66
c17.TrashVisitor.VAluminum:36
c16.TrashVisitor.VCardboard:22
```

Останалата част на програмата създава специфични типове **Visitor** и ги изпраща през единствен списък на **Trash** обекти:

```
//: c17:trashvisitor:TrashVisitor.java
// The "visitor" pattern
package c17.trashvisitor;
import c17.trash.*;
import java.util.*;

// Specific group of algorithms packaged
// in each implementation of Visitor:
class PriceVisitor implements Visitor {
    private double alSum; // Aluminum
    private double pSum; // Paper
    private double gSum; // Glass
    private double cSum; // Cardboard
    public void visit(VAluminum al) {
        double v = al.weight() * al.value();
        System.out.println(
            "value of Aluminum= " + v);
        alSum += v;
    }
    public void visit(VPaper p) {
        double v = p.weight() * p.value();
        System.out.println(
            "value of Paper= " + v);
        pSum += v;
    }
    public void visit(VGlass g) {
        double v = g.weight() * g.value();
        System.out.println(
            "value of Glass= " + v);
        gSum += v;
    }
    public void visit(VCardboard c) {
        double v = c.weight() * c.value();
        System.out.println(
            "value of Cardboard = " + v);
        cSum += v;
    }
    void total() {
        System.out.println(
            "Total Aluminum: $" + alSum + "\n" +
            "Total Paper: $" + pSum + "\n" +
            "Total Glass: $" + gSum + "\n" +
            "Total Cardboard: $" + cSum);
    }
}

class WeightVisitor implements Visitor {
    private double alSum; // Aluminum
    private double pSum; // Paper
    private double gSum; // Glass
    private double cSum; // Cardboard
    public void visit(VAluminum al) {
        alSum += al.weight();
        System.out.println("weight of Aluminum = "
```

```

        + al.weight());
    }
    public void visit(VPaper p) {
        pSum += p.weight();
        System.out.println("weight of Paper = "
            + p.weight());
    }
    public void visit(VGlass g) {
        gSum += g.weight();
        System.out.println("weight of Glass = "
            + g.weight());
    }
    public void visit(VCardboard c) {
        cSum += c.weight();
        System.out.println("weight of Cardboard = "
            + c.weight());
    }
    void total() {
        System.out.println("Total weight Aluminum:"
            + alSum);
        System.out.println("Total weight Paper:"
            + pSum);
        System.out.println("Total weight Glass:"
            + gSum);
        System.out.println("Total weight Cardboard:"
            + cSum);
    }
}

public class TrashVisitor {
    public static void main(String[] args) {
        ArrayList bin = new ArrayList();
        // ParseTrash still works, without changes:
        ParseTrash.fillBin("VTrash.dat", bin);
        // You could even iterate through
        // a list of visitors!
        PriceVisitor pv = new PriceVisitor();
        WeightVisitor wv = new WeightVisitor();
        Iterator it = bin.iterator();
        while(it.hasNext()) {
            Visitable v = (Visitable)it.next();
            v.accept(pv);
            v.accept(wv);
        }
        pv.total();
        wv.total();
    }
} //:~

```

Забележете че видът на **main()** пак се е променил. Сега има само един **Trash** кош. Двата **Visitor** са приети във всеки елемент от последователността, те изпълняват своите операции. Посетителите държат свои собствени вътрешни данни за да се оправят със сметката на цените и теглата.

Накрая, няма идентификация на типа по време на изпълнение с изключение на неизбежния каст на **Trash** когато се вадят неща от последователността. Това също би могло да бъде елиминирано ако се въведат параметризирантипове в Java.

Един начин да се различи това решение от решението с двойното диспечиране от преди малко е да се забележи че, в последното, само един от претоварените методи, **add()**, беше подтиснат когато се създаваше подклас, докато тук всеки от претоварените **visit()** методи е подтиснат във всеки подклас на **Visitor**.

Повече сдвояване?

Има много още код тута и определено има тясна връзка между **Trash** юерархията и **Visitor** юерархията. Има обаче и голямо сцепление между респективните множества от класове: всеки от тях прави едно нещо (**Trash** описва **Trash**, докато **Visitor** описва действията правени от **Trash**), което е индикация за добър проект. Разбира се, в този случай това работи добре само ако добавяте нови **Visitors**, но става и когато добавяте нови типове **Trash**.

Малкото сдвояване между класовете и високата стегнатост вътре в класа е определена цел на дизайна. Прилагано без много мислене, обаче, това може да предотврати достигането на по-елегантен дизайн. Изглежда че някои класове неизбежно имат определена интимност помежду си. Това често става на двойки които може би трябва да се наричат куплети, например колекции и итератори. Двойката **Trash-Visitor** от по-горе излиза че е подобен куплет.

RTTI считана вредна?

Различни проекти в тази глава се опитват да махнат RTTI, което може да създаде впечатлението че е "считана за вредна" (присъдата използвана срещу бедното, нещастно **goto**, което поради нея никога не беше включено в Java). Това не е вярно; **неправилното използване** на RTTI е проблемът. Причината нашите проекти да махнат RTTI е че неправилното прилагане на тази черта пречи на разширяемостта, докато декларираната цел беше да може да се добави нов тип към системата с минимално отражение върху останалия код. Понеже RTTI често се използва неправилно даoglажда всеки един тип във вашата система, това причинява неразширяемост на кода: когато добавяте нов тип, трябва да изловите всички код в който се използва RTTI и ако сгрешите нещо нямаете никаква помощ от компилатора.

Обаче RTTI не създава автоматично неразширяем код. Нека да преразгледаме примера с рециклирата още веднъж. Този път ще бъде въведен нов инструмент, който аз наричам **TypeMap**. Той съдържа **HashMap** който държи **ArrayList**ове, но интерфейсът е прост: може да **add()** нов обект и може да **get()** един **ArrayList** съдържащ всички обекти от конкретен тип. Ключовете за съдържания **HashMap** са типовете в асоциирания **ArrayList**. Красотата на този дизайн (предложен от Larry O'Brien) е че **TypeMap** динамично добавя нова двойка щом намери нов тип, така че винаги когато добавите нов тип към системата (даже ако това стане по време на изпълнение), той се приспособява.

Нашият дизайн пак ще се построи по схемата на **Trash** типовете в **package c16.Trash** (и **Trash.dat** файла използван там може да се използва без промяна):

```
//: c17:dynatrash:DynaTrash.java
// Using a HashMap of ArrayLists and RTTI
// to automatically sort trash into
// vectors. This solution, despite the
// use of RTTI, is extensible.
package c17.dynatrash;
import c17.trash.*;
import java.util.*;

// Generic TypeMap works in any situation:
class TypeMap {
    private HashMap t = new HashMap();
    public void add(Object o) {
```

```

Class type = o.getClass();
if(t.containsKey(type))
    ((ArrayList)t.get(type)).add(o);
else {
    ArrayList v = new ArrayList();
    v.add(o);
    t.put(type,v);
}
}
public ArrayList get(Class type) {
    return (ArrayList)t.get(type);
}
public Iterator keys() {
    return t.keySet().iterator();
}
// Returns handle to adapter class to allow
// callbacks from ParseTrash.fillBin():
public Fillable filler() {
    // Anonymous inner class:
    return new Fillable() {
        public void addTrash(Trash t) { add(t); }
    };
}
}

public class DynaTrash {
    public static void main(String[] args) {
        TypeMap bin = new TypeMap();
        ParseTrash.fillBin("Trash.dat",bin.filler());
        Iterator keys = bin.keys();
        while(keys.hasNext())
            Trash.sumValue(
                bin.get((Class)keys.next()));
    }
} ///:~

```

Макар и мощна, дефиницията на **TypeMap** е проста. Тя съдържа **HashMap** и методът **add()** върши повечето работа. Когато вие **add()** нов обект, манипулаторът на **Class** обекта за него тип се извлича. Той се използва като ключ да се види дали някой **ArrayList** който съдържа обекти от него тип вече е представен в **HashMap**. Ако е така, този **ArrayList** се извлича и обектът се добавя към **ArrayList**. Ако не, **Class** обект и нов **ArrayList** се добавят като двойка ключ-стойност.

Може да вземете **Iterator** за всичките **Class** обекти от **keys()** и да използвате всеки **Class** обект за намиране на съответния **ArrayList** с **get()**. И това е всичко.

Методът **filler()** е интересен понеже се ползва от дизайна на **ParseTrash.fillBin()**, който не само се опитва да запълни **ArrayList** ами което и да е прилагащо **Fillable** интерфейс с неговия **addTrash()** метод. Всичко което иска **filler()** за да работи е да се върне манипулатор към **interface** който прилага **Fillable**, а после този манипулатор може да се използва като аргумент на **fillBin()** подобно на това:

```
| ParseTrash.fillBin("Trash.dat", bin.filler());
```

За да се даде този манипулатор се използва *анонимен вътрешен клас* (описани са в глава 7). Никога не ви трябва именуван клас за реализация на **Fillable**, просто е необходим манипулатор на такъв клас само, така че това е правилна употреба на анонимни вътрешни класове.

Интересно за този дизайн е, че макар и да не беше създаден да поддържа сортиране, `fillBin()` сортира всеки път когато вмъква **Trash** обект в **bin**.

Повечето от класа **class DynaTrash** би трябвало да е познато от предишни примери. Този път, заместо да се слагат нови **Trash** обекти в **bin** от тип **ArrayList**, **bin** е от типа **TypeMap**, така че когато боклукут е хвърлен в **bin** той веднага се сортира от вътрешния сортиращ механизъм на **TypeMap**. Пребројдането на **TypeMap** и оперирането на всеки индивидуален **ArrayList** става просто нещо:

```
Iterator keys = bin.keys();
while(keys.hasNext())
    Trash.sumValue(
        bin.get((Class)keys.next()));
```

Както може да видите, добавянето на нов тип в системата няма да промени този код въобще, нито пък кода в **TypeMap**. Това сигурно е най-малкото по размери решение на проблема, може да се спори и че е най-елегантното. То много разчита на RTTI, но забележете че всяка двойка ключ-стойност в **HashMap** гледа само за един тип. Освен това няма начин да "забравите" да добавите подходящия код към тази система когато добавяте нов тип, тъй като няма код за добавяне.

Резюме

Свършвайки с дизайна **TrashVisitor.java** който съдържа повече код отколкото предишните може да изглежда на пръв поглед контрапродуктивни. Струва си да се погледне какво се целеше при всичките тези проекти да се направи. Шаблонът за проектиране изобщо решава въпроса за разделяне на нещата които се променят от тези които остават същите. "Нещата които се променят" могат да се отнасят за много различни видове промени. Може промяната да е следствие от поставянето на програмата в друго окръжение или има промяна в текущото окръжение (това би могло да бъде: "Потребителят искат да добави нов тип фигура към текущата диаграма"). Или, както в този случай, промяната може да е от еволюцията на самия код. Докато предишните версии на примера с боклуците подчертаваха аспекта на добавянето на типове **Trash** към системата, **TrashVisitor.java** позволява лесно да се добави функционалност без промени в йерархията на **Trash**. Има повече код в **TrashVisitor.java**, но добавянето на нова функционалност към **Visitor** е евтино. Ако това е нещо което се случва много, тогава си струва наличието на повече код щом става по-лесно.

Откриването на вектора на промяната не е тривиален въпрос; то не е нещо, което анализатор може да види преди програмата да е видяла първата си реализация. Нужната информация няма да е достъпна вероятно до късните фази на проектирането или може би на реализациите на проекта. В случая на добавяне на нови типове (което беше във фокуса на повечето "рециклиаторни" примери) бихте могли да разберете че ви трябва конкретна йерархия чак когато сте във фазата на поддържането и разширявате системата!

Едно от най-важните неща които ще научите покрай шаблоните за проектиране е: "ООП е изцяло полиморфизъм." Това твърдение може да доведе до синдрома "двегодишно (дете-бр.) с чук" (всичко изглежда като гвоздей). С други думи, достатъчно трудно е да "имаш" полиморфизъм, а когато го постигнеш, опитваш се да пъхнеш всичките си проекти в него калъп.

Това което казват шаблоните за проектиране е че ООП не е само полиморфизъм. То е "отделяне на нещата които се менят от тези които остават същите." Полиморфизъмът е особено актуален начин да се прави това и излиза че е полезен ако програмният език директно поддържа полиморфизъм (така че не трябва да го пишете сами, което би направило работата забранително скъпа). Но шаблоните за проектиране изобщо показват други пътища за постигане на основната цел, веднъж като ви се отворят очите ще започнете да търсите по-продуктивни решения.

Откакто излезе книгата *Design Patterns* и оказа своето влияние, хората търсят нови шаблони. Може да се очаква че ще се появят нови с времето. Ето няколко сайта препоръчани от Jim Coplien, от славата на C++ (<http://www.bell-labs.com/~cope>), който е един от главните движатели на напредъка с шаблоните:

<http://st-www.cs.uiuc.edu/users/patterns>
<http://c2.com/cgi/wiki>
<http://c2.com/ppr>
<http://www.bell-labs.com/people/cope/Patterns/Process/index.html>
<http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns>
<http://st-www.cs.uiuc.edu/cgi-bin/wikic/wikic>
<http://www.cs.wustl.edu/~schmidt/patterns.html>
<http://www.espinc.com/patterns/overview.html>

Има също годишна конференция наречена PLoP която издава журнал, третият от които излезе късно през 1997 (всички се публикуват от Addison-Wesley).

Упражнения

1. С **SingletonPattern.java** като стартова точка създайте клас който управлява фиксиран брой свои собствени обекти.
2. Добавете клас **Triangle** към **ShapeFactory1.java**
3. Добавете клас **Triangle** към **ShapeFactory2.java**
4. Добавете нов тип **GameEnvironment** наречен **GnomesAndFairies** към **GameEnvironment.java**
5. Променете **ShapeFactory2.java** така че да използва *Abstract Factory* за създаване на различни множества от форми (например, един конкретен тип обекти-фабрики създава "дебели форми," друг създава "тънки форми," но всеки обект-фабрика да може да създава всички форми: кръгове, квадрати, триъгълници и т.н.).
6. Добавете клас **Plastic** към **TrashVisitor.java**.
7. Добавете клас **Plastic** към **DynaTrash.java**.
8. Създайте среда за бизнес-моделиране с три типа **Inhabitant**: **Dwarf** (за инженери), **Elf** (за отдела продажби) и **Troll** (за управлението). Сега създайте клас наречен **Project** който създава различни жители и ги кара да **interact()** (взаимодействият-б.пр.) един с друг чрез многократно диспечиране.
9. Променете горния пример да станат взаимодействията по-детайлни. Всеки **Inhabitant** може случайно да произведе **Weapon** чрез **getWeapon()**: **Dwarf** използва **Jargon** или **Play**, **Elf** използва **InventFeature** или **SellImaginaryProduct**, **Troll** използва **Edict** и **Schedule**. Вие трябва да решите кои оръжия "печелят" и "губят" във всяко взаимодействие (като в **PaperScissorsRock.java**). Добавете член-функция **battle()** към **Project** която взема два **Inhabitants** и ги вкарва в бой един с друг. Сега създайте **meeting()** член-функция за **Project** която създава групи **Dwarf**, **Elf** и **Manager** и ги сбива една с друга докато останат членове само на една група. Тези са "победителите."

10. Реализирайте верига на отговорност да създаде "експертна система" която решава проблеми чрез опитване на начини един след друг докато някой се окаже успешен. Трябва да може динамично да добавяте решения към експертната система. Тестът за решението ще бъде просто съвпадение на стрингове, но когато решението бива, експертната система трябва да връща съответния тип problemSolver обект. Какъв друг шаблон/шаблони се появяват тук?

A: Java Native Interface (JNI)

¹ Езикът Java и неговия стандартен API са достатъчно богати да може да се пишат пълноцени приложения. Но в някои случаи трябва да се вика не-Java код; например ако искате да получите достъп до специфични за ОС черти, да направите интерфейс със специални хардуерни устройства, да използвате вече съществуваща не-Java кодова наличност или да реализирате критична откъв време на изпълнение секция от кода.

Интерфейсът с не-Java код изисква специална поддръжка в компилатора и виртуалната машина, а и допълнителни инструменти за проектиране на Java кода в не-Java кода. (Има също и прост подход: в глава 15, секцијата със заглавие "Web приложение" съдържа пример за връзка към не-Java код чрез използване на стандартния вход и изход.) Стандартното решение за викане на не-Java код които се дава от Javasoft е наречено *Java Native Interface*, с което ще ви запознаем в това приложение. Това не е третиране в дълбочина и в някои случаи се предполага да имате знания за свързани с въпроса концепции и техники.

Java Native Interface

JNI е доста богат програмен интерфейс който позволява да се викат "естествени" за машината методи от Java приложение. Той бе добавен в Java 1.1, обслужвайки в някаква степен съвместимостта с предишния, от Java 1.0 еквивалент, native method interface (NMI). NMI има характеристики на дизайна си които го правят неподходящ за съвместяване с всички виртуални машини. Поради тази причина бъдещите версии на езика може да не поддържат вече NMI и той няма да се разглежда тук.

В момента JNI е проектиран за интерфейс само към нативни методи писани на C или C++. Чрез JNI вашите нативни методи могат да:

- ◆ Създават, инспектират и осъвременяват Java обекти (включително масиви и **Stringове**)
- ◆ Викат Java методи
- ◆ Хващат и изхвърлят изключения
- ◆ Товарят класове и придобиват информация за класовете
- ◆ Изпълняват определяне на типе по време на изпълнение

Така практически всичко което може да правите с нормалните Java методи може да направите и с нативни методи.

¹ This appendix was contributed by and used with the permission of Andrea Provaglio (www.AndreaProvaglio.com).

Викане на нативен метод

Ще започнем прост пример: Java програма която вика нативен метод, който на свой ред вика Win32 **MessageBox()** API функцията за изобразяване на графична текстова кутия (текстът се рисува - б.пр.). Този пример също после ще се използва с J/Direct. Ако вашата платформа не е Win32, просто заместете C недъра include:

```
| #include <windows.h>
```

```
| C
```

```
| #include <stdio.h>
```

и заместете **MessageBox()** с извикване на **printf()**.

Първата стъпка към писане на Java кода е деклариране на нативния метод и неговите аргументи:

```
class ShowMsgBox {  
    public static void main(String [] args) {  
        ShowMsgBox app = new ShowMsgBox();  
        app.ShowMessage("Generated with JNI");  
    }  
    private native void ShowMessage(String msg);  
    static {  
        System.loadLibrary("MsgImpl");  
    }  
}
```

Декларацията на нативния метод е следвана от **static** блок който вика **System.loadLibrary()** (която бихте могли да извикате по всяко време, но този стил е по-подходящ). **System.loadLibrary()** товари DLL в паметта и се свързва с нея. DLL трябва да бъде на вашия системен път или в директорията съдържаща файла на класа на Java. Разширението на името на файла автоматично се добавя от JVM в зависимост от платформата.

Генератор на С хедъри: javah

Сега компилирайте вашия Java сорсов файл и пуснете **javah** с резултантния **.class** файл. **Javah** беше представен в 1.0, но доколкото използвате JNI на Java 1.1 трябва да посочите ключа **-jni**:

```
| javah -jni ShowMsgBox
```

Javah чете Java класовия файл и за всяка декларация на нативен метод генерира прототип на функция в C или C++ заглавен файл. Ето изхода: сорсовия файл на **ShowMsgBox.h** (леко редактиран за да се събере в книгата):

```
/* DO NOT EDIT THIS FILE  
 - it is machine generated */  
#include <jni.h>  
/* Header for class ShowMsgBox */  
  
#ifndef _Included_ShowMsgBox  
#define _Included_ShowMsgBox  
#ifdef __cplusplus  
extern "C" {  
#endif  
/*  
 * Class: ShowMsgBox  
 */
```

```

 * Method: ShowMessage
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL Java_ShowMsgBox_ShowMessage
(JNIEnv *, jobject, jstring);

#ifndef __cplusplus
}
#endif
#endif

```

Както може да видите от **#ifdef __cplusplus** препроцесорската директива, този файл може да бъде компилиран или от C или от C++ компилатор. Първата директива **#include** включва **jni.h**, хедърфайл който, сред всичко друго което прави, дефинира типовете които може да видите в останалата част на файла. **JNIEXPORT** и **JNICALL** са макроси които се разширяват според платформено- зависими директиви; **JNIEnv**, **jobject** и **jstring** са JNI дефиниции на данни типове.

Сигнатури на функциите; имената

JNI дава конвенция за имената (наречена *name mangling*) за нативните методи; това е важно, понеже е част от механизма по който виртуалната машина свързва Java извикванията с нативните методи. Основно всички нативни методи започват с думите "Java," следвано от името на класа в който се появява нативната декларация на Java, следвано от името на Java метод; за разделител се използва знакът **_**. Ако нативният Java метод е претоварен, тогава сигнатурата на функцията се добавя също; може да видите нативната сигнатура в коментарите, предшестващи прототипите. За повече информация по тези въпроси се отнесете към документацията на JNI.

Прилагане на ваша DLL

В тази точка всичко което остава да се направи е да се напише C или C++ сурс файл който включва генерирания от **javah** хедър файл и реализира нативния метод, да се компилира и да се генерира диномично свързвана библиотека. Тази част е зависима от платформата, аз ще предполагам че знаете как да създадете DLL. Кодът по-долу прилага нативния метод чрез викане на Win32 API. Той после се компилира и свързва във файл наречен **MsgImpl.dll** (от "Message Implementation").

```

#include <windows.h>
#include "ShowMsgBox.h"

BOOL APIENTRY DllMain(HANDLE hModule,
    DWORD dwReason, void** lpReserved) {
    return TRUE;
}

JNIEXPORT void JNICALL Java_ShowMsgBox_ShowMessage(JNIEnv * jEnv,
    jobject this, jstring jMsg) {
    const char * msg;
    msg = (*jEnv)->GetStringUTFChars(jEnv, jMsg, 0);
    MessageBox(HWND_DESKTOP, msg,
        "Thinking in Java: JNI",
        MB_OK | MB_ICONEXCLAMATION);
    (*jEnv)->ReleaseStringUTFChars(jEnv, jMsg, msg);
}

```

Ако не ви е интересен Win32, просто пропуснете викането на `MessageBox()`; интересната част е съпровождащия код. Аргументите които се подават на нативния метод са портал обратно към Java. Първият, от типа `JNIEnv`, съдържа всичките "куки" които позволяват обратно извикване на JVM. (Ще разгледаме това в следващата секция.) Вторият аргумент има различно значение в зависимост от типа на метода. За не-**static** методи както примера по-горе (също наричани *instance methods*), вторият аргумент е еквивалент на "this" указателя в C++ и подобен на `this` в Java: той е позоваване на обекта който вика нативния метод. За **static** това е позоваване на `Class` обект където методът е реализиран.

Останалите аргументи представлят Java обектите подавани на нативния метод при извикването. Примитиви също се подават по този начин, но по стойност.

В следващите секции ще обясним този код чрез разглеждане как да се получи достъп до и да се управлява JVM извънре на нативен метод.

ДОСТЪП ДО JNI ФУНКЦИИ: АРГУМЕНТЪТ JNIEnv

JNI са тези които програмистът използва за взаимодействие с JVM извънре на нативен метод. Както може да се види в примера по-горе, всеки JNI нативен метод приема специален аргумент като пръв: аргументът `JNIEnv`, който е указател към специална JNI даннова структура от типа `JNIEnv_`. Един елемент от JNI данновата структура е указател към масив генериран от JVM; всеки елемент от този масив е указател към JNI функция. JNI функциите могат да бъдат викани от нативен метод чрез тези указатели (по-лесно е, отколкото изглежда). Всяка JVM дава своя си реализация на JNI функции, но техните адреси винаги ще бъдат на известни отмествания.

Чрез `JNIEnv` аргумента програмистът има адрес към множество функции. Те могат да бъдат групирани в следните категории:

- ◆ Намиране информация за версията
- ◆ Изпълнение на класови и обектови операции
- ◆ Поддръжка на локални и глобални отнасяния към Java обекти
- ◆ Достъп до полета на екземпляр и статични полета
- ◆ Викане на методи на екземпляр и статични такива
- ◆ Изпълнение на стрингови и операции с масиви
- ◆ Генерация и обработка на Java изключения

Броят на JNI функциите е доста голям и няма да се разглеждат всичките тук. Вместо това ще покажа какво стои зад тях. За по-големи подробности вижте JNI документацията на вашия компилатор.

Ако погледнете файла `jni.h` ще видите че вътре в `#ifdef __cplusplus` препроцесорната директива `JNIEnv_` структурата се определя като клас компилиран от C++ компилатор. Този клас съдържа много онлайн функции които позволяват достъп до JNI функциите с лесен и познат синтаксис. Например редът в текущия пример

```
| (*jEnv)->ReleaseStringUTFChars(jEnv, jMsg,msg);
```

може да бъде пренаписан на C++ така:

```
| jEnv->ReleaseStringUTFChars(jMsg,msg);
```

Ще забележите че вече не трябва двойно разцепване на **jEnv** указателя, а и същият указател вече не се подава като пръв параметър на извикването на JNI функция. В останалата част от примерите ще използвам C++ стила.

Достап до Java стрингове

Като пример за достъп до JNI функция да видим кода показан по-горе. Тук аргументът **jEnv** от **JNIEnv** е използван за достъп до Java **String**. Java **String**овете са в Unicode формат, така че ако вземете един и го подадете на не-Unicode функция (**printf()**, например), първо трябва да го преобразувате в ASCII знаци с JNI функцията **GetStringUTFChars()**. Тази функция взима Java **String** и го превръща в UTF-8 знаци. (Те са 8 битови за да съдържат ASCII знаци или 16 битови да съдържат Unicode. Ако съдържанието на оригиналния стринг е съставено само от ASCII, резултантният стринг ще бъде също ASCII.)

GetStringUTFChars е името на едно от полетата което **JNIEnv** сочи индиректно, а това е указател към функция. За достъп до JNI функция използваме традиционния С синтаксис за викане на функция по указател. Тази форма се използва за всички JNI функции.

Подаване и използване на Java обекти

В предишния пример подадохме **String** към нативен метод. Може също да подадете Java обекти създадени от вас към нативен метод. Вътре във вашия нативен метод имате достъп до полетата и методите на обекта.

За да се подадат обекти използвайте обикновения Java синтаксис когато декларирате нативния метод. В примера по-долу, **MyJavaClass** има едно **public** поле и един **public** метод. Класът **UseObjects** декларира нативен метод който взема обект от клас **MyJavaClass**. За да видим дали нативният метод манипулира аргумента си, **public** полето на аргумента е сетнато, вика се нативния метод, а после се извежда стойността на **public** полето.

```
class MyJavaClass {  
    public void divByTwo() { aValue /= 2; }  
    public int aValue;  
}  
  
public class UseObjects {  
    public static void main(String [] args) {  
        UseObjects app = new UseObjects();  
        MyJavaClass anObj = new MyJavaClass();  
        anObj.aValue = 2;  
        app.changeObject(anObj);  
        System.out.println("Java: " + anObj.aValue);  
    }  
    private native void  
    changeObject(MyJavaClass obj);  
    static {  
        System.loadLibrary("UseObjImpl");  
    }  
}
```

След компилиране на кода и даване на **.class** файла на **javah** може да приложите нативния метод. В примера по-долу щом се сдобиете с ID на полето и метода, те са достъпни чрез JNI функции.

```
JNIEXPORT void JNICALL  
Java_UseObjects_changeObject(  
    JNIEnv * env, jobject jThis, jobject obj) {
```

```

jclass cls;
jfieldID fid;
jmethodID mid;
int value;
cls = env->GetObjectClass(obj);
fid = env->GetFieldID(cls,
    "aValue", "I");
mid = env->GetMethodID(cls,
    "divByTwo", "()V");
value = env->GetIntField(obj, fid);
printf("Native: %d\n", value);
env->SetIntField(obj, fid, 6);
env->CallVoidMethod(obj, mid);
value = env->GetIntField(obj, fid);
printf("Native: %d\n", value);
}

```

Първият аргумент, C++ функцията получава **object**, който е нативната страна на Java обектов манипулатор който сме подали чрез Java код. Просто четем **aValue**, извеждаме го, променяме стойността, викаме метода **divByTwo()** на обекта, а после извеждаме отново.

За достъп до поле или метод първо трябва да се сдобиете с идентификатора му. Подходящи JNI функции дават класа на обекта, името на элемента и сигнатурата. Тези функции връщат идентификатор с който имате достъп до елемента. Този подход може да ви се стори засушен, но вашият нативен метод няма знания за вътрешната структура на Java обекта. Вместо това той трябва да извърши достъпа по отмествания, дадени от JVM. Това позволява различни JVMи да реализират по различен начин вътрешно обектите без да засягат вашите нативни методи.

Ако пуснете Java програмата ще видите че обектът подаден чрез Java страната се манипулира от вашия нативен метод. Но какво точно се подава? Указател или Java манипулатор? И какво прави боклучарят по време на извикването на нативни методи?

Боклучарят продължава да работи по време на изпълнението на нативни методи, но се гарантира че вашият обект няма да бъде почищен по време на работата на нативен метод. За да се осигури това, създават се локални отнасяния (вероятно: псевдоними - б.пр.) преди и се разрушават точно в края на извикването на нативния метод. Понеже тяхното време на живот обгръща (е по-голямо от това на извикването - б.пр.) извикването, знаете че обектът е валиден по време на извикването на нативния метод.

Понеже тези отнасяния се създават и разрушават всеки път когато се вика нативната функция, не може да си направите локални копия на вашите нативни методи, в **static** променливи. Ако искате отнасяне което трае много извиквания на функции, трябва ви глобално отнасяне. Глобални отнасяния не се правят от JVM, но програмистът може да направи локалното глобално чрез викане на определени JNI функции. Когато създадете глобално отнасяне, вие ставате отговорни за времето на живот на обекта към който е отнасянето. Глобалното отнасяне (и обекта към което е то) ще бъде в паметта докато програмистът явно освободи отнасянето чрез съответната JNI функция. Подобно на **malloc()** и **free()** в C.

JNI и изключенията в Java

При JNI могат да се изхвърлят Java изключения, да се хващат, извеждат и преизхвърлят точно както в Java програма. Програмистът обаче трябва ада извика съответните JNI функции да се оправят с изключенията. Ето JNI функциите за поддръжка на изключения:

- ◆ **Throw()**

Изхвърля съществуващ обект на изключение. Използва се в нативните методи за преизхвърляне на изключение.

- ◆ **ThrowNew()**
Генерира нов обект на изключение и го изхвърля.
- ◆ **ExceptionOccurred()**
Определя дали е изхвърлено изключение и още не е изчистено.
- ◆ **ExceptionDescribe()**
Извежда изключението и трасировката на стека.
- ◆ **ExceptionClear()**
Изчиства чакащо изключение.
- ◆ **FatalError()**
Дава фатална грешка. Не се връща.

Между тези не може да игнорирате **ExceptionOccurred()** и **ExceptionClear()**. Повечето JNI функции могат да генерират изключения, а няма в езика нищо което може да използвате вместо try блока на Java, така че трябва да извикате **ExceptionOccurred()** след всяко викане на JNI функция за да видите дали е било изхвърлено изключение. Ако констатирате изключение, може да изберете да го обработите (и възможно - да го преизхвърлите). Трябва да осигурите, обаче, че изключението в края на краищата е изчистено. Това може да се направи във вашата функция чрез **ExceptionClear()** или в някаква друга функция ако изключението е преизхвърлено, но трябва да се направи.

Трябва да осигурите изчистването на изключението, понеже иначе резултатите биха били непредсказуеми ако извикате JNI функция докато има чакащо изключение. Има малко JNI функции които са безопасни при извикване по време на изключение; в това число са, разбира се, тези за обработка на изключения.

JNI и тредингът

Понеже Java е език с многонишковост, няколко нишки могат да викат нативен метод конкурентно. (Нативният метод може да бъде прекъснат по средата на изпълнението си когато го извика друга нишка.) Изцяло програмистът си отговаря дали нативните му методи са безопасни при многонишковост, т.е. да не променят споделяни данни по нежелан начин. Основно имате две възможности: да декларирайте нативния метод като **synchronized** или да реализирате някаква друга стратегия в нативния метод за осигуряване на коректна, конкурентна работа.

Също, никога няма да давате **JNIEnv** указател през нишките, понеже вътрешната структура към която сочи той е изградена на база нишка и нейните данни имат смисъл само за тази нишка.

Използване на съществуващ код

Най-лекият начин да се реализират JNI нативни методи е да се започне с писането на прототипи в Java клас, той да се компилира, да се пусне **.class** файла през **javah**. Но какво ще стане ако има много код от преди, който трябва да се пусне от Java? Преименуването на всички функции от вашите DLLи за да бъдат съгласувани с конвенцията за имената в JNI не е жизнеспособно решение. Най-добрият начин е да напишете обгръщаща DLL "отвън" на вашия стар код. Java кодът вика функции от тази нова DLL, които на свой ред викат оригиналните DLL функции. Това решение не е само заобиколка; в повечето случаи така или иначе трябва да го правите, понеже трябва да викате JNI функции по относянето към обекта преди да ги използвате.

Пътят на Microsoft

Microsoft не поддържа JNI, но дава подходяща поддръжка за викане на не-Java код. Тази поддръжка е вградена в компилатора, майкрософтската JVM и допълнителните инструменти. Може да намерите въведение към Microsoft Java технологии в Appendix A на първата онлайн редакция на *Thinking in Java* (която също беше написана от Andrea Provaglio). Тази книга е достъпна безплатно от <http://www.BruceEckel.com>. Нещата описани в нея секция ще станат само ако използвате Microsoft Java компилатор и ги пускате на Microsoft Java Virtual Machine. Ако смятате да разпространявате своето приложение чрез Internet, или вашият Intranet е построен върху различни платформи, това може да бъде сериозен въпрос.

Чертите дадени в него приложение включват:

1. **J/Direct:** Начин лесно да се викат Win32 DLL функции, с някои ограничения.
2. **Raw Native Interface (RNI):** Може да викате Win32 DLL функции, но трябва после да съберете боклука.
3. **Java/COM integration:** Как да дадете или използвате COM услуги направо от Java. Това също включва въведение в основите на COM.

В: Насоки за програмиране на Java

Това приложение съдържа съвети които да ви помогат когато изпълнявате проектирането на програма на ниско нива, а също и когато пишете кода.

1. Първата буква на имената на класовете да е голяма. Първата буква на полетата, методите и обектите (манипулаторите) трябва да е малка. На всички идентификатори думите в имената да са долепени, да се различават с първа голяма буква новите думи. Например: **ThisIsAClassName** и **thisIsAMethodOrFieldName**. Всичките букви на **static**, **final** идентификатори на примитиви които имат статични инициализатори да са големи. Това показва че са константи по време на компилация. Па-кетите са специален случай: те са само с малки букви, даже и междуинните думи. Разширението на домейна (.com, .org, .net, .edu и т.н.) също да е с малки букви. (Това беше промяна между Java 1.1 и Java 2.)
2. Когато създавате клас за обща употреба, следвайте "канонична форма" и включете дефиниции за **equals()**, **hashCode()**, **toString()**, **clone()** (приложете **Cloneable**) и приложете **Serializable**.
3. За всеки клас който създавате имайте пред вид включване на **main()** която да съдържа код за тестване на класа. Не е необходимо да махате тестовия код за използване на класа в проект и ако направите промени може лесно да повторите тестването. Този код също дава примери как да се използва вашия клас.
4. Методите ще се държат във вид на стегнати единици които изпълняват цялостна част от интерфейс. В идеалния случай методите трябва да бъдат сбити; ако са дълги ще търсите начин да ги разделите на по-кратки. Това също ще поощри повторното използване на вашия клас. (Понякога методите се налага да бъдат големи, но трябва да правят пак само едно нещо.)
5. Когато проектирате клас, мислете за перспективата на клиент-програмист (класът трябва да е очевиден за използване) и за перспективата на човека който ще поддържа кода (противопоставете нещата които ще се променят и леснотата на промяната).
6. Мъчете се да поддържате класовете малки и фокусирани. Може да се покелае препроектиране на клас когато:
 - 1) Има сложен ключов оператор: вижте използване на полиморфизъм
 - 2) голям брой методи които вършат най-различни операции: вижте

използване на няколко класа

3) Голям брой полета които са за много различни характеристики: вижте използване на няколко класа

7. Дръжте нещата "колкото е възможно по-**private**." Щом публикувате аспект на вашата библиотека (метод, клас, поле), никога не може да го мањнете. Ако го мањнете, скапвайте нечий съществуващ код, карайки този някой да пренаписва и препроектира. Ако публикувате само необходимото, може да промените всичко друго с лекота, а понеже проектите имат тенденция да се развиват това е важна свобода. Частността е важно нещо особено във връзка с многонишковостта – само **private** полета могат да бъдат предпазени от не-**synchronized** употреба.
8. Пазете се от "синдрома на гигантския обект." Това често е залитане на процедурни програмисти които са нови в ООП и които завършват с написването на процедурна програма и то сложена в един или два огромни обекта. С изключение на рамките за приложения, обектите представят концепциите във вашето приложение, не приложението.
9. Ако трябва да направите нещо грубо, най-малкото локализирайте го в клас.
10. Всеки път когато забележите класове да са тясно свързани един с друг, вижте възможността за използване на вътрешни класове.
11. Използвайте коментари либерално, използвайте **javadoc** синтаксиса за създаване на програмната си документация.
12. Избягвайте използването на "магически числа," които са числа твърдо кодирани в програмата. Те са кошмар ако трябва да ги промените, понеже никога не знаете дали "100" значи "дължината на масива" или "нещо съвсем друго." Вместо това създайте константа с говорящо име и използвайте идентификатора в програмата. Това прави програмата по-лесна за разбиране и много по-лесна за поддържане.
13. В термините на конструктори и изключения изобщо ще искате да преизхвърлите всяко изключение което сте хванали докато сте в конструктор ако то причинява неправилно създаване на обекта, така че извикващият да не продължи сляпо, смятайки че обектът е създаден коректно.
14. Ако вашият клас се нуждае от някакво заключително действие когато клиент-програмистът свърши с използването му, сложете съответния код в един, добре дефиниран метод с име като **cleanup()** което ясно показва предназначението. Освен това сложете **boolean** флаг в класа да показва дали обектът е създаден правилно или не. В метода **finalize()** за класа проверете за да сте сигурни че обектът е финализиран добре и изхвърлете клас извлечен от **RuntimeException** ако не е, за индикация на програмна грешка. Преди да разчитате на такава схема, осигурете че **finalize()** работи на вашата система. (Може да трябва да извикате **System.runFinalizersOnExit(true)** за осигуряване на това поведение.)
15. Ако обектът трябва да се почисти (по друг начин от събирането на боклука) в конкретен обхват, използвайте следния подход: Инициализирайте обекта и, ако е успешно, веднага влезте в **try** блок с **finally** клауза която изпълнява почистването.
16. Когато подтискате **finalize()** по време на наследяване, не пропускайте да извикате **super.finalize()** (това не е необходимо ако **Object** е непод-

средствения суперклас). Трябва да извикате `super.finalize()` като последен акт във вашия подтиснат `finalize()` а не като пръв, за да осигурите че компонентите на базовия клас са още валидни ако ви трябват.

17. Когато създавате колекция от обекти с фиксирана дължина, сложете ги в масив (особено ако връщате тази колекция от метод). По този начин получавате ползата от проверките на масиви по време на компилация и приемникът на масива може и да не трябва да каства елементите на масива за да ги използва.
18. Изберете **interface** пред **abstract** класове. Ако знаете че нещо ще става базов клас, първо ще пробвате да го направите **interface**, а само ако се наложи да сложите дефиниции на методи или член-променливи ще го превърнете в **abstract** клас. **interface** казва какво клиентът иска да прави, докато класът има тенденция да се фокусира върху (или да позволи) детайли на реализацијата.
19. Вътрешните конструкторите правете само необходимото обектът да дойде в нужното състояние. Активно избегвайте викането на други методи (освен **final** методи) понеже тези методи биха могли да бъдат подтиснати от някой друг за да предизвикат неочаквани резултати при построяването. (Виж глава 7 за детайли.)
20. Обектите няма просто да съдържат данни, те ще имат добре определено поведение също.
21. Първо опитайте с композиция когато създавате нови класове от съществуващи класове. Ще използвате наследяване само ако се налага от проекта. Ако използвате наследяване където композицията ще работи, вашият дизайн ще стане ненужно сложен.
22. Използвайте наследяване и подтискане на методи за модификация на поведението и полета за изразяване на промени в състоянието. Екстремален пример какво да не се прави е наследяването на различни класове за представяне на цветове вместо използване на поле "цвят".
23. За да избегнете много горчив опит осигурете да има един клас за всяко име на вашия път към класовете. Иначе компилаторът може да намери другия клас с идентично име първо и да издаде съобщения за грешка които нямат смисъл. Ако подозирате че имате подобен проблем, опитайте се да погледнете за **.class** файлове с еднакви имена по вашия път към класовете.
24. Когато използвате "адаптерите" на събития в Java 1.1 AWT, има практически лесен капан в който може да паднете. Ако подтиснете един от адаптерните методи и не напишете името точно, на практика ще напишете друг метод заместо да подтиснете този. Обаче това е напълно законно, така че няма да получите никакво съобщение за грешка от компилатора или системата – вашият код просто няма да работи коректно.
25. Използвайте шаблони за проектиране за да избегнете "разголената функционалност." Тоест, ако ще се създава само един обект от вашия клас, направете коментар "Направете само един обект." Обградете го в синглетон. Ако имате много и объркан код във вашата главна програма който създава обектите, огледайте се за шаблон като фабриката в който да капсулирате създаването. Елиминирайки "разголената функционалност" не само ще направите вашия код по-лесен за

разбиране и поддръжка, това ще го направи също и по-устойчив на идващите след вас хора по поддръжката.

26. Внимавайте за "аналитична парализа." Запомнете че обикновено трябва да си придвижите проекта преди да знаете всичко, а най-добрият начин да видите нещата е да ги пробвате като се придвижите напред с проекта, не да си ги рисувате мислено.
27. Внимавайте за преждевременна оптимизация. Първо го направете да работи, после да е бързо – но само ако трябва и ако доказано има тясно място в конкретно парче код. Докато не сте използвали профайлър за констатиране на тясното място вероятно ще пропилеете времето си. Скритата цена на засуканите оптимизации е че вашият код става по-трудно четим и се поддържа по-трудно.
28. Помните че код се чете повече отколкото се пише. Чистото проектиране води до лесни за четене програми, но коментарите, детайлните обяснения и примерите са незаменими. Те ще помогнат на вас и на всеки който изва след вас. Ако не друго, разочарованията при търсенията из документацията на Java ще ви убедят в това.
29. Когато мислите че сте постигнали добър анализ, дизайн или реализация, разгледайте ги основно. Вземете някой извън вашата група – не е необходимо да бъде консултант, но може да бъде от някоя друга група в компанията ви. Разглеждането на работите ви от две непредубедени очи може да изведи наяве проблеми още докато те са лесни за овладяване и многоократно изплаща парите и времето "загубени" в този процес.
30. Елегантността винаги се изплаща. Може да изглежда че твърде много време минава докато се получи добро решение на проблема, но когато то работи добре и лесно се приспособява към нови ситуации без да изисква, дни, или месеци напрегната работа, ще видите наградите (даже и никой да не може да ги измери). И няма нищо което да се сравни с чувството когато получите забележителен дизайн и знаете това. Борете се с напора да бързате; поддаването само би довело до забавяне.
31. Може да намерите още съвети в Web. Доста връзки може да се намерят на
<http://www.ulb.ac.be/esp/ip-Links/Java/joodcs/mm-WebBiblio.html>

C: Препоръчва се за четене

(Това не се превежда, понеже ако някой може да прочете книгите, той може да го прочете и непреведено - б.пр.)

Java in a Nutshell: A Desktop Quick Reference, 2nd Edition, by David Flanagan, O'Reilly & Assoc. 1997. A compact summary of the online documentation of Java 1.1. Personally, I prefer to browse the docs online, especially since they change so often. However, many folks still like printed documentation and this fits the bill; it also provides more discussion than the online documents.

The Java Class Libraries: An Annotated Reference, by Patrick Chan and Rosanna Lee, Addison-Wesley 1997. What the online reference *should* have been: enough description to make it usable. One of the technical reviewers for *Thinking in Java* said, "If I had only one Java book, this would be it (well, in addition to yours, of course)." I'm not as thrilled with it as he is. It's big, it's expensive, and the quality of the examples doesn't satisfy me. *But it's a place to look when you're stuck and it seems to have more depth (and sheer size) than Java in a Nutshell.*

Java Network Programming, by Elliott Rusty Harold, O'Reilly 1997. I didn't begin to understand Java networking until I found this book. I also find his Web site, Café au Lait, to be a stimulating, opinionated, and up-to-date perspective on Java developments, unencumbered by allegiances to any vendors. His almost daily updating keeps up with fast-changing news about Java. See <http://sunsite.unc.edu/javafaq/>.

Core Java, 3rd Edition, by Cornell & Horstmann, Prentice-Hall 1997. A good place to go for questions you can't find the answers to in *Thinking in Java*. Note: the Java 1.1 revision is *Core Java 1.1 Volume 1 – Fundamentals & Core Java 1.1 Volume 2 – Advanced Features*.

JDBC Database Access with Java, by Hamilton, Cattell & Fisher (Addison-Wesley, 1997). If you know nothing about SQL and databases, this is a nice, gentle introduction. It also contains some of the details as well as an "annotated reference" to the API (again, what the online reference should have been). The drawback, as with all books in The Java Series ("The ONLY Books Authorized by JavaSoft") is that it's been whitewashed so that it says only wonderful things about Java – you won't find out about any dark corners in this series.

Java Programming with CORBA Andreas Vogel & Keith Duddy (John Wiley & Sons, 1997). A serious treatment of the subject with code examples for the three main Java ORBs (Visibroker, Orbix, Joe).

Design Patterns, by Gamma, Helm, Johnson & Vlissides (Addison-Wesley 1995). The seminal book that started the patterns movement in programming.

UML Toolkit, by Hans-Erik Eriksson & Magnus Penker, (John Wiley & Sons, 1997). Explains UML and how to use it, and has a case study in Java. An accompanying CD-ROM contains the Java code and a cut-down version of Rational Rose. An excellent introduction to UML and how to use it to build a real system.

Practical Algorithms for Programmers, by Binstock & Rex (Addison-Wesley 1995). The algorithms are in C, so they're fairly easy to translate into Java. Each algorithm is thoroughly explained

Индекс

Моля забележете че някои имена може да се дублират с големи букви. Следвайки стила на Java, имената с големи букви се отнасят за Java класове, докато тези с малки - за общата концепция.

Е, драги Читателю, моята работа свършва тук: индексът ще се получи автоматично, а от глава 5-та до края аз няма да преработвам полетата, както и няма да превеждам бележките под линия, поради слабия интерес към книгата. Оценявайки високо труда ти да стигнеш до тук, както и може би хитростта или любопитството ти да погледнеш по-към края, вярвам, че си се убедил вече, че ако искаш да правиш нещо повече от учене на езика Паскал по древни учебници трябва да научиш английски език и се надявам този превод да ти помогне и за това.

От преводача.

abstract.....
class.....234
abstract keyword.....235
Abstract Window Toolkit (AWT).....527
AbstractButton.....658
AbstractSequentialList.....332
AbstractSet.....318
accept().....748
access.....
specifiers.....182
action().....535, 557
cannot combine with listeners.....609
ActionEvent.....577, 578, 599, 650
actionPerformed().....580
add().....551
addActionListener().....647, 710
addElement(), Vector.....292
addListener.....575
addTab().....673
Adler32.....429
aliasing.....93
during a method call.....485
align.....532
AlreadyBoundException.....768
AND.....
логическа (&&).....98
anonymous inner class.....723, 863
appendText().....543
applet.....529
advantages for client/server systems563
combined applets and applications582
packaging applets in a JAR file to optimize loading.....585
Applet.....551
appletviewer.....532
application.....
application framework.....262
application framework.....529

archive tag, for HTML and JAR files585
array.....
first-class objects.....286
ArrayList.....319, 323
Arrays.....341
Arrays.asList().....341
assignment.....91
associative array.....291, 303
automatic compilation.....177
automatic type conversion.....196
available().....407
bag.....316
base class.....186, 198
base-class interface.....230
BasicArrowButton.....658
beanbox Bean testing tool.....651
Beans.....
and multithreading.....707
Bill Joy.....97
binarySearch().....342
binding.....
late binding.....227
BitSet.....300
bitwise.....
AND.....107
operators.....101
bitwise copy.....493
blank final, in Java 1.1.....215
blocking.....
and available().....408
and threads.....711
on IO.....718
Booch, Grady.....834
Boolean.....118
Boolean.....
and casting.....108
vs. C and C++.....99
BorderLayout.....552, 584

Borland.....681
 Borland.....
 Delphi.....639
 bound properties.....652
 BoxLayout.....674
 break ключова дума.....123
 BufferedInputStream.....393, 406
 BufferedOutputStream.....396, 408
 BufferedReader.....377, 420
 BufferedWriter.....420
 business objects/logic.....606
 button.....534, 551
 creating your own.....559
 ButtonGroup.....659
 ByteArrayInputStream.....385
 ByteArrayOutputStream.....388
 C/C++, interfacing with.....867
 C++.....23, 97
 copy constructor.....507
 STL.....315
 template.....853
 vector class, vs. array and Java Vector.....285
 callback.....310, 398
 Canvas.....559, 737
 CardLayout.....547, 672
 case оператор.....128
 cast.....141
 cast.....
 оператори.....107
 catch.....
 catching an exception.....355
 catching any exception.....359
 keyword.....356
 CD ROM за книгата.....15
 change.....
 vector of change.....265
 CharArrayReader.....419
 CharArrayWriter.....419
 Checkbox.....542, 543
 Java 1.1.....591
 CheckboxGroup.....543
 CheckboxMenuItem.....566
 Java 1.1.....596
 CheckedInputStream.....427
 CheckedOutputStream.....427
 Checksum.....429
 Choice.....
 Java 1.1.....593
 class.....188
 access.....188
 browser.....188
 class literal.....468, 471
 inner classes and overriding.....260
 read-only classes.....511
 инициализиране на членове в точката на деклариране ..156
 Class.....660
 Class object.....449, 466, 703
 reflection.....840
 ClassCastException.....282, 469
 classpath.....92, 175, 533
 cleanup.....

with finally.....373
 client, network.....746
 clone().....490, 837
 clone().....
 and inheritance.....501
 Cloneable interface.....491
 CloneNotSupportedException.....493
 close().....407
 codebase.....532
 collection.....
 class.....284, 291
 of primitives.....289
 Collection.....316
 Collections.....342, 345
 comma operator.....105
 common pitfalls when using operators.....106
 Common-Gateway Interface (CGI).....797
 Comparable.....329, 344
 Comparator.....329, 343
 compare().....343
 compareTo().....344
 compile-time constant.....212
 Component.....551, 557, 673
 ComponentAdapter.....580
 composition.....194
 and cloning.....496
 and design patterns.....818
 choosing composition vs. inheritance.....207
 combining composition & inheritance.....202
 vs. inheritance.....212
 ConcurrentModificationException.....349
 const, in C++.....516
 constant.....
 implicit constants, and String.....515
 constrained properties.....652
 constructor.....
 and exception handling.....376
 and finally.....376
 base-class constructors and exceptions.....202
 Constructor.....660
 for reflection.....477, 840
 consume().....589
 Container.....551
 container class.....284
 ContainerAdapter.....580
 control framework, and inner classes.....262
 CORBA.....771
 couplet.....690, 861
 coupling.....357
 CRC32.....429
 createStatement().....790
 creational design patterns.....820, 835
 critical section, and synchronized block.....706
 daemon threads.....697
 database.....
 Java DataBase Connectivity (JDBC).....788
 URL.....788
 DatabaseMetaData.....797
 DataFlavor.....633
 Datagram.....758
 DatagramPacket.....759, 762

DatagramSocket.....	759
DataInput.....	397
DataInputStream	392, 406, 407, 408, 420
DataOutput.....	397
DataOutputStream.....	395, 408, 420
dead, Thread.....	712
deadlock, multithreading.....	721
decorator design pattern.....	391
deep copy.....	490
and Vector.....	497
default package.....	184
default ключова дума, в switch оператор	128
DefaultMutableTreeNode.....	669
defaultReadObject().....	445
DefaultTreeModel.....	670
defaultWriteObject().....	445
DeflaterOutputStream.....	427
design.....	
and inheritance.....	276
design patterns.....	191, 817
behavioral.....	821
structural.....	820
destroy().....	530, 724
destructor.....	
Java doesn't have one.....	203
development, incremental.....	209
Dialog.....	570
Java 1.1.....	600
Dictionary.....	303
directory.....	
lister.....	397
dispatchEvent().....	599
dispose().....	570, 628, 630
do-while.....	121
Domain Name Service (DNS).....	745
dotted quad.....	745
double dispatching.....	849
double, маркер за стойност на литерал (D)	109
Double.valueOf().....	842
downcast.....	211, 280
type-safe downcast in run-time type identification	469
dynamic.....	
behavior change with composition	277
binding.....	223, 227
early binding.....	227
East.....	552
efficiency.....	
and arrays.....	285
elementAt(), Vector.....	292, 296
enableEvents().....	618
encapsulation.....	187
end().....	628
Enumeration.....	319
equals().....	98, 328, 536
equals().....	
overriding for HashMap.....	308
vs. ==.....	417
event.....	
Event object, AWT.....	535
event-driven system.....	263
Java Beans.....	639
target.....	535
unicast.....	606
EventSetDescriptors.....	645
exception.....	
Error class.....	363
Exception class.....	363
exception handler.....	356
exception matching.....	380
handler.....	353
handling.....	204
restrictions.....	369
specification.....	358
termination vs. resumption.....	357
Throwable.....	359
typical uses of exceptions.....	381
exceptional condition.....	354
exceptions.....	
and JNI	874
executeQuery().....	790
extends.....	186, 199, 279
extends.....	
and interface.....	243
Externalizable.....	438
alternative approach to using.....	443
factory method.....	835
fail fast collections.....	349
FeatureDescriptor.....	652
Field, for reflection.....	477
fields, initializing fields in interfaces	245
file.....	
data file output shorthand.....	411
formatted file output shorthand.....	410
input shorthand.....	410
File.....	385, 420, 460
File.....	
class.....	397
File.list().....	397
File Transfer Protocol (FTP).....	533
FileDescriptor.....	385
FileInputStream.....	385, 406
FilenameFilter.....	397, 458
FileNotFoundException.....	378
FileOutputStream.....	388, 408
FileReader.....	377, 419
FileWriter.....	419
fillInStackTrace().....	361
FilterInputStream.....	387
FilterOutputStream.....	390
FilterReader.....	419
FilterWriter.....	419
final.....	238
argument.....	400
classes.....	217
data.....	212
keyword.....	212
method.....	227
methods.....	216
static primitives.....	214
with object handles.....	213
finalize().....	146, 206, 379
finalize().....	

and inheritance.....271
runFinalizersOnExit().....150
директно викане.....148
finally.....204, 205
keyword.....372
pitfall.....375
flat-file database.....792
float, маркер за стойност на литерал (F)
.....109
FlowLayout.....551
Focus traversal.....623
FocusAdapter.....580
Font.....628
for ключова дума.....121
forName().....467, 680
FORTRAN.....109
Frame.....551, 566
friendly.....248
friendly
and interface.....238
functor.....398
garbage collection.....
forcing finalization.....206
generic.....295
get(), HashMap.....306
getAddress().....762
getAppletContext().....537
getBeanInfo().....643
getClass().....359, 474
getConstructor().....660
getConstructor(), reflection.....840
getConstructors().....479
reflection.....840
getContents().....632
getDirectory().....575
getEventSetDescriptors().....645
getFile().....575
getFloat().....790
getInputStream().....748
getInt().....790
getInterfaces().....475
getMethodDescriptors().....645
getMethods().....479
getModel().....670
getName().....476, 645
getOutputStream().....748
getPort().....762
getPrintJob().....629
getPriority().....725
getProperties().....309
getPropertyDescriptors().....645
getPropertyType().....645
getReadMethod().....645
getSelectedItems().....546
getState().....569
getString().....790
getSuperclass().....475
getText().....541
getTransferData().....633

getTransferDataFlavors().....633
getWriteMethod().....645
gotFocus().....558
goto.....
липсва в Java.....124
graphical user interface (GUI).....527
graphics.....
Graphics object.....628
GridLayout.....553
GridLayout.....553, 737
guarded region, in exception handling.....356
GUI.....
builders.....528
GZIPInputStream.....427
GZIPOutputStream.....427
handle.....
finding exact type of a base handle.....466
handleEvent().....536, 557
has-a relationship, composition.....208
hashCode().....305, 326
hashCode().....
overriding for HashMap.....308
HashMap.....309, 329
HashSet.....326
Hashtable.....316, 349, 415, 561
Hashtable.....
used with Vector.....457
hasNext().....319
hasNext(), Iterator.....297
Hexadecimal.....108
HTML.....797
HTML.....
name.....696
param.....696
value.....696
Icon.....660
idltojava.....774
if-else оператор.....105, 119
IllegalMonitorStateException.....717
ImageIcon.....660
immutable objects.....511
implementation.....
and interface.....207
hiding.....187, 248
implements keyword.....238
import ключова дума.....172
indexed property.....652
indexOf().....842
indexOf().....
String.....479
InflaterInputStream.....427
inheritance.....186, 194, 198
from inner classes.....259
init().....530, 586
initialization.....
and class loading.....218
base class.....200
inline method calls.....216
inner class.....
and upcasting.....247
in methods & scopes.....249

referring to the outer class object	258
InputStream	384, 751
InputStreamReader	418, 419, 751
insertNodeInto()	670
instance	
instance initialization in Java 1.1	254
instanceof	
dynamic instanceof	473
Integer	
parseInt()	573
interface	
and implementation, separation	187
and inheritance	242
Cloneable interface used as a flag	491
keyword	237
upcasting to an interface	240
vs. abstract	242
interfacing with hardware devices	867
Internet	
Internet Protocol	745
Internet Service Provider (ISP)	533
InterruptedException	685
Intranet	564
Introspector	643
IO	
deprecated	383
Java 1.1 compression library	426
library	383
is-a	278
is-a	
relationship, inheritance	208
is-like-a	279
isDaemon()	697
isDataFlavorSupported()	633
isFocusTraversable()	623
isInstance	473
isInterface()	476
ItemEvent	591
ItemListener	591
iterator	296, 819
Iterator	296, 309, 314, 319, 833
iterator()	319
JAR	650
JAR	
file	173, 563
files	612
jar files and classpath	176
Java	
and pointers	484
and set-top boxes	101
capitalization style source-code checking tool	452
crashing Java	299
версий	17
Java 1.018, 77, 102, 150, 166, 206, 288, 299, 301, 302, 310, 383, 384, 403, 407, 412, 414, 418, 419, 420, 458, 470, 473, 527, 535, 539, 589, 594, 599, 609, 641, 680, 751, 761, 798, 819	
Java 1.112, 18, 44, 50, 52, 67, 69, 77, 87, 102, 150, 151, 162, 166, 173, 206, 215, 246, 254, 262, 300, 302, 376, 378, 383, 401, 404, 407, 412, 414, 418, 419, 420, 423, 425, 426, 427, 429, 433, 434, 437, 461, 462, 463, 468, 471, 473, 474, 476, 477, 499, 528, 535, 539, 552, 557, 559, 563, 583, 584, 589, 593, 594, 596, 599, 609,	
612, 613, 618, 623, 627, 628, 630, 641, 642, 681, 718, 751, 760, 764, 788, 797, 798, 840, 841, 877, 880, 882	
Java 1.1	
and Swing	653
IO streams	417
JAR utility	431
new array initialization syntax	288
reflection	477, 838
String behavior	298
Java 218, 77, 175, 299, 313, 401, 402, 528, 680, 712, 716, 719, 721, 735, 819, 877	
Java 2	
new collections library	315
Swing library	653
Java Beans	
see Beans	638
Java Foundation Classes (JFC/Swing)	528
Java operators	90
Java Virtual Machine	466
javah	869
JButton	658
JCheckbox	660
JColorChooser	674
JComboBox	666
JComponent	656
JFileChooser	674
JFrame	656
JHTMLPane	674
JInternalFrame	674
JIT	
Just-In Time compilers	62
JLabel	657
JLabels	665
JLayeredPane	674
JList	666
JMenu	665
JMenuItem	660, 666
JMenuItems	665
JNICALL	869
JNIEnv	871
JNIEXPORT	869
join	792
JOptionPane	548
JPasswordField	674
JPopupMenu	665
JProgressBar	668
JRadioButton	660
JScrollPane	654, 667, 670
JSlider	668
JTabbedPane	672
JTextField	666
JTextPane	674
JToggleButton	658
JToolbar	674
JTree	668, 670
KeyAdapter	580
keyDown()	557
keySet()	338
keyUp()	557
Label	540

layout.....	558
controlling layout.....	551
layout manager.....	551
lazy evaluation.....	315
length, for arrays.....	286
lightweight persistence.....	433
LineNumberInputStream.....	393, 408
LineNumberReader.....	420
LinkedList.....	323
list.....	323
boxes.....	546
List.....	285, 291, 316, 323, 546
Java 1.1.....	594
sorting.....	345
list box.....	666
listener adapters.....	581
ListIterator.....	323
local loopback IP address.....	747
localhost.....	747
and RMI.....	767
logical.....	323
operators.....	98
lostFocus().....	557
lvalue.....	91
main().....	199
manifest file, for JAR files.....	431
map.....	303
Map.....	285, 291, 303, 316, 329
mark().....	397
Math.random().....	305
Math.random().....	305
стойности давани от.....	130
mathematical operators.....	94
max().....	346
MDI.....	674
member initializers.....	270
Menu.....	566
Java 1.1.....	596
Swing.....	661
menu shortcuts.....	599
MenuBar.....	566, 599
MenuComponent.....	566
MenuItem.....	566, 599
MenuItem.....	566
Java 1.1.....	596
meta-class.....	466
method.....	466
lookup tool.....	676
method call binding.....	226
protected methods.....	209
Method.....	645
Method.....	645
for reflection.....	477
MethodDescriptors.....	645
Microsoft.....	681
min().....	346
mistakes, and design.....	192
mkdirs().....	403
monitor, for multithreading.....	703
MouseAdapter.....	581
mouseDown().....	558, 572
mouseDrag().....	558
mouseEnter().....	558
mouseExit().....	558
MouseMotionAdapter.....	581
mouseMove().....	558
mouseUp().....	558
multi-tiered systems.....	606
multicast.....	650
multicast.....	650
multicast events.....	606
multiple dispatching.....	849
multiple inheritance, in C++ and Java.....	240
multiple-document interface.....	674
multitasking.....	683
multithreading.....	348
and collections.....	348
drawbacks.....	741
name.....	532
name.....	532
spaces.....	172
Naming.....	767
bind().....	767
rebind().....	768
unbind().....	768
native method interface (NMI) in Java 1.0.....	867
natural comparison method.....	344
network programming.....	754
dedicated connection.....	754
serving multiple clients.....	754
testing programs without a network	747
new оператор.....	146
и примитиви, масив.....	164
newInstance().....	660, 673
newInstance().....	660, 673
reflection.....	475
newInstance(), reflection.....	840
next().....	319
next(), Iterator.....	297
nextToken().....	459
no-arg.....	459
constructors.....	137
North.....	552
notifyListeners().....	711
notifyObservers().....	828, 830
null.....	69, 287
NullPointerException.....	364
object.....	433
serialization.....	433
Object.....	285, 305, 833
Object.....	285, 305, 833
clone().....	494
ObjectOutputStream.....	434
Observable.....	828
Observer.....	828
ODBC.....	789
OMG.....	771
OOP.....	188
optional methods, in the Java 2 collections.....	340
OR.....	98
(11).....	98
order.....	98

of constructor calls with inheritance	269
OutputStream	384, 387, 751
OutputStreamWriter	418, 419, 751
overloading	206
lack of name hiding during inheritance	
operator overloading for String	516
overriding	
vs. overloading	233
package	833
package	
access, and friendly	182
and applets	534
creating unique package names	174
оператор и поддиректории за главите на книгата	
paint()	559, 572, 623, 629
parameterized type	295
pass	
pass by value	488
persistence	446
PipedInputStream	386, 412
PipedOutputStream	386, 389, 412
PipedReader	419
PipedWriter	419
polymorphism	223, 865
polymorphism	
behavior of polymorphic methods inside constructors	
274	
primitive	
dealing with the immutability of primitive wrapper classes	
511	
wrappers	306
print()	623, 629
PrintGraphics	629
printInfo()	476
printing	630
PrintJob	628
printStackTrace()	359, 360
PrintStream	395, 408, 409
PrintWriter	420, 751
priority	
thread	725
private	28, 172, 182, 185, 209, 703
private	
and the final specifier	216
inner classes	265
process, and threading	683
processEvent()	617
Properties	309, 457
property	639
PropertyChangeEvent	652
PropertyDescriptors	645
ProptertyVetoException	652
protected	172, 182, 186, 209
and friendly	209
use in clone()	491
protocol	238
unreliable protocol	758
prototype	837
public	28, 172
and interface	238
class, and compilation units	173

pure inheritance, vs. extension	278
pure substitution	278
pushBack()	459
PushbackInputStream	393
PushBackReader	420
put(), HashMap	306
queue	315
Quicksort	310
RAD (Rapid Application Development)	476
Random.nextBytes()	343
RandomAccessFile	396, 408, 420
re-throwing an exception	360
read()	384
readChar()	409
readDouble()	409
Reader	417, 419, 718, 751
readExternal()	438
reading from standard input	411
readLine()	379, 406, 408, 409, 412, 420
readObject()	434
with Serializable	444
receive()	762
redirecting standard I/O	425
reflection	476, 642, 676
registry	
remote object registry	767
relational	
database	792
operators	97
reliable protocol	758
Remote Method Invocation (RMI)	764
RemoteException	770
remove()	319
removeActionListener()	647, 710
renameTo()	403
repaint()	618
requestFocus()	623
reset()	397
ResultSet	790
resume()	715
and deadlocks	721
reuse	
code reuse	194
right-shift operator (>>)	102
RMI	
and CORBA	778
Remote	765
RemoteException	765
rmic	769
rmiregistry	767
RMSecurityManager	767
rollover	661
RTTI	
and cloning	494
eliminating from your design	849
misuse of RTTI	845, 862
using the Class object	474
run-time binding	227
run-time type identification (RTTI)	281
shape example	463

when to use it.....	482
runFinalizersOnExit().....	273, 407
Runnable.....	735
Runnable.....	
interface.....	693
Thread.....	712
RuntimeException.....	285, 364
rvalue.....	91
seek().....	396, 409
SequenceInputStream.....	386, 420
Serializable.....	433, 438, 442, 451, 649
serialization.....	
RMI arguments.....	769
server.....	746
servlets.....	54
Set.....	285, 291, 316, 326
setActionCommand().....	599
setBorder().....	657
setChanged().....	830
setCheckboxGroup().....	543
setContents().....	632
setDaemon().....	697
setDirectory().....	575
setEditable().....	539
setErr(PrintStream).....	425
setFile().....	574
setIcon().....	661
setIn(InputStream).....	425
setLayout().....	551
setOut(PrintStream).....	425
setPriority().....	725
setSelectedIndex().....	673
setText().....	541
setToolTipText().....	656
shallow copy.....	489
shape.....	
example.....	227
shift operators.....	101
show().....	575
showDocument().....	674
showMessageDialog().....	673
showStatus().....	537
side effect.....	487
Simula-67.....	188
singleton.....	819
singleton.....	
design pattern.....	191
size(), Vector.....	292
sizeof().....	
липсата му в Java.....	110
sleep().....	685, 702, 714
Smalltalk.....	23, 25, 146
Socket.....	753
Software Development Conference	7
sort().....	342
South.....	552
specialization.....	208
splitter control.....	674
SQL.....	
stored procedures.....	794
Structured Query Language.....	788
Stack.....	302, 349
start().....	530, 586
Statement.....	790
static.....	238
and final.....	213
and inner classes.....	256
clause.....	467
initialization.....	220
inner classes.....	256
блок.....	161
ключова дума.....	145
конструкционна клауза.....	161
stop().....	530, 586
and deadlocks.....	721
deprecation in Java 2.....	721
stream-based sockets.....	758
StreamTokenizer.....	412, 420, 458, 480
String.....	
immutability.....	515
indexOf().....	399, 842
methods.....	517
operator +.....	294
Operator +.....	106
operator + and += overloading.....	199
toString().....	293
StringBuffer.....	385, 407
methods.....	522
StringBufferInputStream.....	385, 407
StringReader.....	419
StringSelection.....	632
StringTokenizer.....	415
StringWriter.....	419
subList().....	346
super.....	201
and finalize().....	273
and inner classes.....	260
super.action().....	537
super.clone().....	491, 494, 506
superclass.....	200
suspend().....	715
and deadlocks.....	721
Swing.....	
Java Foundation Classes (JFC).....	528
keyboard navigation.....	654
UI Component library.....	653
switch ключова дума.....	128
synchronized.....	703
and inheritance.....	711
and wait() & notify().....	716
collections.....	348
deciding what methods to synchronize.....	711
efficiency.....	707
method, and blocking.....	712
static.....	703
synchronized block.....	706
System.err.....	411
System.in.....	411, 422
System.out.....	411
System.out.println().....	298
TCP/IP, and RMI.....	768
template.....	
in C++.....	295
TextArea.....	539, 631

TextArea.....	446
Java 1.1.....	589
TextComponent.....	538
Java 1.1.....	589
TextField.....	538
Java 1.1.....	588
this ключова дума.....	143
Thread.....	683, 685
combined with main class.....	692
interrupt().....	721
new Thread.....	712
notify().....	712
notifyAll().....	712
properly suspending & resuming.....	723
resume().....	712
run().....	686
sharing limited resources.....	698
sleep().....	712
start().....	687
stopping.....	721
suspend().....	712
threads and efficiency.....	686
wait().....	712
Throwable.....	362
throwing an exception.....	355
TitledBorder.....	657, 673
toArray().....	338
token.....	412
Toolkit.....	629
TooManyListenersException.....	606, 650
toString().....	196, 298, 309
Transferable.....	632
transient.....	442
Transmission Control Protocol (TCP).....	758
TreeMap.....	329
TreeSet.....	326
trim().....	842
try.....	205, 373
try.....	
try block in exceptions.....	356
TYPE field, for primitive class literals.....	468
type-conscious Vector.....	294
UML, Unified Modeling Language.....	58
undo.....	674
unicast.....	650
UnicastRemoteObject.....	766
Unicode.....	418
unmodifiable collections.....	348
UnsupportedOperationException.....	340
upcasting.....	210, 224, 833
update().....	618
URL.....	675
use case.....	57
User Datagram Protocol (UDP).....	758
user interface.....	
and threads, for responsiveness.....	688
responsive, with threading.....	684
variable.....	
variable argument lists (unknown quantity and type of arguments).....	166
Vector.....	296, 299, 302, 309, 349, 833
vector of change.....	837
versioning, serialization.....	446
visual.....	
programming.....	639
programming environments.....	528
wait().....	716
Web.....	
displaying a Web page from within an applet.....	674
safety, and applet restrictions.....	562
Web browser.....	
status line.....	537
web of objects.....	434, 490
West.....	552
while.....	120
WINDOW_CLOSING.....	599
WINDOW_DESTROY.....	570
WindowAdapter.....	581
windowed applications.....	564
WindowEvent.....	599
write().....	384
writeBytes().....	409
writeChars().....	409
writeDouble().....	409
writeExternal().....	438
writeObject().....	434
with Serializable.....	444
Writer.....	417, 419, 718, 751
yield().....	712
ZipEntry.....	430
ZipInputStream.....	427
ZipOutputStream.....	427
\wedge =.....	101
$\&$ =.....	101
<.....	97
\ll =.....	102
\equiv =.....	
operator.....	493
\mid =.....	101
анализ на изискванията.....	57
бинарни.....	
оператори.....	101
битов.....	
AND оператор (&).....	101
NOT ~.....	101
OR оператор (!).....	101
вектор на промяната.....	60
двоично-комплементарно със знак.....	105
деструктор.....	146
достъп.....	
спецификатори.....	28, 171
еквивалентност.....	
==.....	97
екземпляр.....	
инициализация на нестатичен екземпляр.....	162
Експоненциална нотация.....	109
етикетиран break.....	124
етикетиран continue.....	124
индексиращ оператор ().....	163
инициализация.....	
инициализация с конструктора.....	134
на променливи в метод.....	155
инкрементално разработване.....	61

интерфейс.....	108
разделяне на интерфейса и реализацията	28
итерация, по време на разработката на	
софтуер.....	60
книгата.....	
грешки, съобщаване за.....	18
КОД.....	
стандарти за кодирането.....	17
коментари.....	
и вградена документация.....	82
компилационна единица.....	173
комплементиращ оператор.....	101
конструктор.....	
викане от други конструктори.....	144
връщана стойност.....	135
и претоварване.....	136
конструктор по подразбиране.....	137
литерал.....	
long.....	109
стойности.....	108
метод.....	
претоварване.....	135
псевдоними по време на викане на	93
не еквивалентни (!=).....	97
обект.....	
еквивалентност.....	97
обектно-ориентиран.....	
анализ & проектиране.....	55
обектно-ориентирано програмиране	23
оператор.....	
пренатоварване.....	106
приоритет.....	91
оператор за изместване наляво (<<)	102
оператор за условие.....	105
Осмичен.....	109
основа 16.....	
основа 8.....	109
плаваща запетая.....	
true и false.....	99
по-голямо (>).....	97
по-голямо или равно на (>=).....	97
по-малко (<).....	97
по-малко или равно на (<=).....	97
преобразуване.....	
стесняващо преобразуване.....	107
претоварване.....	
по връщаните стойности.....	141
различаване на претоварени методи	138
приятелски.....	172
програмиране при сървъра.....	54
проектиране.....	23
разширяващо преобразуване ..	107
реализация.....	
и интерфейс, разделяне.....	28
ред.....	
на инициализация.....	157
спецификация на системата.....	57
стесняващо превръщане.....	141
стесняващо преобразуване.....	107
страничен ефект.....	90, 97
събиране	94
тип.....	
сигурност на типовете в Java.....	107
транслационна единица.....	173
унарен.....	
минус (-).....	95
оператор.....	101
плюс (+).....	95
унарни.....	
оператори.....	95