



J A V A A C T U A L L Y



J A V A A C T U A L L Y

A Comprehensive Primer in Programming

Khalid A. Mughal, Torill Hamre and Rolf W. Rasmussen



Java Actually

Khalid A. Mughal, Torill Hamre and Rolf W. Rasmussen

Publishing Director

John Yates

Production Editor

Fiona Freel

Typesetter

WordMongers Ltd

Cover Design

Jackie Wrout

Publisher

Gaynor Redvers-Mutton

Manufacturing Manager

Helen Mason

Production Controller

Maeve Healy

Text Design

Design Delux, Bath, UK

Development Editor

Laura Priest

Marketing Manager

Mark Lord

Cover Design Controller

Jackie Wrout

Printer

C & C Offset Printing Co., Ltd

Copyright © 2007

Thomson Learning

The Thomson logo is a registered trademark used herein under licence.

For more information, contact

Thomson Learning
High Holborn House
50–51 Bedford Row
London WC1R 4LR

or visit us on the

World Wide Web at:

<http://www.thomsonlearning.co.uk>

All rights reserved by Thomson Learning 2007. The text of this publication, or any part thereof, may not be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, storage in an information retrieval system, or otherwise, without prior permission of the publisher, with the exception of any material supplied specifically for purpose of being entered and executed on a computer system for exclusive use by the purchaser of the book.

While the publisher has taken all reasonable care in the preparation of this book the publisher makes no representation, express or implied, with regard to the accuracy of the information contained in this book and cannot accept any legal responsibility or liability for any errors or omissions from the book or the consequences thereof.

Products and services that are referred to in this book may be either trademarks and/or registered trademarks of their respective owners.

The publisher and author/s make no claim to these trademarks.

British Library Cataloguing in-Publication Data

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

ISBN-13: 978-184480-418-4

ISBN-10: 1-84480-418-6

First edition published 2007 by Thomson Learning.

DEDICATION

To my one and only Chachaji, a.k.a. Mohammad Amin, for always inspiring to discover wonders in the written word.

– K.A.M.

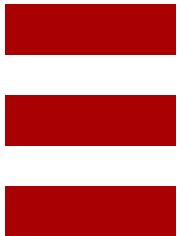
To my mother Anna and in loving memory of my father Hallstein.

– T.H.

To my brother Knut Henrik and my sister Elisabeth.

– R.W.R.

OVERVIEW



PART 1 Structured Programming 1

1 Getting started 3

2 Basic programming elements 17

3 Program control flow 45

Object-based Programming 73

4 Using objects 75

5 More on control structures 99

6 Arrays 123

7 Defining classes 161

8 Object communication 207

Some Useful Techniques for Building Programs

9 Sorting and searching arrays	235
10 Exception handling	271
11 Text file I/O and simple GUI dialogs	293
Object-oriented Programming (OOP) 327	
12 Inheritance	329
13 Polymorphism and interfaces	361
Applying OOP 393	
14 Test-driven program development	395
15 Using Dynamic Data Structures	463
16 Implementing Dynamic Data Structures	509
17 Recursion	543
18 More on exception handling	569
19 Files and Streams	585
20 Graphical User Interfaces	633
Appendices 713	
A Answers to review questions	715
B Language reference	733
C Formatted values	739
D The Unicode character set	745
E Console I/O and simple GUI dialog boxes	749
F Numeral systems and representation	757

G Programming tools in the JDK 767

H Introduction to UML 771

Index 777

CONTENTS



List of programs xxvii

List of figures xxxi

List of tables xxxv

Preface to second version xxxvii

About the book xxxvii

Book audiences xxxviii

Prerequisites xxxviii

Book themes xxxviii

Book features xxxix

Practical use of Java xxxix

Topics for programming courses xl

Conventions used in the source code xlivi

The book's web site xlivi

Feedback xlivi

The authors xlivi

Acknowledgements xlvi

Structured Programming 1

1 Getting started 3

1.1 Programming 3

1.2 Editing source code 5

Source code file naming rules 5

1.3 Development tools for Java 6

1.4 Compiling Java programs 6

1.5 Running Java programs 7

1.6 The components of a program 8

Operations 8

Programming with objects 8

Object-based programming 9

1.7 The Java programming language 9

Classes and methods **9**
Comments and indentation **10**
Program entry point **11**
Method calls **12**
Variables **12**

- 1.8** The Java Virtual Machine **13**
- 1.9** Review questions **15**
- 1.10** Programming exercises **16**

2 Basic programming elements 17

- 2.1** Printing to the terminal window **17**
 - The `print()` and `println()` methods **17**
 - Creating program output using strings **19**
- 2.2** Local variables **20**
 - Declaring variables **20**
 - Assigning variables **21**
 - Logical errors **22**
 - Literals and constants **24**
 - Choosing names **24**
- 2.3** Numerical data types **25**
 - Primitive data type `int` **26**
 - Primitive data type `double` **26**
- 2.4** Arithmetic expressions and operators **26**
 - Arithmetic expression evaluation rules **26**
 - Conversion between primitive data types **27**
 - Precedence and associativity rules **28**
 - Integer and floating-point division **29**
- 2.5** Formatted output **31**
 - Format string **31**
 - Sample format specifications **31**
 - Printing with fixed field widths **31**
- 2.6** Reading numbers from the keyboard **35**
 - The `Scanner` class **35**
 - Reading integers **36**
 - Reading floating-point numbers **37**
 - Error handling **38**
 - Reading multiple values per line **38**
 - Skipping the rest of the line when reading values from the keyboard **39**
- 2.7** Review questions **40**
- 2.8** Programming exercises **41**

3 Program control flow 45

- 3.1** Boolean expressions **46**
 - Boolean primitive data type **46**

Relational operators	46
Understanding relational operators	47
Logical operators	48
Short-circuit evaluation	49
De Morgan's laws	50
Using Boolean expressions to control flow of execution	50
3.2	Control flow: selection statements 51
	Simple selection statement: if 51
	Blocks of statements: { ... } 52
	Local variables in a block 53
	Selection statement with two choices: if-else 54
	Nested selection statements 56
3.3	Control flow: loops 59
	Loop test before loop body: while 60
	Loop test after loop body: do-while 60
	Infinite loops 61
	Using loops to implement user dialogue 61
	Choosing the right loop 63
	Nested loops 63
3.4	Assertions 64
	Making assertions 64
	Assertions as a testing technique 65
3.5	Review questions 66
3.6	Programming exercises 68

Object-based Programming **73**

4 Using objects **75**

4.1	Introduction to the object model 75
	Abstractions, classes and objects 76
	Objects, reference values and reference variables 77
	The new operator 78
	Using objects 78
	Object state 80
4.2	Strings 81
	Characters and strings 81
	String concatenation 83
	Creating string objects 85
	String comparison 85
	Converting primitive values to strings 86
	Other useful methods for strings 88
4.3	Manipulating references 89
	Reference types and variables 89
	Assignment to reference variables 89
	Aliases: one object, several references 89
	The null literal 90

- Comparing objects **90**
- 4.4** Primitive values as objects **92**
 - Auto-boxing **92**
 - Auto-unboxing **92**
 - Explicit boxing and unboxing **93**
 - Useful methods in the wrapper classes **93**
- 4.5** Review questions **95**
- 4.6** Programming exercises **97**

5 More on control structures 99

- 5.1** The extended assignment operators **100**
- 5.2** The increment and decrement operators **101**
- 5.3** Counter-controlled loops **102**
 - Local variables in the `for(;;)` loop **105**
 - Other increments in `for(;;)` loops **106**
 - Counting backwards with `for(;;)` loops **106**
 - Nested `for(;;)` loops **106**
- 5.4** Changing control flow in loops **108**
 - The `break` statement **108**
 - The `continue` statement **108**
- 5.5** Common pitfalls in loops **110**
 - Infinite loop: `for(;;)` **110**
 - One-off errors **110**
 - Errors in initialization **110**
 - Errors in the loop condition **111**
 - Optimizing loops **111**
- 5.6** Multiple-selection statements **112**
 - The `default` label **113**
 - The `case` label values **113**
 - Falling through case labels **115**
- 5.7** Review questions **117**
- 5.8** Programming exercises **120**

6 Arrays 123

- 6.1** Arrays as data structures **124**
- 6.2** Creating and using arrays **125**
 - Declaring array reference variables **125**
 - Creating arrays **125**
 - Default initialization **125**
 - Arrays of objects **126**
 - The `length` field **127**
 - Accessing an array element **127**
 - Array bounds **128**
 - Array aliases **128**

	Alternate notation for array declaration	128
6.3	Initializing arrays	129
6.4	Iterating over an array	132
	Comparing all values in an array with a given value	132
	Finding the lowest value in an array	132
	Finding the highest value in an array	133
	Adding the values in an array	133
	Iterating over an array of objects	135
6.5	Multidimensional arrays	136
	Creation and initialization of multidimensional arrays	136
	Printing a two-dimensional array in tabular form	139
	Iterating over a specific row in a two-dimensional array	139
	Iterating over a specific column in a two-dimensional array	139
	Iterating over all the columns in a two-dimensional array	139
	Ragged arrays	141
	Arrays with more than two dimensions	143
6.6	More on iteration: enhanced <code>for</code> loop	143
	Iterating over multidimensional arrays with the <code>for(:)</code> loop	144
6.7	More miscellaneous operations on arrays	145
	Copying arrays	145
	Comparing arrays	146
	Working with partially-filled arrays	147
6.8	Pseudo-random number generator	149
	The <code>Random</code> class	149
	Determining the range	149
	Simulating a dice roll	150
	Generating the same sequence of pseudo-random numbers	151
6.9	Review questions	152
6.10	Programming exercises	156

7 Defining classes 161

7.1	Class members	162
7.2	Defining object properties using field variables	163
	Field declarations	163
	Initializing fields	163
7.3	Defining behaviour using instance methods	164
	Method declaration and formal parameters	164
	Method calls and actual parameter expressions	167
	Parameter passing: call-by-value	168
	The current object: <code>this</code>	173
	Method execution and the <code>return</code> statement	174
	Passing information using arrays	177
	Automatic garbage collection	179
7.4	Static members of a class	179

- Accessing static members **182**
- Initializing static variables **183**
- The `main()` method and program arguments **184**
- 7.5** Initializing object state **186**
 - Default constructors: implicit or explicit **186**
 - Constructors with parameters **187**
 - Overloading constructors **189**
- 7.6** Enumerated types **192**
 - Simple form of enumerated types **192**
 - Selected methods for enumerated types **192**
 - General form of enumerated types **194**
 - Declaring enumerated types inside a class **196**
- 7.7** Review questions **197**
- 7.8** Programming exercises **204**

8 Object communication 207

- 8.1** Responsibilities and roles **207**
 - A naive solution to a given problem **208**
 - Combining properties and behaviour in classes **209**
 - Using references as parameters **211**
 - Advantages of good abstractions **212**
 - Rethinking responsibility **212**
- 8.2** Communication and cooperation **213**
 - Dividing and assigning responsibility **213**
 - Clarifying the responsibilities of classes in the program **216**
 - Communication between objects at runtime **217**
 - Assigning the responsibility for calculating salaries **218**
- 8.3** Relationships between objects **218**
 - Fields with reference values **218**
 - Objects that communicate **219**
 - Object ownership **220**
 - One-to-one and one-to-many associations **221**
 - Suggestions for further development **223**
- 8.4** Method overloading **223**
- 8.5** Documenting source code **224**
 - Multi-line comments **225**
 - Documenting classes and members **225**
 - Hiding internal methods and fields **226**
 - A fully-documented Java class **226**
 - How to document programs **228**
- 8.6** Review questions **231**
- 8.7** Programming exercises **231**

Some Useful Techniques for Building Programs

9 Sorting and searching arrays 235

9.1 Natural order 236

Relational operators for primitive data types 236

Understanding relational operators 236

9.2 Selection sort 238

Sorting an array of integers 238

Pseudocode for the selection sort 239

Coding the selection sort 240

Analysing the amount of work required by a selection sort 241

9.3 Insertion sort 242

Sorting an array of integers using an insertion sort 242

Pseudocode for the insertion sort 242

Coding the insertion sort 243

Analysing the amount of work required by an insertion sort 244

9.4 Sorting arrays of objects 245

Comparing objects: the **Comparable** interface 245

Implementing the natural order for time objects 246

Sorting arrays of comparable objects 249

9.5 Linear search 251

9.6 Binary search 253

9.7 Sorting and searching a CD collection 256

9.8 Sorting and searching using the Java standard library 259

9.9 Review questions 262

9.10 Programming exercises 264

10 Exception handling 271

10.1 What is an exception? 271

10.2 Method execution and exception propagation 272

Method execution 272

Stack trace 274

Exception propagation 275

10.3 Exception handling 276

try-catch scenario 1: no exception 278

try-catch scenario 2: exception handling 279

try-catch scenario 3: exception propagation 280

10.4 Checked exceptions 283

Dealing with checked exceptions using the **throws** clause 284

Programming with checked exceptions 284

10.5 Unchecked exceptions 286

10.6 Review questions 287

10.7 Programming exercises **290**

11 Text file I/O and simple GUI dialogs **293**

11.1 File handling **294**

 Data records **295**

11.2 Text files **298**

 Writing to text files **298**

 Reading from text files **304**

11.3 Simple GUI dialogue design **307**

 Overview of the **JOptionPane** class **308**

 Message dialogs – presenting information to the user **311**

 Input dialogs – reading data from the user **312**

 Confirmation dialogs – getting confirmation from the user **313**

11.4 Review questions **316**

11.5 Programming exercises **320**

Object-oriented Programming (OOP) **327**

12 Inheritance **329**

12.1 Main concepts **330**

 Superclasses and subclasses **330**

 Specialisation and generalisation **330**

 Inheritance in Java **331**

12.2 Extending properties and behaviour **332**

 Initialising object state **334**

 Extending behaviour **335**

12.3 Using inherited members **338**

 Accessing inherited members in the subclass **338**

 Accessing inherited members using subclass references **338**

 Using superclass references to refer to subclass objects **340**

 Conversion between reference types **341**

 Using superclass and subclass references **342**

12.4 Extending behaviour in the subclass **344**

 Overriding instance methods in the subclass **344**

 Using an overridden method from the superclass **346**

 Using a shadowed field **348**

12.5 Final classes and methods **349**

 Final classes **349**

 Final methods **349**

 Overloading methods **350**

12.6 Review questions **350**

12.7 Programming exercises **356**

13 Polymorphism and interfaces 361

13.1 Programming with inheritance 362

- A superclass with multiple subclasses 362
- Polymorphism and polymorphic references 365
- Arrays of polymorphic references 367
- Consequences of polymorphism 369
- Inheritance is transitive 369

13.2 Abstract classes 370

- Abstract and concrete classes 370
- Implementing abstract methods in subclasses 372
- Abstract classes in the Java standard library 373

13.3 Aggregation 374

13.4 Interfaces in Java 375

- The concept of contracts in programs 375
- Declaring an interface 376
- Implementing an interface 377
- Implementing multiple interfaces 378
- Inheritance between interfaces 379
- Polymorphic use of interface references 379

13.5 Review questions 382

13.6 Programming exercises 385

Applying OOP 393

14 Test-driven program development 395

14.1 Developing large programs 395

14.2 Simple testing with `assert` 396

- The game board 397
- Game board state 399
- The state of several cells 401
- Stacking pieces 404
- Better error messages when tests fail 405

14.3 Testing framework 406

- JUnit framework 407
- Writing test methods 407
- Running tests with JUnit 411
- Fixing test failures with JUnit 412
- Full columns 413

14.4 Printing the game board 415

- Simulated game play 415
- Method for printing game board 418

14.5 Refactoring program code 420

14.6 Interactive four-in-a-row game 422

- Functional decomposition 422

Dropping the next piece (step 3a + 3b) **424**
A simplified end of game test (step 2) **424**

14.7 Ending a game **427**

No winners **427**
Four pieces in a row **428**

14.8 Machine-controlled player **433**

The interface **IPlayer** and implementations **433**
A game between arbitrary players **435**

14.9 Class libraries **436**

Packages and the import statement **437**
Static import **437**
Declaring packages **438**
Designing frameworks **439**

14.10 Encapsulating implementation details **442**

Access modifiers for package members **442**
Access modifiers for class members **443**
Application Programming Interface (API) **447**
Limiting access to fields **448**
Tell — don't ask **449**

14.11 Base classes **449**

Base classes in the **game** package **449**
The package **game.terminal** **451**

14.12 Compiling and running code in packages **454**

The package **game.test** **455**

14.13 Review questions **459**

14.14 Programming exercises **460**

15 Using Dynamic Data Structures **463**

15.1 Overview **464**

Abstract Data Types **464**
Organising and manipulating data **464**

15.2 Character sequences: **StringBuilder** **465**

Creating a character sequence using a **StringBuilder** **465**
Modifying the contents of a **StringBuilder** **466**
Other classes for handling character sequences **467**

15.3 Introduction to generic types **469**

Declaring generic classes **471**
Using generic classes **472**
Generic interfaces **473**
Handling of generic types at compile-time **474**

15.4 Collections **475**

Superinterface **Collection<E>** **475**
Traversing a collection using an iterator **477**
Traversing a collection using the **for(:)** loop **478**

Default string representation of a collection	479
15.5 Lists: ArrayList<E>	479
Subinterface List<E>	479
Using lists	480
15.6 Sets: HashSet<E>	482
Subinterface Set<E>	482
Using sets	482
15.7 Maps: HashMap<K, V>	485
Hashing	486
The Map<K, V> interface	489
Map views	491
Using maps	492
15.8 More on generic types	495
Specifying subtypes: <? extends T>	495
Specifying supertypes: <? super T>	496
Specifying any type: <?>	496
Generic methods	496
15.9 Sorting and searching methods from the Java APIs	498
15.10 Review questions	500
15.11 Programming exercises	505

16 Implementing Dynamic Data Structures 509

16.1 Simple linked lists	509
Manipulating references in a linked list	511
Operations on linked lists	512
Inserting at the head of a linked list	518
Inserting at the tail of a linked list	519
Removing from the head of a linked list	521
Removing from the tail of a linked list	522
Removing a node inside a linked list	525
Iterator for a linked list	527
Converting to an array	528
Remarks on linked lists	528
16.2 Other data structures: stacks and queues	528
The data structure stack	528
Using stacks	531
The data structure queue	534
Using queues	536
16.3 Review questions	537
16.4 Programming exercises	538

17 Recursion 543

17.1 Recursion and iteration	544
Factorials: an example from mathematics	544

Using recursive method calls	545	
17.2	Designing recursive algorithms	546
17.3	Infinite recursion	547
17.4	Recursive binary search	548
17.5	Towers of Hanoi	551
	Recursive solution	552
	Iterative solution	555
17.6	Quicksort: a recursive sorting algorithm	556
17.7	Fibonacci series: an example of iterative solution	561
17.8	Review questions	563
17.9	Programming exercises	565

18 More on exception handling 569

18.1	Exception classes	569
	The <code>Exception</code> class	571
	The <code>RuntimeException</code> class	571
	The <code>Error</code> class	571
18.2	Throwing an exception programmatically	571
18.3	Handling several types of exceptions	573
	Typical programming errors in exception handling	575
18.4	Defining new exceptions	576
18.5	Using the <code>finally</code> block	578
18.6	Review questions	578
18.7	Programming exercises	581

19 Files and Streams 585

19.1	Streams	586
	Overview of byte streams	586
	Overview of character streams	587
19.2	Terminal window i/O	588
	Writing to the terminal window	589
	Reading from the keyboard	589
19.3	Binary files	592
	File path	592
	Writing to binary files	593
	Reading binary values	598
19.4	Object serialisation	602
	Writing objects	603
	Reading objects	610
	Effective storing of objects	611
19.5	Random access files	614
	Overview of the class <code>RandomAccessFile</code>	614

Setting up random access	616
The file pointer	616
Reading and writing on random access file	617
19.6 Review questions	626
19.7 Programming exercises	630
20 Graphical User Interfaces	633
20.1 Building GUIs	634
Designing GUI-based dialogue	634
Delegating events	634
GUI building packages	635
Basics of GUI development	635
20.2 Components and containers	635
Components	636
Containers	638
Windows and frames	639
Panels	640
Steps in developing GUI	640
GUI and the <code>main()</code> method	643
20.3 GUI control components	644
Text fields	644
Buttons	645
GUI design with panels	646
Using different types of buttons	649
20.4 Designing layout	652
<code>FlowLayout</code>	653
<code>BorderLayout</code>	653
<code>GridLayout</code>	656
20.5 Event-driven programming	661
Events	661
Event delegation model	662
Classes <code>ActionEvent</code> and <code>WindowEvent</code>	663
Programming paradigms for event handling	665
20.6 Dialogue windows	678
Application using dialogue windows	680
Implementing a dialogue window	684
20.7 Anonymous classes as listeners	687
Creating and registering of anonymous listener objects	687
Using anonymous listener objects	688
Some remarks on anonymous classes	690
20.8 Programming model for GUI-based applications	690
20.9 GUI for the four-in-a-row game	691
20.10 Review questions	697
20.11 Programming exercises	709

Appendices 713

A Answers to review questions 715

- A.1 Getting started 715
- A.2 Basic programming elements 716
- A.3 Program control flow 718
- A.4 Using objects 720
- A.5 More on control structures 721
- A.6 Arrays 723
- A.7 Defining classes 724
- A.8 Object communication 729
- A.9 Sorting and searching arrays 729
- A.10 Text file I/O and simple GUI dialogs 731

B Language reference 733

- B.1 Reserved words 733
- B.2 Operators 734
- B.3 Primitive data types 736
- B.4 Java modifiers 737

C Formatted values 739

- C.1 Syntax for format strings 739
- C.2 Conversion codes and types 740
- C.3 Examples 743

D The Unicode character set 745

- D.1 Excerpt from the Unicode character set 745
- D.2 Lexicographical order and alphabetical order 747

E Console I/O and simple GUI dialog boxes 749

- E.1 Support for console I/O 749
- E.2 Support for simple GUI dialog boxes 752

F Numeral systems and representation 755

- F.1 Numeral systems 757
 - Decimal numeral system 757
 - Binary numeral system 758
 - Octal numeral system 759
 - Hexadecimal numeral system 759
- F.2 Conversions between numeral systems 760

F.3 Conversions from decimal numeral system **761**

F.4 Integer representation **762**

F.5 String representation of integers **764**

G Programming tools in the JDK 767

G.1 Programming tools **767**

G.2 Commands **767**

Compiling source code: **javac 767**

Running the program: **java 768**

Generating documentation: **javadoc 769**

G.3 Configuring the **CLASSPATH 769**

H Introduction to UML 771

H.1 Class diagram **771**

H.2 Object diagrams **773**

H.3 Sequence diagrams **773**

H.4 Activity diagrams **774**

Index 777

LIST OF PROGRAMS

Program 1.1	The source code for a simple Java program 4
Program 2.1	Printing strings and numerical values to the terminal window 18
Program 2.2	A simple program with local variables 22
Program 2.3	Assigning new values to local variables 23
Program 2.4	Testing the division and modulus operators 30
Program 2.5	Printing a simple bill for computer goods 34
Program 2.6	Reading an integer from the keyboard 35
Program 2.7	Reading multiple integers from the keyboard 36
Program 2.8	Reading floating-point values from the keyboard 37
Program 2.9	Reading multiple values from the same line 38
Program 2.10	Skipping the rest of the current line when reading 39
Program 3.1	Calculating Boolean expressions with relational operators 47
Program 3.2	Calculating salary using a simple selection statement 52
Program 3.3	Variables defined inside a block 53
Program 3.4	Selecting alternative actions 55
Program 3.5	Calculating payment with a nested selection statement 56
Program 3.6	Calculating salary with a chain of if-else statements 59
Program 3.7	Adding a sequence of integers using loops 62
Program 3.8	Nested loops 63
Program 3.9	Using assertions to validate user input and computed results 65
Program 4.1	Objects 80
Program 4.2	String concatenation 84
Program 4.3	Use of miscellaneous String methods 88
Program 4.4	Swapping reference values 90
Program 4.5	Conversion between a primitive value and an object 93
Program 4.6	Conversions between strings, primitive values and wrappers 94
Program 5.1	Using extended assignment operators 100
Program 5.2	Using increment and decrement operators 101
Program 5.3	Use of the for(;;) statement 104
Program 5.4	Several examples using the for(;;) statement 106
Program 5.5	Using break and continue statements in a for(;;) loop 109
Program 5.6	Using a switch statement 114
Program 5.7	Common action for case labels in a switch statement 116
Program 6.1	Array declaration, creation, initialization and assignment 130
Program 6.2	Iterating over an array 133
Program 6.3	Iterating over an array of strings 135
Program 6.4	Multidimensional arrays 140
Program 6.5	Ragged arrays 142
Program 6.6	Copying and comparing arrays 146
Program 6.7	Working with partially-filled arrays 148
Program 6.8	Frequency of dice roll values using two dice 150
Program 7.1	Declaration of instance methods 166
Program 7.2	Parameter passing: call-by-value 169
Program 7.3	Arrays as actual parameters 172

Program 7.4	Handling arrays 178
Program 7.5	Static members 180
Program 7.6	Reading program arguments 184
Program 7.7	Non-default constructors 188
Program 7.8	Constructor overloading 190
Program 7.9	Use of enumerated types 193
Program 7.10	Serving meals 195
Program 8.1	Naive employee information processing 208
Program 8.2	Combining related properties and behaviour 209
Program 8.3	Class responsible for communicating with the user 213
Program 8.4	Employee class with clear responsibilities 215
Program 8.5	Responsibilities and object communication 216
Program 8.6	Use of the manager association 219
Program 8.7	One-to-many association in company with personnel register 222
Program 8.8	The complete and documented PersonnelRegister class 226
Program 9.1	Using relational operators to compare primitive values 237
Program 9.2	Sorting by selection in an array of integers 240
Program 9.3	Sorting by insertion in an array of integers 243
Program 9.4	Comparing integer values wrapped as objects 246
Program 9.5	Testing the natural order of Time objects 248
Program 9.6	Sorting arrays of Comparable objects 250
Program 9.7	Linear search in an array of integers 252
Program 9.8	Binary search in arrays of integers 255
Program 9.9	A simple CD class 256
Program 9.10	A simple CD collection with sorting and searching 258
Program 9.11	Sorting and searching using methods in the Arrays class 260
Program 10.1	Method entry and return 273
Program 10.2	Exception handling 278
Program 10.3	Exception handling (scenario 3 in Figure 10.4) 281
Program 10.4	Handling checked exceptions 285
Program 11.1	Registering employees 295
Program 11.2	Text files 300
Program 11.3	Using the showMessageDialog() method 311
Program 11.4	Using the showInputDialog() method 313
Program 11.5	Using the showConfirmDialog() method 314
Program 12.1	The superclass EmployeeVO 333
Program 12.2	A simple subclass for managers 336
Program 12.3	A client using the ManagerVO subclass 337
Program 12.4	Accessing inherited members using subclass references 339
Program 12.5	Using superclass references 340
Program 12.6	Using superclass and subclass references 343
Program 12.7	A subclass overriding an instance method 344
Program 12.8	Using the overridden instance method 345
Program 12.9	A subclass using an overridden method from the superclass 347
Program 12.10	A client using the new subclass for managers 347
Program 13.1	The subclass HourlyWorkerVO 363
Program 13.2	The subclass PieceWorkerVO 364
Program 13.3	Superclass references handling objects of multiple subclasses 366
Program 13.4	Handling arrays of employees using polymorphic references 368
Program 13.5	Polymorphic Object references 370

- Program 13.6** An abstract class with an abstract method **371**
Program 13.7 A subclass extending an abstract superclass **372**
Program 13.8 Using a reference of an abstract class **373**
Program 13.9 A simple interface **376**
Program 13.10 Class implementing an interface **377**
Program 13.11 Implementing the `ISportsClubMember` interface **380**
Program 13.12 Using interface references **381**
Program 14.1 Testing an empty game board **397**
Program 14.2 Minimal `GameBoard` implementation **398**
Program 14.3 Minimal implementation of the `pieceAt()` method **398**
Program 14.4 Adding a new test method `testDropPiece()` **399**
Program 14.5 A naive version of the `dropPieceDownColumn()` method **400**
Program 14.6 Game board with the state of only one cell **400**
Program 14.7 A naive version of the `testDropPiece()` method **401**
Program 14.8 Game board with two-dimensional array **402**
Program 14.9 Functional two-dimensional game board with tests **403**
Program 14.10 Checking stacking of pieces in a column **404**
Program 14.11 Better error messages when testing **405**
Program 14.12 Writing tests using JUnit **408**
Program 14.13 Writing tests for the game board **409**
Program 14.14 Find the bottommost vacant row **412**
Program 14.15 Testing whether a column is full **413**
Program 14.16 Simulating a sequence of moves **416**
Program 14.17 Testing of new helper method for simulating the game **417**
Program 14.18 First attempt at printing the game board **418**
Program 14.19 Printing the game board **419**
Program 14.20 Refactoring the method `gameBoardRowAsText()` **421**
Program 14.21 Text interface with the main loop and user interaction **423**
Program 14.22 Playing the game **424**
Program 14.23 Simplified simulation and testing of the column selection **426**
Program 14.24 No winner configurations **428**
Program 14.25 Unsuccessful identification of winning configuration **429**
Program 14.26 Game board that finds a winning configuration **430**
Program 14.27 `IPlayer` interface and its implementations **434**
Program 14.28 Incorporating players **435**
Program 14.29 User playing against a naive machine-controlled player **436**
Program 14.30 Interfaces in the `game` package **440**
Program 14.31 Player with random moves in the `game` package **441**
Program 14.32 Class with no access modifiers **442**
Program 14.33 Using access modifiers for class members **444**
Program 14.34 Abstract user class in the `game` package **450**
Program 14.35 Terminal user interface in the package `game.terminal` **451**
Program 14.36 Running a game against an easy opponent **453**
Program 14.37 Tests for the game in the `test` package **456**
Program 15.1 Legacy version of a pair with values **469**
Program 15.2 Generic class `Pair<T>` **471**
Program 15.3 Parameterized types **472**
Program 15.4 Lists **481**
Program 15.5 Set Theory **484**
Program 15.6 Hashing **487**

- Program 15.7** Using maps and views 492
Program 15.8 Sorting and searching with methods from the Java APIs 500
Program 16.1 Class `Node<E>` 510
Program 16.2 Interface `ILinkedList<E>` 513
Program 16.3 Classes `LinkedList<E>` and `LinkedListIterator<E>` 514
Program 16.4 Stack implementation by aggregation 529
Program 16.5 Stack client 530
Program 16.6 Graph traversal 533
Program 16.7 Queue implementation by inheritance 535
Program 16.8 Queue client 536
Program 17.1 Iterative calculation of factorial numbers 544
Program 17.2 Recursive calculation of factorial numbers 545
Program 17.3 Recursive binary search 548
Program 17.4 Recursive Tower of Hanoi solution 553
Program 17.5 Iterative Tower of Hanoi 555
Program 17.6 Sorting with the Quicksort algorithm 557
Program 17.7 Naive recursive calculation of Fibonacci numbers 561
Program 17.8 Iterative calculation of Fibonacci numbers 562
Program 18.1 Throwing an exception programmatically 572
Program 18.2 One `try` block and several `catch` blocks 573
Program 18.3 Using user-defined exceptions 576
Program 19.1 Reading from the keyboard 590
Program 19.2 Writing and reading binary values 595
Program 19.3 Handling end of file when reading binary values 600
Program 19.4 Object serialisation 604
Program 19.5 Object serialisation without duplication 612
Program 19.6 Handling of employee records using random access file 618
Program 20.1 Steps in GUI development 642
Program 20.2 GUI design with panels 648
Program 20.3 Ordering pizza 650
Program 20.4 GUI design with `BorderLayout` manager 655
Program 20.5 GUI design with `GridLayout` manager 657
Program 20.6 GUI design avoiding component stretching 660
Program 20.7 A simple application with external listener (version 1) 667
Program 20.8 External listener terminates the application (version 2) 670
Program 20.9 A simple application with external listener adapter (version 3) 671
Program 20.10 Main window as listener (version 4) 673
Program 20.11 Echo GUI with event handling 676
Program 20.12 Main window uses a dialogue window 683
Program 20.13 A dialogue window 685
Program 20.14 Implementing listener using an anonymous class (version 5) 688
Program 20.15 GUI implementation for the four-in-a-row game 693
Program 20.16 Color coding GUI components to show player identity 695
Program 20.17 GUI-based player move 696
Program 20.18 Game for two players playing against each other 697
Program E.1 Console I/O 749
Program E.2 Using console I/O 751
Program E.3 GUI dialog boxes 752
Program E.4 Using GUI dialog boxes 754
Program F.1 String representation of integers 765

LIST OF FIGURES

- Figure 1.1** Main activities in writing programs **4**
Figure 1.2 Compiling source code **7**
Figure 1.3 Terminal window showing program execution **7**
Figure 1.4 Classes and instances **9**
Figure 1.5 Class and method declarations **10**
Figure 1.6 Syntax of statements in Program 1.1 **12**
Figure 1.7 Sequence of method calls during program execution **13**
Figure 1.8 Program code at several levels **14**
Figure 2.1 Printing the sum of two integers to the terminal window **18**
Figure 2.2 Declaring a variable **20**
Figure 2.3 Declaring and initializing a variable **21**
Figure 2.4 Assigning values to local variables (Program 2.3) **24**
Figure 2.5 Operators and operands in arithmetic expressions **26**
Figure 2.6 Reading from the keyboard **35**
Figure 3.1 Control flow in selection statements and loops **51**
Figure 3.2 Simple selection statement **51**
Figure 3.3 A block of statements **53**
Figure 3.4 A selection statement with two alternatives **54**
Figure 3.5 Executing a selection statement with two alternatives **54**
Figure 3.6 Chaining `if-else` statements **58**
Figure 3.7 Execution of chained `if-else` statements **58**
Figure 3.8 Example of a `while` loop **60**
Figure 3.9 Execution of a pre-test loop **60**
Figure 3.10 Example of a `do-while` loop **61**
Figure 3.11 Execution of a post-test loop **61**
Figure 4.1 Class notation in UML **76**
Figure 4.2 Class declaration in Java **77**
Figure 4.3 Object creation **78**
Figure 4.4 Object notation in UML **79**
Figure 4.5 Dot notation **79**
Figure 4.6 Object state **80**
Figure 4.7 String creation **85**
Figure 4.8 Aliases **91**
Figure 4.9 Conversions between strings, primitive values and wrappers **95**
Figure 5.1 The `for(;;)` loop syntax **103**
Figure 5.2 Executing the `for(;;)` loop **103**
Figure 5.3 Multiple selection statement: `switch` **112**
Figure 5.4 Executing `switch` statement **113**
Figure 6.1 An array **124**
Figure 6.2 Declaration, creation and default initialization of arrays **126**
Figure 6.3 Array of objects **127**
Figure 6.4 Array creation with initialization **129**
Figure 6.5 A two-dimensional array **137**
Figure 6.6 Array of arrays: explicit structure **138**

Figure 6.7	Enhanced <code>for</code> loop 144
Figure 7.1	Overview of members in a class declaration 162
Figure 7.2	Field initialization (no constructors defined) 164
Figure 7.3	Method declaration 165
Figure 7.4	Parameter passing 168
Figure 7.5	Method execution 176
Figure 7.6	UML diagram showing all members in a class 182
Figure 7.7	Initializing of object state with non-default constructors 188
Figure 7.8	Simple form of enumerated types 192
Figure 7.9	The general form of enumerated types 195
Figure 8.1	Modifying the state of an object given as a parameter 211
Figure 8.2	Field variable storing reference value 218
Figure 8.3	References between objects 220
Figure 8.4	Associations between classes 221
Figure 8.5	Association between <code>PersonnelRegister</code> and <code>Employee</code> classes 222
Figure 8.6	Generated Javadoc documentation 229
Figure 9.1	First pass of an integer array during selection sort 239
Figure 9.2	The main steps of selection sort in an array of integers 239
Figure 9.3	The main steps of insertion sort 243
Figure 9.4	Binary search for a key that is contained in the array 254
Figure 9.5	Sorting an array of integers using bubble sort 267
Figure 10.1	Method execution (Program 10.1) 274
Figure 10.2	Exception propagation (integer division by 0) 276
Figure 10.3	<code>try-catch</code> statement 277
Figure 10.4	<code>try-catch</code> scenarios 277
Figure 10.5	Exception handling (Program 10.2) (scenario 1 in Figure 10.4) 279
Figure 10.6	Exception handling (Program 10.2) (scenario 2 in Figure 10.4) 280
Figure 10.7	Exception handling (Program 10.3) (scenario 3 in Figure 10.4) 282
Figure 11.1	Data records 295
Figure 11.2	Writing to a text file 299
Figure 11.3	Reading from a text file 306
Figure 11.4	Reading records and converting fields to appropriate values 306
Figure 11.5	Dialog windows with the <code>showMessageDialog()</code> method 311
Figure 11.6	Dialog windows with the <code>showInputDialog()</code> method 312
Figure 11.7	Dialog windows with the <code>showConfirmDialog()</code> method 314
Figure 12.1	Inheritance hierarchy 330
Figure 12.2	Overview of members in the <code>EmployeeVO</code> class 332
Figure 12.3	Object diagram for a manager 338
Figure 12.4	Superclass and subclass references 341
Figure 13.1	Inheritance hierarchy with multiple subclasses 362
Figure 13.2	Array of polymorphic references 369
Figure 13.3	Implementing abstract methods in subclasses 372
Figure 13.4	Class implementing an interface 377
Figure 13.5	Two classes implementing the same interface 379
Figure 14.1	The strategy game four-in-a-row 396
Figure 14.2	Running tests in the JUnit graphical user interface 412
Figure 14.3	Textual representation of the game board 415
Figure 14.4	Game board after several moves have been performed 416
Figure 14.5	A primitive form of four-in-a-row 421
Figure 14.6	Game without winners 428

Figure 14.7	Diagonal winning configuration 429
Figure 14.8	Counting consecutive pieces in all directions 430
Figure 14.9	Packages with game functionality 439
Figure 14.10	Package hierarchy mapped on the file system 454
Figure 15.1	Selected collections and interfaces from the <code>java.util</code> package 476
Figure 15.2	Set Theory 482
Figure 15.3	Map interface and its implementation 490
Figure 16.1	Class diagram for a simple linked list 511
Figure 16.2	A linked list of strings 512
Figure 16.3	Inheritance hierarchy for implementing linked lists 513
Figure 16.4	Inserting at the head of a linked list 519
Figure 16.5	Inserting at the tail of a linked list 520
Figure 16.6	Removing from the head of a linked list 522
Figure 16.7	Removing from the tail of a linked list (Step 1 and 2) 524
Figure 16.8	Removing from the tail of a linked list (Step 3 and 4) 525
Figure 16.9	Removing a node inside a linked list 527
Figure 16.10	A stack 529
Figure 16.11	Graph 532
Figure 16.12	Steps in finding cities reachable from city no. 3 532
Figure 16.13	A queue 534
Figure 17.1	Recursive method calls 546
Figure 17.2	Binary search for the prime number 73 550
Figure 17.3	Towers in the Hanoi game 551
Figure 17.4	Hanoi solution for one ring 551
Figure 17.5	Hanoi solution for two rings 552
Figure 17.6	Hanoi solution for three rings 552
Figure 17.7	Method execution in recursive Hanoi 555
Figure 17.8	Partitioning the array 560
Figure 17.9	Exponential increase in the number of method calls 560
Figure 18.1	Partial hierarchy of exception classes 570
Figure 19.1	Byte streams 587
Figure 19.2	Character streams 588
Figure 19.3	Writing to the terminal window 589
Figure 19.4	Reading from the keyboard 590
Figure 19.5	Writing to a binary file 593
Figure 19.6	Reading binary values 599
Figure 19.7	Writing objects and binary values 604
Figure 19.8	Reading objects and binary values 611
Figure 19.9	Random access file 617
Figure 20.1	Partial inheritance hierarchy of containers and components 636
Figure 20.2	GUI top-level window 641
Figure 20.3	GUI design with panels (Program 20.2) 647
Figure 20.4	Component hierarchy for GUI design with panels (Figure 20.3) 647
Figure 20.5	GUI for ordering pizza (Program 20.3) 650
Figure 20.6	Component hierarchy of GUI for ordering pizza (Figure 20.5) 650
Figure 20.7	<code>FlowLayout</code> manager 653
Figure 20.8	GUI design using the <code>FlowLayout</code> manager (Program 20.2) 653
Figure 20.9	<code>BorderLayout</code> manager 654
Figure 20.10	GUI design using the <code>BorderLayout</code> manager (Program 20.4) 654
Figure 20.11	Component hierarchy using <code>BorderLayout</code> manager 655

- Figure 20.12** `GridLayout` manager 657
Figure 20.13 GUI design using `GridLayout` manager (Program 20.5) 657
Figure 20.14 Component hierarchy with `GridLayout` manager 657
Figure 20.15 GUI avoiding component stretching (Program 20.6) 659
Figure 20.16 Component hierarchy with panels (Program 20.6) 659
Figure 20.17 The event delegation model 663
Figure 20.18 A simple GUI application (Program 20.7) 665
Figure 20.19 Class diagram for Program 20.7 665
Figure 20.20 Listener registration and event delegation in Program 20.7 666
Figure 20.21 Echo GUI with event handling (Program 20.11) 676
Figure 20.22 Partial inheritance hierarchy showing the `JDialog` class 679
Figure 20.23 GUI using the `JDialog` class (Program 20.12) 680
Figure 20.24 Class diagram for GUI in Figure 20.23 681
Figure 20.25 Setup for event delegation in the owner window 681
Figure 20.26 Setup for event delegation in the dialogue window 682
Figure 20.27 Anonymous class implements an interface 687
Figure 20.28 GUI for the four-in-a-row game 692
Figure 20.29 Component hierarchy for the four-in-a-row game 692
Figure E.1 Example GUI dialogs 755
Figure F.1 Conversion between decimal, octal and hexadecimal numbers 761
Figure H.1 Class diagram 771
Figure H.2 Associations 772
Figure H.3 Object diagram 773
Figure H.4 Sequence diagram 774
Figure H.5 Activity diagram 775

LIST OF TABLES

Table 2.1	Precedence of arithmetic operators 29
Table 2.2	Format specifications in Java 33
Table 3.1	Relational operators in Java 47
Table 3.2	Logical operators in Java 48
Table 4.1	Characters, Unicode representation and character literals 82
Table 4.2	String literals 83
Table 4.3	Selected methods from the String class 87
Table 4.4	Wrapper classes for primitive values 92
Table 4.5	Selected methods from the Integer class 94
Table 6.1	Default values for types 126
Table 7.1	Selected method for all enumerated types 193
Table 8.1	Javadoc markup tags for methods 225
Table 9.1	Relational operators for primitive data types 236
Table 9.2	Selected methods from the Arrays class 260
Table 10.1	Selected checked exceptions 283
Table 10.2	Selected unchecked exceptions 286
Table 11.1	Default size of primitive data types 294
Table 11.2	Summary of methods from the JOptionPane class 309
Table 11.3	Common parameters for the showTypeDialog() methods 310
Table 11.4	Specifying the message type in a dialog box 310
Table 11.5	Values returned by the showConfirmDialog() method 310
Table 11.6	Specifying the option type in the showConfirmDialog() method 311
Table 14.1	Selected methods from the org.junit.Assert class 409
Table 14.2	Access to package members 442
Table 14.3	Access to class members 443
Table 15.1	Overview of dynamic data structures 464
Table 15.2	Selected methods from the StringBuilder class 467
Table 15.3	Basic operations from the Collection<E> interface 476
Table 15.4	Bulk operations from the Collection<E> interface 477
Table 15.5	Operations in the Iterator<E> interface 478
Table 15.6	Selected list operations from the List<E> interface 480
Table 15.7	Schematic illustration of a map 486
Table 15.8	Basic operations from the Map<K, V> interface 490
Table 15.9	View operations from the Map<K, V> interface 491
Table 15.10	Selected methods from the Collections class 498
Table 15.11	Selected generic methods from the Arrays class 499
Table 15.12	Morse code [Table is not a valid element here. please fix.] 507
Table 16.1	Stack operations 529
Table 16.2	Queue operations 534
Table 17.1	Number of methods calls when calculating Fibonacci numbers 562
Table 18.1	Selected methods from the Throwable class 570
Table 19.1	Selection of methods for random access files 615
Table 20.1	Selected methods common for all components 637
Table 20.2	Selected constructors and methods for labels 637

Table 20.3	Predefined colours 638
Table 20.4	Selected methods common to all containers 639
Table 20.5	Selected methods for the root container JFrame 639
Table 20.6	Selected GUI control components 644
Table 20.7	Selected constructors and methods for text fields 645
Table 20.8	Selected constructors for buttons 646
Table 20.9	Selected methods for buttons 646
Table 20.10	Source identification for events 662
Table 20.11	Selected events and their sources 664
Table 20.12	Selected listener interfaces 664
Table 20.13	Selected constructor and methods of the root container JDialog 679
Table B.1	Keywords in Java 733
Table B.2	Reserved words for literals 734
Table B.3	Reserved words for future use 734
Table B.4	Operators in Java 734
Table B.5	Primitive data types in Java 736
Table B.6	Accessibility modifiers for classes and interfaces 737
Table B.7	Other modifiers for classes and interfaces 737
Table B.8	Access ability modifiers for class members 737
Table C.1	Methods for formatting values 739
Table C.2	Formatting parameter p with different conversion codes 741
Table C.3	Conversion flags 742
Table C.4	Combinations of conversion flags and codes 742
Table C.5	Formatting of integer values 743
Table C.6	Formatting of integer values (cont.) 743
Table C.7	Formatting of floating-point values 743
Table C.8	Formatting of strings 744
Table C.9	Using the argument index 744
Table D.1	Selected values from the Unicode character set 745
Table D.2	Characters used in Norwegian 747
Table F.1	Numerical systems 758
Table F.2	Representation of byte value with 2's complement 762
Table F.3	Methods for different string representations of integers 764
Table G.1	A subset of options for javac in the JDK 768
Table G.2	Option for turning on assertion checking 769
Table G.3	A subset of options for javadoc in the JDK 769
Table H.1	Multiplicity 772

PREFACE TO SECOND VERSION

About the book

This book provides a coherent coverage of relevant topics for a comprehensive course in programming using Java. It is ideal either for a one-semester course or a two-semester course at college and university level. Techniques for problem solving on the computer are emphasised, and the Java programming language is used for implementing the solutions.

The book assumes no prior knowledge of programming beyond the basic skills required to use a computer. It is also both platform- and programming-tool independent. The book is backed by a web site that offers lecture slides, source code for every programming example in the book, links to Java-related resources and more.

This book is a second version of our previous book, *Java Actually: A first course in programming*, and the topics covered in the first three parts of the second version essentially comprise the first version of this book. In addition, the second version, *Java Actually: A comprehensive primer in programming*, includes additional and more advanced topics suitable for a more comprehensive course.

The topics covered in this book are up-to-date with Java technology as of JDK 1.6. The aim is *not* to cover every Java-based technology under the sun, but to teach fundamental programming concepts and thereby build a foundation that the reader can use to move on to the more specialised and advanced technologies that use Java.

Our approach can be called *Objects ASAP*, meaning that objects are introduced as soon as possible. Enough structured programming concepts are covered first to write meaningful examples, before proceeding to object-based programming (OBP) for working with objects. The book emphasises testing of program behaviour using assertions, and includes common sorting and searching algorithms. It covers the way in which programs interact with their environment via the terminal window, text files, and through simple GUI (Graphical User Interface) dialogue boxes.

Object-oriented programming (OOP) and its application are the central themes in this book. Test-driven program development is emphasised and encouraged. Use of collections (lists, sets, maps) is covered after an introduction to generic types. Recursion as a powerful programming technique and exception handling for developing robust programs are discussed and demonstrated. File I/O and GUI development are used to show how a program communicates with its surroundings.

Basic UML (Unified Modelling Language) diagrams are used to illustrate Java language constructs, basic program design and programming concepts.

Book audiences

The book should be readily accessible to the following audiences:

- Anybody learning how to program for the first time.
- Students who intend to pursue studies in fields other than Computer Science and only require an introductory course in programming. The first three parts of the book can suffice for this purpose.
- Students who intend to pursue studies in Computer Science and require a sound foundation in object-oriented programming (OOP).
- Programmers with background from other languages wanting to migrate to Java.

Prerequisites

The book assumes basic knowledge in the use of the following:

- Basic computer equipment, i.e. computer with keyboard, mouse and screen.
- A normal graphical user interface with windows, buttons and menus.
- A command-line window, to execute commands in the operating system, for example to start a program.
- The file system, to create, delete and find files.
- A text editor, to write text files, for example *emacs* or *vi* on Unix, or *Notepad* on Windows. An IDE (Integrated Development Environment) can be substituted at a later stage in the course.
- The operating system, to install new programs.
- A web browser, to search on the Internet for programming resources.

Book themes

The book emphasises the following themes: [Bold element not formatting properly in the document.]

- OOP and its application. The book is structured around OOP and shows its application in different contexts. The book requires no previous experience of Java programming, and explains concepts from the ground up. It uses the classes from the Java standard library and includes numerous examples of the development of user-defined classes. A case study of developing a game (*Four in a Row*) is used to illustrate test-driven program development.
- Concepts before syntax. The book emphasises concepts and shows how these are implemented by the language features in Java. Java syntax is illustrated through examples of typical usage of the language constructs, in which the elements of the syntax are clearly identified.
- Fundamental data modelling. Both *data modelling* and *programming* are necessary in order to solve problems on the computer. Modelling of abstractions and data structures is thoroughly explained and illustrated with diagrams.
- Development of algorithms. The book encourages algorithm development, and uses pseudocode to show the progression from problem analysis to implementation of the solution.

- Example-driven exposition. The book uses appropriate examples to explain and apply concepts. Each program example is complete and shows the output, or a screenshot, from the program, so that one can easily reproduce and compare the results. The reader is made aware of the common pitfalls in programming, and practical usage of Java is emphasised through examples.
- Use of UML (Unified Modelling Language). We illustrate important programming concepts using easy-to-understand diagrams based on UML. Appendix H provides a short introduction to the notation. The book does not require any prior exposure to UML, and its use is applied only where it intuitively makes sense.
- Focus on problem solving techniques. The book uses a few well-chosen case studies to illustrate programming concepts. This approach ensures that the reader becomes familiar with the problem, and the book can focus on problem-solving techniques. Testing program behaviour using assertions is emphasised.

Book features

The source code for all the program examples in the book is available on the book's web site, and can be downloaded and experimented with. All examples are complete and can be compiled and run immediately. They have been tested thoroughly on several platforms.

Each chapter contains the following sections:

- Learning objectives. The objectives listed at the beginning of each chapter clearly outline the concepts and topics covered by the chapter.
- Review questions. Ample review questions test the topics covered in each chapter. Annotated answers to all review questions are provided in Appendix A.
- Programming exercises. The programming exercises vary in scale and level of difficulty. They help to develop programming skills.

In addition, the book offers the following:

- Best practices callouts. These callouts help to promote and facilitate good programming practices.
- Appropriate cross-references. The book provides appropriate cross-references that link related concepts.
- An extensive index. A comprehensive index aids in locating definitions, concepts and topics in the book.

Practical use of Java

We have paid special attention to the presentation and practical use of the Java programming language. The book emphasises the following aspects:

- Platform-independent programming language. Java encourages platform-independent programming, and so does this book. Specific platform-dependent details are provided only where necessary.
- Programming tool-independent exposition. The book uses Java 6.0 and the standard tools provided by Java Development Kit 1.6 (JDK 1.6). Details about compil-

ing and running Java programs using the standard command line tools *javac* and *java* are given in Appendix G. If it is desirable, other programming tools or IDEs (Integrated Development Environments) can be used. However, we feel that it is necessary to avoid the idiosyncrasies of a full-blown IDE at the early stages when learning to program.

- Use of the Java standard library. Methods from classes in the Java standard library, which are used in the programming examples, are fully documented where they are used in the book. In addition, we recommend that the reader should have access to the documentation for the Java standard library, either online or installed locally.
- Creating dialogue between the program and the user. In the examples we do not use customised classes for interaction between the program and the user. However, the book offers two classes that can be used for this purpose: a class (`Console`) that can be used to read values from the terminal window, and a class (`GUIDialog`) for creating simple graphical user interfaces for reading values via dialogue boxes (see Appendix E). The `Console` class encapsulates the use of the `java.util.Scanner` class, while the `GUIDialog` class uses the `javax.swing.JOptionPane` class. Both classes offer methods for reading integers, floating-point numbers and strings.
- Emphasising the Java Advantage. The book utilises what the Java language has to offer for an introductory course in programming. For example, assertions are introduced early in the book and used for verifying the behaviour of programs. The `Scanner` class is used to read input from the terminal window, and the `printf()` method is used to format values written to the terminal window or stored in files. Other Java 2 features in the book include the enhanced `for` loop and enumerated types.

Topics for programming courses

We have organised the material as follows:

- Part 1: Structured programming (Chapters 1 - 3)
- Part 2: Object-based programming (OBP) (Chapters 4 - 8)
- Part 3: Useful techniques for building programs (Chapters 9 - 11)
- Part 4: Object-oriented programming (OOP) (Chapters 12 - 13)
- Part 5: Applying OOP (Chapters 14 - 20)
- Part 6: Appendices (A - H)

The readers can choose either a linear or a non-linear route through the book, depending on their choice of topics. Some suggestions as to how the book can be used to tailor different types of programming courses (both in size and topic sequence) can be found in Figure P.1 and Figure P.2. Topics up to and including OBP ought to be covered in all courses: these topics are shown in shaded boxes in Figure P.1. The extent of the course can be varied by selecting the remaining topics to be included from the two diagrams. Solid arrows show optimal coverage of the material, and dashed arrows show the earliest point at which additional topics can be introduced to form appropriate course variants.

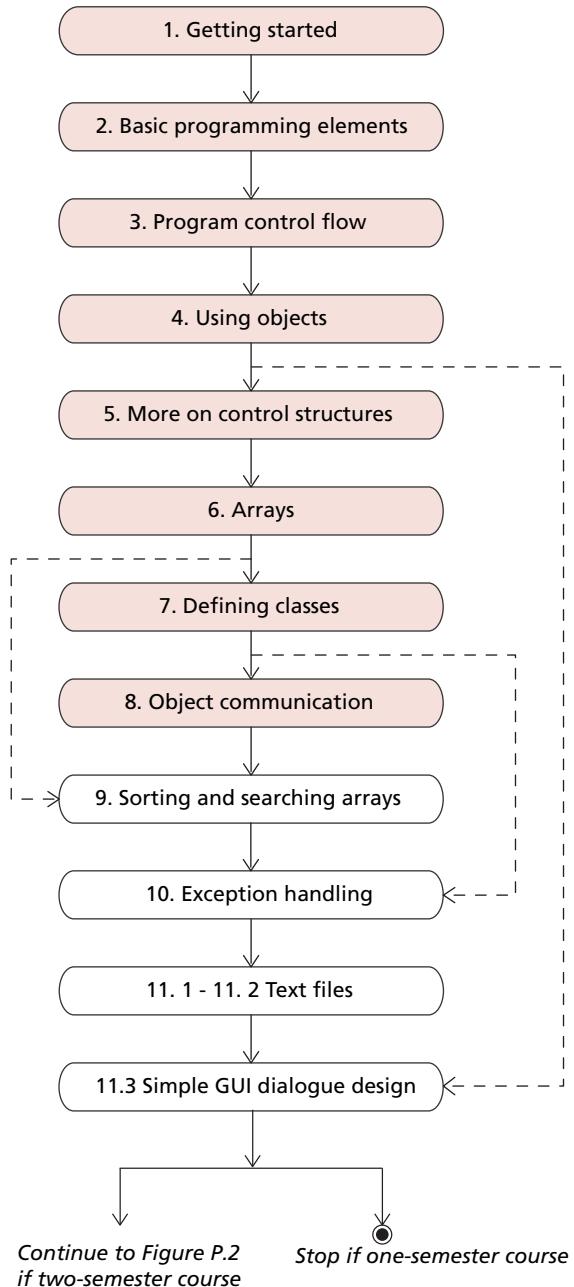


Figure P.1

Structured programming (i.e. control structures, together with strings and arrays) and OBP (i.e. objects only and no inheritance) are covered in Part 1 and Part 2 of the book, respectively. This organization allows objects to be introduced as early as possible. In Part 3, useful programming techniques include a first look at simple sorting and searching algorithms for arrays, handling exceptions, reading and writing text files, and creating simple GUI dialog boxes. This approach lays the foundation for more advanced topics in OOP (classes and interfaces using inheritance).

The first three parts of the book (depicted in Figure P.1) have been successfully used for a one-semester course. Figure P.2 shows topics from Part 4 and Part 5 of the book, that can be included to create a more comprehensive two-semester course.

Inheritance and its consequences for OOP are covered in Part 4. After covering Chapter 12 on inheritance, the remaining chapters of the book can be selected more or less independently. Part 5 includes a number of topics where OOP techniques are applied: a case study in test-driven program development, an introduction to generic types, using dynamic data structures, applying recursive techniques, understanding the exception hierarchy, doing file I/O and developing GUIs for programs.

Appendices in Part 6 provide annotated answers to review questions, useful references (keywords, operators, primitive data types, character codes, formatting code) and succinct introductions to supplementary topics (Console I/O and simple GUI dialog design, number representation, JDK tools, UML).

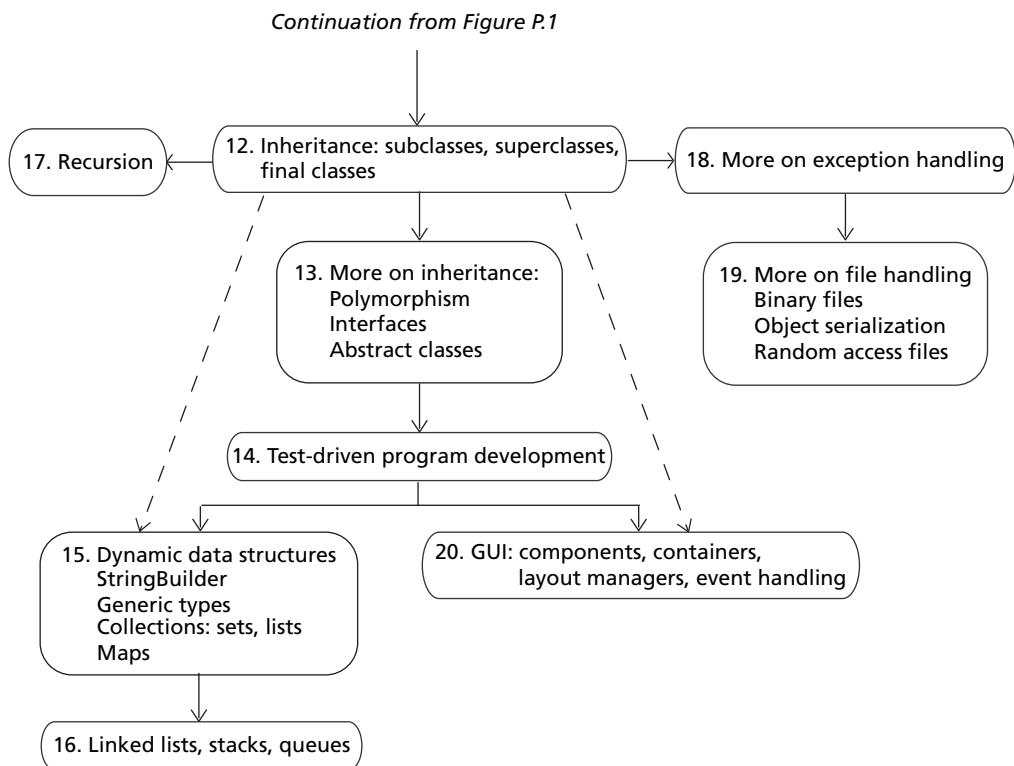


Figure P.2

Conventions used in the source code

Names in Java source code

All class and interface names begin with an uppercase letter. Names of packages, variables and methods begin with a lowercase letter. Constants are always specified with uppercase letters in their names. In addition, all method names in the text end with () to distinguish them from other names.

Code line references in the text

Code lines in examples or code snippets in the text often end with a comment, where a number in parenthesis, (), is specified after the comment characters //:

```
System.out.println("Important business"); // (4)
```

The number is used in the text to refer to the corresponding line in the code. For example, (4) in this text refers to the code line above that has the corresponding number.

The book's web site

We have created a web site for the book at <http://www.ii.uib.no/~khalid/jac/>.

The book's web site offers the following resources:

- Errata
- Source code for all the examples in the book
- Links to other useful resources: articles/tutorials on programming, web browsers, Java tools, and more.

In addition, the following resources for lecturers can be found here:

- Lecture slides
- Source code for all the examples in the lecture notes
- Links providing suggestions for projects and weekly assignments

Feedback

We appreciate getting feedback. Questions, comments, suggestions and corrections can be sent to: jac@ii.uib.no.

The authors

In 1997 the Department of Informatics, University of Bergen (UoB), switched to Java in its introductory course in programming. Mughal and Rasmussen were responsible for developing a new format for this course. In autumn 2004 the Norwegian Quality Reform in Higher Education led to further changes in the curriculum. Mughal was one of the main architects behind the revision of the programming courses at the Department of Informatics, UoB.

The three authors have collaborated on an introductory textbook for programming in Java, written in Norwegian, *Java som første programmeringsspråk /Java as First Programming*

Language, Third Edition, Cappelen Akademisk Forlag, ISBN 82-02-24554-0, Sept. 2006, <http://www.ii.uib.no/~khalid/jfps3u/>. Both versions of our book, *Java Actually*, are heavily based on the Norwegian textbook.

Mughal and Rasmussen are also co-authors of a book on the first exam in certification for the Java technology, *A Programmer's Guide to Java Certification: A Comprehensive Primer, Second Edition*, Addison-Wesley, ISBN 0201728281, Aug. 2003, <http://www.ii.uib.no/~khalid/pgjc2e/>.

The three authors have been involved in developing web-based variants of the programming courses offered by the Department of Informatics, UoB. These web-based courses are offered every spring (see <http://nettkurs.ii.uib.no/jafu/>). Mughal and Hamre have run a series of seminars on object orientation at the Department of Informatics, UoB. The authors also collaborate on research in application of object-oriented technology.

Principal author – Khalid Azim Mughal

Khalid A. Mughal is an Associate Professor at the Department of Informatics, UoB. He has developed and given courses for students and the IT industry on programming in Java and Java-related technologies. In 1999, on the basis of the introductory course in programming using Java, he was awarded the Best Lecturer prize at the Faculty of Mathematics and Natural Sciences, UoB. His teaching experience spans programming languages, object-oriented software development, web application development, e-learning, data bases and compiler techniques.

His current work involves applying object-oriented technology in the development of learning content management systems, and building software security into applications.

He has spent over four years at the Department of Computer Science, Cornell University, both as a Visiting Fellow and as a Visiting Scientist. He is a member of the ACM.

Co-author – Torill Hamre

Torill Hamre is a Research Leader at the Nansen Environmental and Remote Sensing Center in Bergen. The Nansen Center is affiliated to the University of Bergen, where she is an Adjunct Associate Professor at the Department of Informatics.

Her main research is in marine information technologies. She develops object-oriented solutions for marine information systems. She has also given courses in object-oriented software development, both at the University and to the IT industry.

Co-author – Rolf W. Rasmussen

Rolf W. Rasmussen is a System Development Manager at *vizrt* in Bergen, a company that provides solutions for the television broadcasting industry worldwide. He works on control and information systems, video processing, typography and real-time graphics visualization. Over the years he has worked both academically and professionally with numerous programming languages, including Java. He has contributed to the development of GCJ (GNU's implementation of Java), which is a part of the GNU Compiler Collection, where he has worked on the clean room implementation of graphics libraries for Java.

Acknowledgements

First, we would like to thank Gaynor Redvers-Mutton and Matthew Lane at Cengage Learning. Without Gaynor's conviction that we could pull this off, we are not sure whether we would have embarked on this venture. She managed to convince us to write, not one version of the book, but two! Thank you also for catering to our whims and for all the support during the writing process of the second version as well.

We were impressed with the results that FrameMaker guru Steve Rickaby of WordMongers Ltd. managed to conjure forth when it came to the design of the first version of the book. Using the same book design to write the second version was a breeze. Steve put the finishing touches to create the final product, with which we are very pleased. Again, many thanks, Steve!

We are also very much indebted to the four anonymous technical reviewers of the first version of this book who gave us encouraging and invaluable feedback on the initial draft of some of the book chapters.

We are fortunate to have Marit Seljeflot Mughal as our personal expert on language washing for our writing. Your efforts have saved us, time after time, not only from language bloopers, but also from errors in our Java code. Thank you for reading countless chapter drafts for both versions, and providing invaluable advice on improvements to the manuscript.

We would also like to thank the Department of Informatics, UoB, for providing an environment conducive to writing this book, and testing the course material on which the book is based.

Khalid Mughal would also like to thank the Department of Computer Science, Cornell University, for allowing him to spend his sabbatical (fall 2007 to spring 2008) there. Many chapters for the second version were written and revised at this department.

Without family support, this book simply would not have seen the light of day. Many thanks for being patient, both when the dinner got cold and when family plans changed in favour of working on the book. It is not always easy to get our priorities right when the book deadline is approaching at the speed of light.

Bergen/ Ithaca, 21 December, 2007.

Khalid A. Mughal
Torill Hamre
Rolf W. Rasmussen



P A R T O N E

Structured Programming

Getting started

LEARNING OBJECTIVES

By the end of this chapter you will understand the following:

- The main activities involved in creating and maintaining programs.
- What source code is, and how we create it.
- What the basic components of a Java program are.
- How to compile the Java source code into an executable program.
- How to run a compiled Java program.

INTRODUCTION

This chapter illustrates some important programming concepts by way of an example. We will look at how to write, build and run programs. Later chapters will provide a more thorough explanation of the concepts introduced here.

1.1 Programming

A *program* is a set of instructions that can be executed on a computer to accomplish a specific task. Modern computers execute sequences of instructions quickly, accurately and reliably. Most people who use computers today are *end users*, i.e. they do not write their own programs. They mainly use off-the-shelf programs (*software*) for many purposes: word processors to write documents, drawing programs to make illustrations and spreadsheets to perform calculations.

The task of writing new programs is called *programming*. We write programs using the language constructs of a programming language. The program in this form is called *source code* and is stored in *text files*. The source code contains human readable descriptions of



FIGURE 1.1 Main activities in writing programs

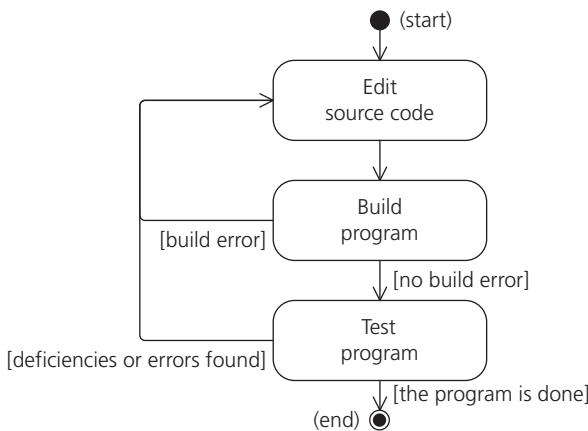


Figure 1.1 shows the main activities involved in creating programs. First we write the source code in text files, commonly called *source code files*. The source code describes exactly what tasks the computer should perform. Then we build an executable program from these source code files. At this point we need to correct any errors in the source code that prevent the program from being built. After building the program, we usually need to test it to make sure that it actually behaves the way we intended. If we detect any deficiencies or errors in the program, we need to go back and improve the source code.

Many other activities are involved when writing large programs, but the activities described in this chapter will be sufficient to get you started with programming.

There are many programming languages, but in this book we focus on the popular programming language, Java. Many of the concepts and programming techniques shown in this book are also applicable to other programming languages. The Java *compiler* is the program we use to build new programs from files containing Java source code.

Let's jump right in and take a quick look at some source code. Program 1.1 is the source code for a small program that calculates and reports the number of characters in a particular proverb. We will dissect this source code line by line later in the chapter, but for now it is sufficient to keep in mind that source code like this is stored in text files and is used to build executable programs. Before we examine the contents of Program 1.1 any further, we will look at how to enter the source code, build Java programs and run Java programs.

PROGRAM 1.1 The source code for a simple Java program

```

// (1) This source code file is called SimpleProgram.java
public class SimpleProgram {
    // Print a proverb, and the number of characters in the proverb.
    public static void main(String[] args) {                                // (2)
  
```

```

System.out.println("A proverb:");
// (3)

String proverb = "Practice makes perfect!";
// (4)
System.out.println(proverb);
// (5)

int characterCount = proverb.length();
// (6)
System.out.println("The proverb has " + characterCount + " characters.");
}

}

```

1



1.2 Editing source code

The source code files contain only characters that constitute the actual text of the source code, and no text formatting information. Word processors such as Microsoft Word are not suited for writing source code, because their primary function is creating formatted documents. Many text editors for editing source code exist (e.g. JEdit), but any application (e.g. Microsoft Notepad) that can edit and save plain text can be used.

BEST PRACTICES

Choose a good editor and spend a few hours learning its features. In the long run, this effort will pay off handsomely in terms of productivity.

Source code file naming rules

To build an executable program, the compiler requires the source code files to be named according to specific rules. A Java source code file usually contains a language construct called a *class*. The name of this class is important when naming source code files. When saving the source code of a Java program, you need to make sure that:

- The name of the source code file is the same as the name of the class it contains, followed by the extension “.java”.
- Use of lower and uppercase letters is the same as in the class name.

According to these rules, the source code file for Program 1.1 must be named “`SimpleProgram.java`”, as it contains a class named “`SimpleProgram`”.

If a source code file contains more than one class, only one class is designated as the *primary class*. The primary class is declared with the keyword `public`, and the source code file is named after this class.

The desktop environment of some operating systems hides the extension at the end of the file name by default. The compiler will refuse to compile source code files whose names



do not have the correct file extension. A common mistake is to store the source code file as either “`<name>.java.doc`” or “`<name>.java.txt`”. Such file names will not be accepted by the compiler, even if the operating system shows them as simply “`<name>.java`”. Microsoft Windows also provides alternative *short names* for files, e.g. “`MainCl~1.java`”. Such file names are also not accepted by the compiler.

1.3 Development tools for Java

There are several ways to compile and run programs, depending on the development tools you are using. The following sections show how to compile and run programs using the standard software development tools for Java.

Sun Microsystems provides a package of tools called the Java Development Kit (JDK). This package contains the basic tools needed to compile and run programs written in Java. Appendix G provides more information about this development kit.

This book will show how to compile and run programs in a *terminal window*. Most operating systems have some sort of a terminal window with a command line where you can enter commands you want to run. Read through the documentation for your operating system if you need information on how to run commands from the command line in a terminal window.

1.4 Compiling Java programs

The syntax for running the Java compiler from the command line is:

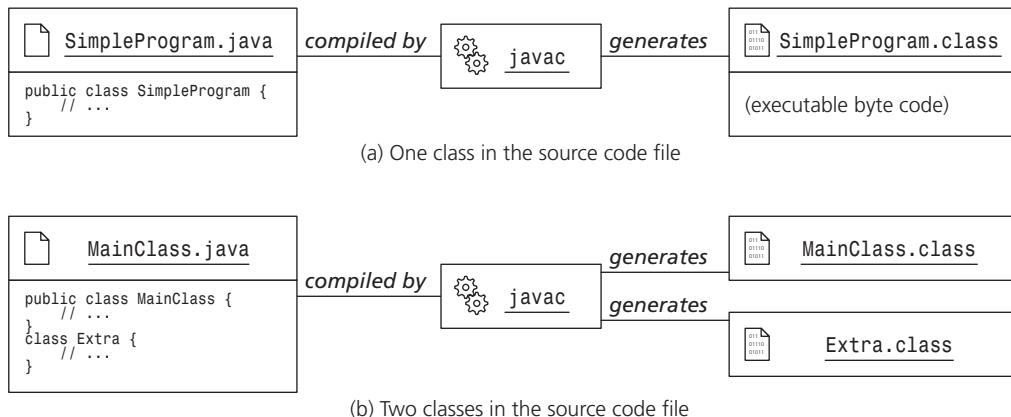
```
> javac SimpleProgram.java
```

The `javac` compiler reads the source code from the specified Java source code file, and translates each class in the source code into a compiled form known as *Java byte code* (see Figure 1.2). The compiler creates files named “`<name>.class`” that contain the byte code for each class. Section 1.8 on page 13 explains byte code.

The compiler may detect errors in the source code when translating it to byte code. The compiler will report any errors and terminate the compilation. The errors must be corrected in the source code and the compiler run again to compile the program. With a little practice, you will be able to interpret the most common errors reported by the compiler, and identify the cause of the errors in the source code.

FIGURE 1.2 Compiling source code

1



1.5 Running Java programs

The command for running a compiled Java program is “`java`”. This command should not be confused with the command “`javac`” used to compile the source code.

The syntax for running a Java program from the command line is:

```
> java -ea SimpleProgram
```

The `java` command starts the *Java virtual machine* (see Section 1.8 on page 13) that provides the runtime environment for executing the byte code of a Java program. The virtual machine starts the execution in the `main()` method of the class specified on the command line. More information on the `java` command is provided in Appendix G.

FIGURE 1.3 Terminal window showing program execution

```
C:\WINDOWS\system32\cmd.exe
C:\programming\java\source>javac SimpleProgram.java
C:\programming\java\source>java -ea SimpleProgram
A proverb:
Practice makes perfect!
The proverb has 23 characters.

C:\programming\java\source>
```

The output from running the `SimpleProgram` example is shown in Figure 1.3. The length of a string includes all characters between the double quotes, including any spaces.



The `java` command requires that the exact name of the class containing the `main()` method is specified. Here are some guidelines in case there are problems running the `java` command:

- Specify the exact class name, without any “`.class`” or “`.java`” extensions.
- Check the use of upper and lowercase letters in the class name.
- Make sure that the source code has been compiled, so that there is a “`.class`” file for each class in the program.

1.6 The components of a program

This section introduces terminology that we will use in the next section to identify the language constructs used in the source code of Program 1.1. Later chapters will elaborate on the concepts mentioned here.

Operations

When creating a new program, we break down the given problem into smaller tasks that can be accomplished by performing one or more operations. Each operation is realised by a sequence of actions that describe in detail what the computer should do. These actions are written in the chosen programming language.

Programming with objects

The operations that need to be performed to complete a task often involve several types of objects. To start thinking in terms of objects and operations, let's consider the operations needed to make an omelette:

- 1 Open the refrigerator
- 2 Take out an egg carton
- 3 Open the egg carton
- 4 Take out two eggs
- 5 Close the egg carton
- 6 Place the egg carton back in the refrigerator
- 7 Close the refrigerator
- 8 Turn on the stove
- 9 ...

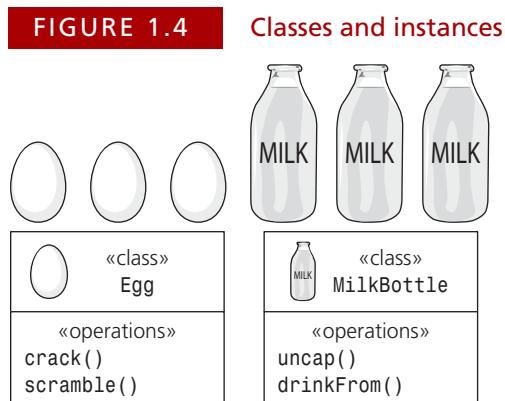
In this context we can consider the refrigerator, the egg carton, the eggs and the stove to be different types of objects. The operations that are performed are inherently connected to these objects. The type of an object determines the operations that can be performed on it. For example, it is possible to open an egg carton, but it is not possible to open a frying pan.



Object-based programming

Object-based programming (OBP) involves describing tasks in terms of operations that are executed on objects. What the objects represent depends on what the program is trying to accomplish. A program to keep track of library loans, for example, might use objects to represent tangible items such as books, journals, audio tapes, etc., and also objects to represent non-tangible concepts such as the lending date and personal information about library users.

Programs usually have more than one object of the same type. Back in the kitchen we have several objects representing bottles of milk. Each bottle can be handled independently, but they all have certain common properties, e.g. they can be uncapped and drunk from. Objects that share a set of common properties can be considered to belong to the same *class* of objects. Every egg object is a concrete *instance* of a particular Egg class, just as every milk bottle is a concrete instance of a particular MilkBottle class. This distinction is illustrated in Figure 1.4.



A class is defined by describing the properties specific to the objects of the class, and the operations that can be performed on these objects. A program can consist of user-defined classes as well as classes from other sources. The Java language comes bundled with a large collection of ready-to-use classes called the *Java standard library*. These classes contain program code that can readily be used for solving a wide range of programming problems.

1.7 The Java programming language

Classes and methods

The language constructs of the Java programming language have a prescribed structure. We call this structure the *syntax* of the language. A *class declaration* is a language construct to define classes. Operations in a class are defined by *method declarations* containing sequences of *statements* describing the actions that need to be performed.

Figure 1.5 on page 10 shows the language constructs used in the source code of Program 1.1. The program has a class declaration that specifies a class called `SimpleProgram`, and this class has a method called `main`.



Comments and indentation

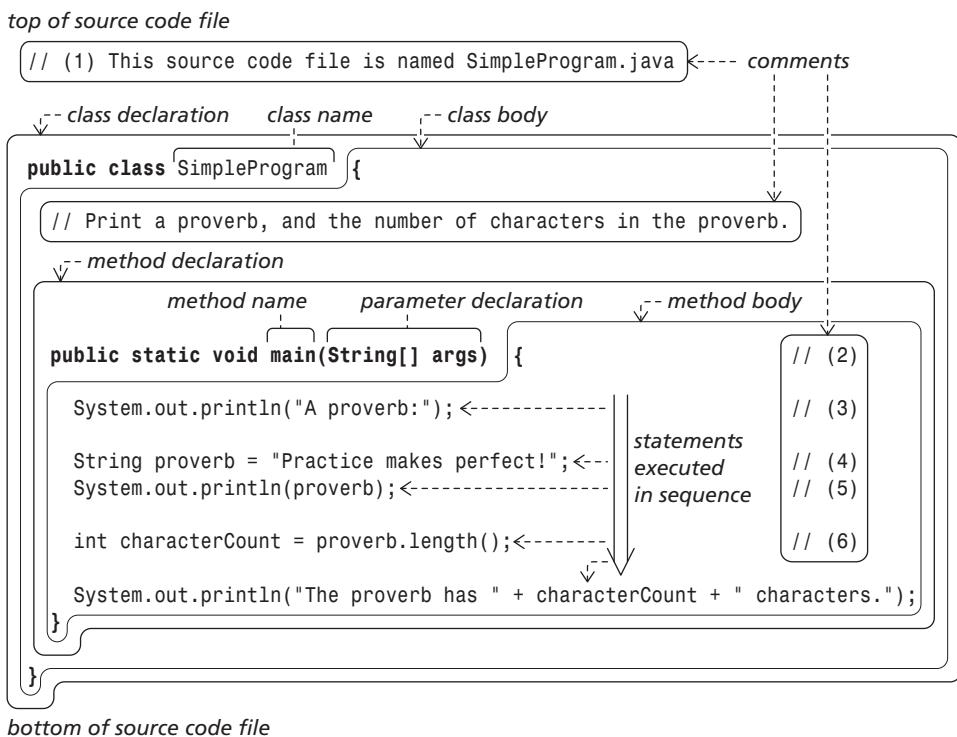
The first line of source code in Program 1.1 is:

```
// (1) This source code file is called SimpleProgram.java
```

This is a source code comment, and is ignored by the compiler. We write comments in the source code to aid programmers in understanding the inner workings of the program. Everything following the characters // on a line of source code is a comment.

When this book wants to draw attention to specific lines of code from the code examples, the code line have a comment of the form // (n), and the text refers to the lines using markers such as (n), where n is an integer.

FIGURE 1.5 Class and method declarations



The compiler completely ignores all comments, as well as some special characters such as spaces and end-of-line characters. As far as the compiler is concerned, the `main()` method declaration starting at (2) could just as well have been written like this:

```
public static void main(String[]args){System.out.println("A proverb:");
String proverb="Practice makes perfect!";System.out.println(proverb);int
characterCount=proverb.length();System.out.println("The proverb has "+
characterCount+" characters.");}
```

The layout of the source code has no bearing on program execution. It is a common practice to write one statement per line, and use indentation from the left margin to show

the nesting of the language constructs. This book uses two spaces for each indentation step to conserve the available page width.

BEST PRACTICES

In Java it is customary to use four spaces for each indentation step, and we recommend that you use do the same when you write source code yourself. Proper indentation makes the source code easier to read and modify.

1



Program entry point

Line (2) of Program 1.1 is the start of a method declaration that defines a method called `main`:

```
public static void main(String[] args) { // (2)
```

The method being defined here has the *parameter declaration* `String[] args`, as shown in Figure 1.5. A parameter declaration specifies the information a method is given when it is executed. In this particular program we are not interested in giving any information to the method, so we will just ignore the parameter declaration for now.

Throughout this book we follow the convention of writing “`()`” at the end of names to indicate that the names refer to methods. This does not necessarily mean that the methods do not have any parameters.

For a Java program to be executable, it must define exactly one `main()` method that is declared with `public static void main(String[] args) { ... }`, which is the method where program execution will start. The statements within the method body obviously vary from program to program, reflecting the task each program is trying to accomplish. The significance of using a particular method declaration to designate the *entry point* where execution of the program starts becomes clear when writing programs that have more than one method declaration.

Each line in the `main()` method body in Program 1.1 is a separate statement. When the program starts to run, the statements in the `main()` method are executed one by one, starting with the first statement. Figure 1.6 on page 12 shows the statements annotated with the language constructs they use.

BEST PRACTICES

For very small programs it is normal to have only one source code file, and to define the primary class in the file that contains the `main()` method. For larger programs, it is more practical to split the source code into several files. In such cases, it is customary to declare only one class in each file.



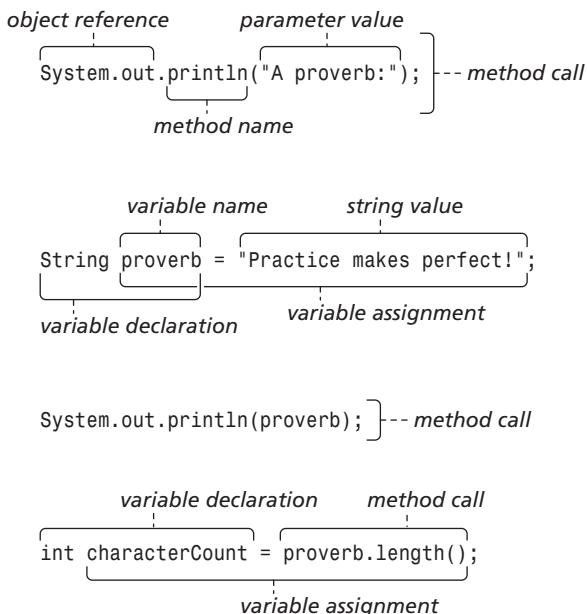
Method calls

The first statement that is executed when the program starts is (3):

```
System.out.println("A proverb:"); // (3)
```

This statement *calls* the method `println` in the object known as `System.out`. The method `println()` in this object is responsible for printing a line of text. The statement calling the `println()` method passes it the sequence of characters to be printed. Such a sequence of characters is called a *string*. The statement at (3) passes the string "A proverb:" as a parameter to the `println()` method. Figure 1.3 on page 7 shows a *terminal window* where Program 1.1 has been compiled and run. As we can see, the strings given in the `println()` method calls are printed to the terminal window.

FIGURE 1.6 Syntax of statements in Program 1.1



Variables

Variables are named locations in the computer's internal storage (memory) where values can be held during program execution. Methods often use variables to hold intermediate results. The `main()` method of Program 1.1 uses two such variables (`proverb` and `characterCount`) to store values it subsequently uses in later statements. Numeric values are very common, but as we will see later, there are other *types* of values. When we *assign* a value to a variable, we store the value in the variable so that the value can later be used by referring to the variable.

The statement at (4) declares a variable called `proverb`, and assigns the string "Practice makes perfect!" to it:

```
String proverb = "Practice makes perfect!"; // (4)
```



The program can now refer to this text string using the name `proverb`. The statement at (5) prints the string referred by the variable `proverb` to the terminal window:

```
System.out.println(proverb); // (5)
```

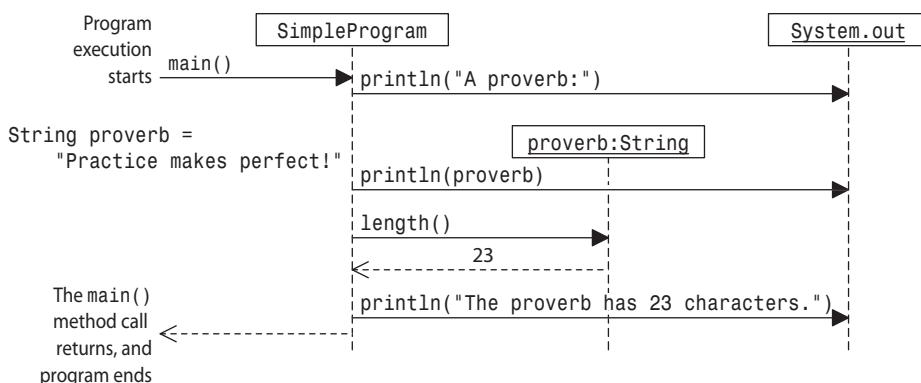
A string of text is an object in Java, and statement (6) calls the method `length()` on the string object to find out how many characters there are in the string:

```
int characterCount = proverb.length(); // (6)
```

The character count returned by the `length()` method is stored in the variable `characterCount` that is declared in the same line of code. The class of the `proverb` string objects is `String`, which is one of the many classes provided by the Java standard library. Later chapters will explore many classes from this library, including the `String` class.

When a method is called, as in statement (3), the body of the called method will be executed before program execution continues past the method call statement. Figure 1.7 shows the method calls that are executed when Program 1.1 is run. The *flow of execution* in the program can be traced by reading the method calls in Figure 1.7 from top to bottom.

FIGURE 1.7 Sequence of method calls during program execution



1.8 The Java Virtual Machine

The Java programming language provides a rich set of language constructs that allow us to express program behaviour in a way that is natural for humans. On the other hand, Java byte code is a small set of basic instructions suited for execution by machines. The Java language is considered a *high-level language*, while Java byte code is considered a *low-level language*.

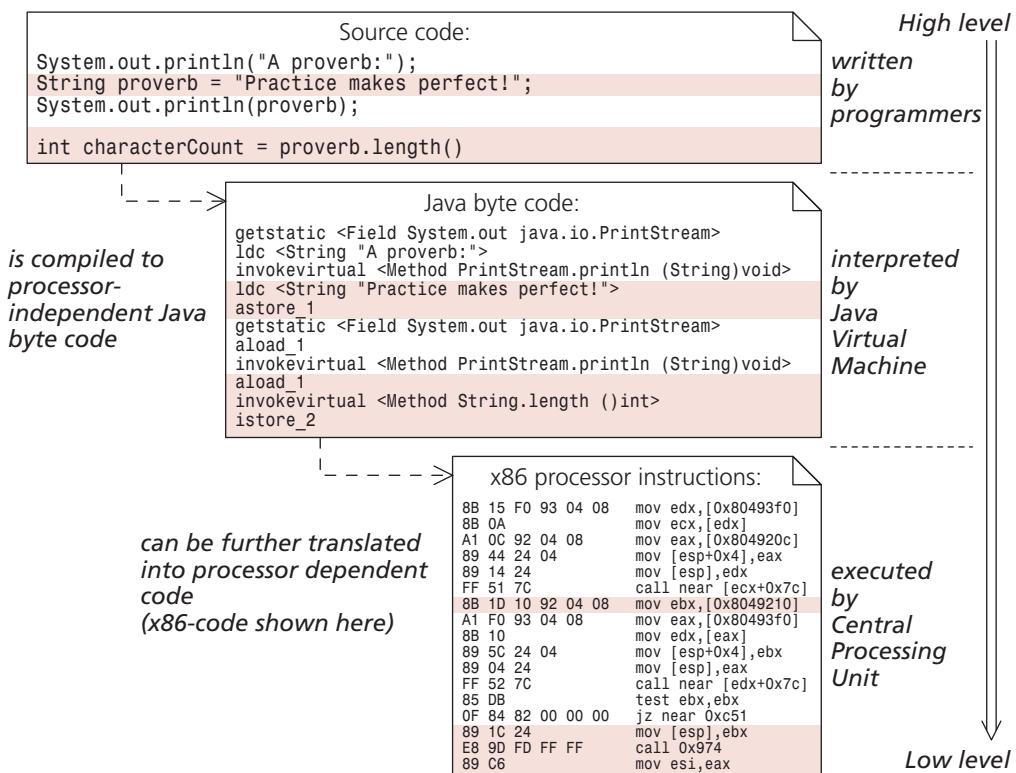
During execution, the byte code is interpreted by a virtual machine (the *Java Virtual Machine, JVM*). Rather than being a physical machine, the virtual machine is a program that interprets byte code instructions, in the same way that a central processing unit (CPU) in a computer would execute machine code instructions.



Figure 1.8 shows how a few lines of Java source code are translated to an executable form. The Java compiler translates the source code to Java byte code. The byte code can now be interpreted by a Java Virtual Machine installed on any computer. We call this *platform independence*, since it is not limited to a particular type of computer. Some virtual machines interpret the byte code directly, while others recompile it to a third form called *machine code* during execution. Machine code is at an even lower level than byte code, and will always be created specifically for the instruction set of the CPU in the computer. This means that machine code can only be executed on the specific type of computer for which it has been compiled. The machine code shown in Figure 1.8 is specific to the x86-processor series.

Some programming languages do not have an intermediate byte code representation, and lock the program to a specific platform as soon as the source code is compiled. Such programming languages allow the programmer to obtain speed improvements by programming directly at the machine code level, but this also increases the risk of programming errors. Most Java programmers will never need to examine the byte code or the machine code representation of programs.

FIGURE 1.8 Program code at several levels





1.9 Review questions

1. How many comments does Program 1.1 have?
2. A computer has a _____ that executes low-level platform-specific instructions, and makes the computer work.
3. _____ is a high-level description of the tasks the computer should perform, which are written in a high-level _____ and stored in text files.
4. Which of these components are software?
 - a A compiler.
 - b A keyboard.
 - c A virtual machine.
 - d The result of compiled source code.
 - e A CPU.
5. We specify the set of common properties we want a group of objects to share by defining a _____.
6. All Java programs have a _____ called _____, where the execution of the program starts.
7. Which statement best describes Object-Based Programming (OBP)?
 - a Making an omelette.
 - b Defining classes of objects that have common properties and operations that can be performed on these objects.
 - c Compiling source code to Java byte code.
 - d Translating programs to processor-dependent code.
8. Program 1.1 has a class called _____ and a method named _____.
9. The command _____ can be executed on the command line to compile a source code file called `TestProgram.java`.
10. The command _____ can be executed on the command line to run a Java program consisting of a primary class called `TestProgram`.
11. In which form is program code usually written and edited?
 - a Source code.
 - b Processor-independent byte codes.
 - c Processor-dependent instructions.
12. Which of these file names are valid for a Java source code file that defines a primary class called `Dog`?
 - a `Cat.java`
 - b `Dog.jav`



- c Dog.java
- d Dog.java.doc
- e DOG.JAVA

1.10 Programming exercises

1. Use a text editor to write a file called `SimpleProgram.java` containing the source code from Program 1.1.
2. Compile the source code you wrote in the previous exercise. Fix any errors that the compiler finds in the source code.
3. Run the program that was compiled in the previous exercise.
4. Write a program that prints out the number of characters in your surname. Use the source code from Program 1.1 as a base for your own program.
5. Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch is the name of a village in Wales. Taumatawhakatangihangakoauauotamateapokaiwhenuakitanatahu is the name of a hill in New Zealand. Write a program to calculate the number of characters the hill in New Zealand has in its name compared to that of the village in Wales. The result should be printed out like this, where ? is replaced by the answer:

The hill has ? characters more than the village.

Basic programming elements

LEARNING OBJECTIVES

By the end of this chapter you will understand the following:

- How to print strings and numerical values to the terminal window.
- How to store values in your programs.
- What a primitive data type is, and which primitive data types are defined by Java.
- How to write arithmetic expressions.
- How to format program output.
- How to read numbers and strings from the keyboard.

INTRODUCTION

This chapter describes some basic programming elements that we need to master in order to write computer programs. Many of these elements are also found in other programming languages, and understanding them is therefore also useful for learning languages other than Java. However, this chapter focuses on basic programming elements provided in Java, and on demonstrating their use in simple, but functioning, programs.

2.1 Printing to the terminal window

The `print()` and `println()` methods

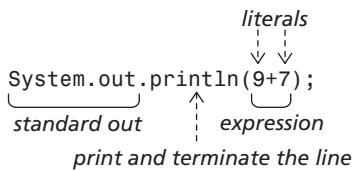
Programs can calculate values and print them directly to the terminal window. For example, the method call shown in Figure 2.1 prints the sum of the integers 9 and 7. Each of the integers is called a *literal*, and $9+7$ is an *expression*. A literal is a value written

directly in the source code. There are many types of expressions, one of the simplest being the addition of two integers. An expression always evaluates to a value.

2



FIGURE 2.1 Printing the sum of two integers to the terminal window



PROGRAM 2.1 Printing strings and numerical values to the terminal window

```
// Printing text strings and numerical values to the terminal window.  
public class SimplePrint {  
    public static void main(String[] args) { // (1)  
        System.out.println("The value of 9 + 7 is "); // (2)  
        System.out.println(9+7); // (3)  
        System.out.print("The value of 9 + 7 is "); // (4)  
        System.out.println(9+7); // (5)  
    }  
}
```

Program output:

```
The value of 9 + 7 is  
16  
The value of 9 + 7 is 16
```

Program 2.1 prints strings and numerical values to the terminal window at (2) to (5) using the `System.out` object. This object is called *standard out*, and is by default connected to the terminal window. The `System.out` object offers a `println()` method for printing a string and terminating the line. Terminating the current line results in the cursor in the terminal window moving to the beginning of the next line. There is also a `print()` method that does *not* move the cursor to the next line in the terminal window after printing on the current line. Thus, the code at (4) and (5) prints:

```
The value of 9 + 7 is 16
```

The following commands will compile and run the program, respectively:

```
> javac SimplePrint.java  
> java -ea SimplePrint
```

The JVM will start the execution of the `main()` method, which must be declared exactly as shown at (1) in Program 2.1. The syntax is fully described in later chapters, but for now we will use the following source code as a template for our programs:

```
// Comment explaining the purpose of the program.  
public class PrimaryClass {  
    public static void main(String[] args) {  
  
        // Code to solve the problem at hand...  
  
    }  
}
```

2



Creating program output using strings

A sequence of characters enclosed in double quotes ("") is called a *string literal*, or just a *string*. We can see examples of strings at (2) and (4) in Program 2.1. The operator + can be used to *concatenate* strings and string representations of other values. Concatenation means that the contents of two strings are appended and the result is assigned to a new string. For example, concatenating the strings "High" and "Five" produces a new string "HighFive". Some more examples of string concatenation are provided below. Prudent use of the concatenation operator + and the print methods can aid in formatting results before they are printed.

Details of strings can be found in Section 4.2 on page 81. Here we will only use the string operator + to create the program output we want to print to the terminal window.

The two print statements below illustrate how we can use the + operator to create strings that can be printed by calling the `println()` method:

```
System.out.println("Multiple strings can be printed" + " on the same line");  
System.out.println(  
    "We can also print a number together with this string, e.g. " + 2006);
```

We get the following output from these statements:

```
Multiple strings can be printed on the same line  
We can also print a number together with this string, e.g. 2006
```

In the first print statement, we join the two strings "Multiple strings can be printed" and " on the same line" using the + operator, and the resulting string "Multiple strings can be printed on the same line" is printed by the `println()` method.

In the second print statement, we join the string "We can also print a number together with this string, e.g." and the string representation of the value 2006, i.e. "2006", and the resulting string "We can also print a number together with this string, e.g. 2006" is printed by the `println()` method. The value 2006 is converted to its representation "2006" by the + operator.

Using the + operator on numerical values, such as in the statement:

```
System.out.println(9+7);
```



causes the two values to be added. In this case the result will be the value 16, which is sent to the `println()` method as a parameter. This method call will print the given value, converted to a string, i.e. "16", to the terminal window.

The `+` operator can be used several times in a statement. This is commonly done, for example, to print a combination of text and numerical values to the terminal window in the same method call.

Thus, we can replace the statements at (4) and (5) in Program 2.1 with the following:

```
System.out.println("The value of 9+7 is " + (9 + 7));
```

Note the mandatory use of the parentheses around the expression `9 + 7`. We want the result of the arithmetic addition, i.e. 16, to be joined with the explanatory text. Without the parentheses, the output would be:

```
The value of 9+7 is 97
```

The string "The value of 9+7 is " is first concatenated with the string representation of the value 9, then the result is concatenated with the string representation of the value 7.

2.2 Local variables

Programs can evaluate expressions and print results to the terminal window immediately. However, we often want to store values in computer memory, to retrieve them for use during program execution.

Programming languages offer *variables* for this purpose. A variable designates a location in memory where a value of a certain *data type* can be stored. A variable has a *name* that can be used to access the memory location. A data type (or just *type*) is defined by a set of valid values and a set of operations that can be performed on those values.

In this section we will look at variables defined inside methods. Such variables are called *local variables*.

Declaring variables

Figure 2.2 shows how we can declare a variable.

FIGURE 2.2 Declaring a variable

<i>declaration</i>	<i>comment</i>
<pre>int numberofCourses; // number of courses this semester</pre>	
$\begin{array}{c} \uparrow \\ \text{type} \end{array}$ $\begin{array}{c} \uparrow \\ \text{variable name} \end{array}$	

The declaration starts with a *keyword*, `int`, that specifies that the variable can only store integer values. The keyword is followed by the variable's name. A semicolon (`;`) termi-

nates the declaration. The rest of the line is a comment explaining the purpose of the variable. The comment starts with the character sequence “//”. It is not a part of the declaration.

If we need several variables of the same type, we can write them in the same declaration, separated by a comma (,). For example, two integer variables are declared by the following declaration:

```
int numberOfCourses, numberOfStudents; // number of courses and number of  
// students this semester
```

This declaration is equivalent to writing multiple declarations, one declaration per variable:

```
int numberOfCourses; // number of courses this semester  
int numberOfStudents; // number of students this semester
```

Some examples in this chapter leave out such comments, because the purpose of the variable is obvious from its name.

BEST PRACTICES

Documenting the source code with comments is very important, as it aids in understanding the program. Writing a short comment for each variable makes the purpose of the variable evident.

Assigning variables

After declaring a variable, we can *assign* a value to it. The first time we assign a value to a variable, we are *initializing* it. Later in the program we can change the value of the variable by assigning a new value. The old value is then overwritten by the new value.

In Java, we use the assignment operator = for this purpose. For example, the following statement assigns the value 2 to the integer variable `numberOfCourses`:

```
numberOfCourses = 2;
```

Whatever value the variable had before the assignment is overwritten.

FIGURE 2.3 Declaring and initializing a variable

```
declaration      assignment  
int numberOfCourses = 2;  
↑   ↑   ↑  
type  variable name  initial value
```

Program 2.2 uses two integer variables, `numberOfCourses` and `numberOfStudents`. After declaring and initializing the `numberOfCourses` variable, we print its value at (1). We can also initialize a variable as part of the declaration, as shown in Figure 2.3 and at (2) in



Program 2.2. If we want to update the value in a variable, we simply assign a new value, using the assignment operator `=`, as shown at (3).

A local variable must always be initialized before it is used. If we try to remove the line:

```
numberOfCourses = 2;
```

from Program 2.2, the compiler will report an error at (1), because the value of the variable `numberOfCourses` is not known. The compiler will not generate a class file, and we cannot run the program.

2

PROGRAM 2.2 A simple program with local variables

```
// Using local variables to hold numerical values.  
public class LocalVariables {  
    public static void main(String[] args) {  
        int numberOfCourses = 2;  
        System.out.println("Number of courses this semester is " +  
                           numberOfCourses); // (1)  
  
        int numberOfStudents = 37; // (2)  
        System.out.println("The course Java-101 has " +  
                           numberOfStudents + " students");  
  
        numberOfStudents = 23; // (3)  
        System.out.println("The course Java-102 has " +  
                           numberOfStudents + " students");  
    }  
}
```

Program output:

```
Number of courses this semester is 2  
The course Java-101 has 37 students  
The course Java-102 has 23 students
```

BEST PRACTICES

Declare local variables where they are first assigned a value. This initializes the variable properly. The source code is easier to understand, and the risk of using or modifying variables incorrectly is reduced.

Logical errors

Program 2.3 uses two local variables, which are both assigned integer values several times during program execution. It is important that if the value of a variable changes, calculations that depend on its value are also repeated, otherwise the program will report incor-



rect results. This is the case at (11) in Program 2.3, where we have intentionally skipped the calculation of the area variable after changing the value in the breadth variable at (10). The last printout is thus:

```
The area of a rectangle with length 6 and breadth 5 is 24
```

We have a *logical error* in our program, which does not behave as expected. The program reports the value that the area variable had before the value of the breadth variable was changed. The area must be recalculated with the new breadth in order to report the results correctly.

Figure 2.4 on page 24 shows how the values of the local variables change as the different assignment statements are executed from (1) to (10) in Program 2.3. The coloured background marks the value being changed by the corresponding statement (i).

PROGRAM 2.3 Assigning new values to local variables

```
// Calculating and printing the area of a rectangle.
public class Assignment {
    public static void main(String[] args) {
        int length = 5;                                // (1)
        int breadth = 4;                               // (2)
        int area = length * breadth;                  // (3)
        System.out.println("The area of a rectangle with length "
            + length + " and breadth " + breadth
            + " is " + area);
        length = 2;                                  // (4)
        breadth = length;                            // (5)
        area = length * breadth;                  // (6)
        System.out.println("The area of a rectangle with length "
            + length + " and breadth " + breadth
            + " is " + area);
        length = 6;                                  // (7)
        breadth = 4;                               // (8)
        area = length * breadth;                  // (9)
        breadth = 5;                               // (10)
        System.out.println("The area of a rectangle with length "
            + length + " and breadth " + breadth
            + " is " + area);                         // (11)
    }
}
```

Program output:

```
The area of a rectangle with length 5 and breadth 4 is 20
```

```
The area of a rectangle with length 2 and breadth 2 is 4
```

```
The area of a rectangle with length 6 and breadth 5 is 24
```



Literals and constants

A **literal** is written directly in the program. Literals can also be used to define mathematical constants (e.g. *pi*), a factor in an expression (e.g. the interest rate) or a size denoting some maximum capacity (e.g. the number of seats in a cinema hall). However, if the same literal is used in several places in a program, it is a good idea to define it as a constant.

A **constant** is a variable that cannot change its value after initialization. In Java, a constant is defined like this:

```
final double INTEREST_RATE = 3.5;
```

FIGURE 2.4

Assigning values to local variables (Program 2.3)

	length	breadth	area
After (1)	5		
After (2)	5	4	
After (3)	5	4	20
After (4)	2	4	20
After (5)	2	2	20
After (6)	2	2	4
After (7)	6	2	4
After (8)	6	4	4
After (9)	6	4	24
After (10)	6	5	24

We prefix the declaration with the keyword `final`, which indicates that the variable's value cannot be changed. If we try to assign a new value to such a variable, the compiler will report an error and the compilation will be terminated.

If the value of a constant needs to be changed, for example if the interest rate changes, we need only make the change in the declaration and compile the program. Names of constants are usually written with uppercase letters. This allows us to easily distinguish constants from variables, which may change value many times after initialization. Using uppercase letters for constants makes them easier to identify in the source code.

Choosing names

When choosing names for variables we need to remember two things: rules for what Java accepts as valid names, and conventions for variable names. The following example illustrates these rules:

```
// Rules for variable names
int agent007;           // Names can contain letters and digits
                        // but cannot start with a digit.
int 1001nights;         // This declaration will not compile!
```



```

// A digit CANNOT be the first character in a name.

int my_lucky_number; // Underscore (_) is allowed.
int _number_drawn; // Underscore can be the first character in the name.
int numberOfMinutes; // Java distinguishes between lower and upper-case
int numberofminutes; // letters, so these are two different variables.
                     // However, both names are valid in Java.

int int;           // This declaration will not compile!
                   // Keywords like int CANNOT be used as variable names.

// Conventions for variable names
int size          // Use lower-case letters.
                  numberOfHours, // Use lower-case letters, except for the first
                  itemPrize,    // letter of each consecutive word.
                  discountedItemPrize;

final int DAYS_IN_WEEK = 7,   // Constants are always in upper-case letters,
                            HOURS_IN_WEEK = 168; // and underscore is used to separate words.

int sportscarWithFourCylinders; // Avoid more than 15-20 characters in a name.

```

Table B.1 on page 733 lists all keywords in Java. These keywords cannot be used as ordinary names. It is a good idea to avoid long names. Long names can be tiresome to type, and it is easy to make a mistake. Long names also make the source more difficult to read.

BEST PRACTICES

Remember that names in Java are *case sensitive*, i.e. the name UPdown is not the same as the name upDOWN.

2.3 Numerical data types

Many programming languages provide *primitive data types* for numerical values. The primitive data types define the range of valid values and provide a set of *operators* to perform calculations on these values. A value of a primitive type is not an object, and therefore it is not possible to call any methods on a value of a primitive data type.

Java provides six different data types for integers (i.e. whole numbers) and floating-point numbers (i.e. decimal numbers). In this section, we will look at two of the most common primitive data types: `int` and `double`. In addition, Java provides the data type `char` for values that are single characters.



Primitive data type `int`

The primitive data type `int` in Java can hold values ranging from -2^{31} to $+2^{31}-1$, i.e. from -2147483648 to +2147483647. The language provides the common arithmetic operators (+, -, *, /) for calculations involving `int` values.

Integer values are commonly used for counting purposes, for example to keep track of the number of students enrolled in a course or the number of points scored in a game. Integer variables are also used to hold values that must be whole numbers, such as the number of characters in a string, and the number of books ordered from an online bookstore.

Primitive data type `double`

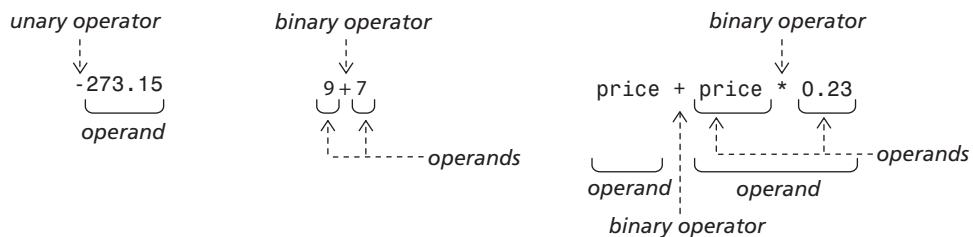
The data type `double` in Java can hold values ranging from approximately -1.7×10^{308} to $+1.7 \times 10^{308}$ (that is 170000000... followed by three hundred more zeros), with fifteen significant figures. This provides sufficient range and accuracy for floating-point values in most programming problems. The usual arithmetic operators (+, -, *, /) are provided for calculations involving `double` values.

Floating-point values can be used for fractional numbers. For example, when calculating the total cost of an order from an online bookstore, the current balance of a customer's bank account or the distance between two points when planning a hiking trip.

2.4 Arithmetic expressions and operators

The most common arithmetic operators for numerical values – multiplication, division, addition and subtraction – are available in all common programming languages. Figure 2.5 illustrates how some arithmetic expressions are constructed by using *operators* and one or more *operands*. An operator defines an arithmetic operation. Each operator accepts one or more operands as arguments, i.e. the values of the operands are used in the calculation for the chosen operation.

FIGURE 2.5 Operators and operands in arithmetic expressions



Arithmetic expression evaluation rules

The operators for multiplication (*), division (/), addition (+), subtraction (-) and modulus (%) all require two operands. They are therefore termed *binary operators*.

You might be unfamiliar with the modulus operator (%). It calculates the remainder after division is performed. Here are some examples:

- $16 \% 4$ evaluates to 0
- $15 \% 2$ evaluates to 1
- $15 \% 4$ evaluates to 3

The modulus operator is useful to calculate, for example, the number of players without an opponent in a tennis tournament. If fifteen players have signed on, and they are to play single matches simultaneously, there will be $15 \% 2$, i.e. one player, without an opponent. For a tournament with double matches, on the other hand, there would be $15 \% 4$, i.e. three players left who would not be able to play a doubles match.

There are two *unary operators*, + (unary plus) and - (unary minus), which only accept one operand. This operand must also be an arithmetic expression that evaluates to a numerical value. The simplest forms of arithmetic expression are a literal, a variable or a constant.

The following expressions are all valid in Java:

- $4 + 16$, which evaluates to the integer value 20
- $12 / 3$, which evaluates to the integer value 4
- $12.0 / 3.0$, which evaluates to the floating-point value 4.0
- $-16.50 / -2.0$, which evaluates to the floating-point value 8.25

With the exception of the addition operator +, the operators above are only defined for numerical values. The + operator can also be used to concatenate strings, as we have already seen.

Conversion between primitive data types

If the operands of a binary operator are of the same type, the result is also of the same type as the operands. If we combine numerical values of different types using a binary operator, the two values will be converted to a common type before the operation is performed. This is called *type conversion*. In some cases the rules of the Java programming language ensure that the proper conversion is implicitly carried out (*implicit conversion*), while in other cases we have explicitly to specify the conversion in the source code (*explicit conversion*).

Implicit conversions

Since the range of values for the data type `double` is broader than that of the data type `int`, we say that `double` is a *broader data type* than `int`. Vice versa, we say that `int` is a *narrower data type* than `double`.

If we use both integers and floating-point values in an expression, Java automatically converts the integer value to a floating-point value before the expression is evaluated. For example, the integer value 4 in the expression $4 + 12.6$ will be converted to the floating-point value 4.0 before the addition is carried out. In arithmetic expressions with binary operators, Java automatically *promotes* operand values to the broader type before evaluation is performed.





Similarly, Java will automatically perform type conversion if we assign a value of a narrow data type to a variable of a broader data type. For example, when assigning an integer value to a floating-point variable, as in (1):

```
int numberOfFullHours = 10;  
double numberOfHours = numberOfFullHours; // (1)
```

Java automatically performs an implicit conversion, converting the integer value 10 to the floating-point value 10.0 (of type `double`) before the assignment is performed.

Explicit conversions

If we assign a numerical value of a broad data type to a variable of a narrower data type, we risk loss of information. This can happen, for example, if we assign a floating-point value to an integer variable. If the floating-point value has a decimal part, this cannot be stored in the integer variable. To avoid accidental loss of information, Java demands that we explicitly specify that this type of conversion is to be performed, as shown below:

```
double numberOfHours = 40.65;  
int numberOfFullHours = (int) numberOfHours; // (1) Value 40 is assigned.
```

The floating-point value is *truncated*. We specify explicit type conversion, or *casting*, by the notation `(typename)`, where `typename` is the type to which we want to convert. If we leave out `(int)` from the assignment statement in (1) above, the compiler will reject it. Hence explicit conversion is required in cases where we would otherwise risk losing precision, such as when truncating a decimal value to the nearest integer. Explicit casts inform the compiler that we are aware of the potential loss of precision, and that this is acceptable in our program.

Precedence and associativity rules

Evaluation of the *operands* of an arithmetic operator in Java is always performed from left to right. How the *operators* are applied is determined by operator *precedence*, which specifies the mutual ranking between different operators. If two operators with *different precedence* are next to each other in an expression, the operator with the highest precedence is applied first. For example, the expression `4 + 5 * 2` will be interpreted as `4 + (5 * 2)` during compilation. This expression will be evaluated to 4 + 10, and finally to 14, because the `*` operator has higher precedence than the `+` operator. The parentheses are used to indicate the sequence in which the compiler associates operands to operators when interpreting the expression.

We can change the order in which operators are applied by using parentheses. The parentheses in the expression below force the addition operator to be applied first, resulting in the value 18:

```
(4 + 5) * 2
```

Table 2.1 shows the precedence of the arithmetic operators in Java.



TABLE 2.1 Precedence of arithmetic operators

Precedence level	Type operator	Operator
high	unary	+, -
	binary	*, /, %
low	binary	+, -

Associativity rules are used to determine which operator will be evaluated first if there are two consecutive operators with the *same precedence* in an expression. *Left associativity* means that *operands are grouped from left to right*. For example, the expression:

$$1 + 2 - 3$$

will be interpreted as $((1 + 2) - 3)$, since the binary operators + and - are both left-associative. *Right-associativity* means the operands are *grouped from right to left*. For example, the expression $- - 4$ will be interpreted as $(- (- 4))$, which evaluates to 4, because the unary operator - is right-associative.

Unary operators have higher precedence than binary operators. Multiplication, division and modulus are on the same level of the precedence hierarchy, and all have higher precedence than addition and subtraction. Consequently, the expression $5 * - 6 / 2$ is evaluated as follows:

$$\begin{aligned} &= 5 * (- 6) / 2 \\ &= (5 * -6) / 2 \\ &= -30 / 2 \\ &= -15 \end{aligned}$$

BEST PRACTICES

Use additional parentheses in arithmetic expressions to make the evaluation order explicit, and include extra spaces to make the expression easier to read.

Integer and floating-point division

Most arithmetic operators in Java behave as one would expect them to do from basic knowledge of arithmetic or from using a pocket calculator. One exception is the division operator / when both the numerator and denominator are integer expressions. *Integer division* always results in an integer value. If we try to calculate the expression $30 / 4$ in a Java program, the result will be 7, and not 7.5 as a pocket calculator would have computed. However, if one of the operands of the division operator / is a floating-point value, a *floating-point division* is performed.

Program 2.4 illustrates arithmetic operators. It also shows what happens if we try to divide by zero. Note the difference between integer and floating-point division. If we divide the numerator by the floating-point value 0.0, we get the value *Infinity* or *NaN*.

(*Not a Number*), depending on whether the numerator is zero or not. Integer division by 0 always results in a runtime error, called an *exception*, no matter what value the numerator has. In this case, the Java Virtual Machine will print a message about the cause of the error and terminate the program.

Program 2.4 illustrates the use of the division and the modulus operators. It is instructive to try these operators on other values.

PROGRAM 2.4 Testing the division and modulus operators

```
2 // Experimenting with the division operator.  
public class Division {  
    public static void main(String[] args) {  
        System.out.println("Integer division and modulus:");  
        System.out.println(" 3/2 = " + (3/2));  
        System.out.println(" 4/4 = " + (4/4));  
        System.out.println(" 3%2 = " + (3%2));  
  
        System.out.println("Floating-point division and modulus:");  
        System.out.println(" 3.0/2.0 = " + (3.0/2.0));  
        System.out.println(" 3.0/4.0 = " + (3.0/4.0));  
        System.out.println(" 3.0%2.0 = " + (3.0%2.0));  
  
        System.out.println("Division by zero:");  
        System.out.println(" 2.0/0.0 = " + (2.0/0.0));  
        System.out.println(" 0.0/0.0 = " + (0.0/0.0));  
        System.out.println(" 2.0/0 = " + (2.0/0));  
        System.out.println(" 2/0 = " + (2/0));  
    }  
}
```

Program output:

```
Integer division and modulus:  
 3/2 = 1  
 4/4 = 1  
 3%2 = 1  
Floating-point division and modulus:  
 3.0/2.0 = 1.5  
 3.0/4.0 = 0.75  
 3.0%2.0 = 1.0  
Division by zero:  
 2.0/0.0 = Infinity  
 0.0/0.0 = NaN  
 2.0/0 = Infinity  
Exception in thread "main" java.lang.ArithmaticException: / by zero  
 at Division.main(Division.java:18)
```

2.5 Formatted output

Program output printed to the terminal window can be formatted using the `printf()` method of the `System.out` object:

```
printf(String format, Object... args)
```

The parameter `format` specifies how formatting will be done. This is a *format string* containing *format specifications* that determine how each subsequent value in the parameter `args` will be formatted and printed. The parameter declaration `Object... args` means that the method accepts zero or more parameters.

2



The following calls to the `printf()` method:

```
System.out.printf("Player\Game      %6d%6d%6d%n", 1, 2, 3);
System.out.printf("%-20s%6d%6d%6d%n", "F. Resmann", 320, 160, 235);
System.out.printf("%-20s%6d%6d%6d%n", "A. King", 1250, 1875, 2500);
```

will generate this tabular printout of game results:

Player\Game	1	2	3
F. Resmann	320	160	235
A. King	1250	1875	2500

The format specifications used here are explained below when we write a short program to print a formatted bill for a computer store (see Program 2.5).

Format string

A format string can contain both *fixed text* and *format specifications*. The fixed text is printed exactly as specified in the format string. The format specifications control how the values of the subsequent parameters are formatted and printed.

Formatting of floating-point values is *localised*. This means that the formatting is customised to a *locale*. A locale defines, among other things, the character used as the decimal point in a country or a region. Comma (,) is used as decimal point in Norway and Denmark, while in many other countries, such as the United Kingdom and the USA, the dot (.) is used to separate the integer and decimal part of a floating-point value. We will primarily use the dot (.) as decimal point in this book.

Sample format specifications

Table 2.2 shows some common format specifications in Java. See also Appendix C. All values are converted to their string representation and formatted by the `printf()` method before being printed to the terminal window.

Printing with fixed field widths

Before diving further into formatted printing, let's look at an example. Program 2.5 on page 34 prints a simple bill using the `printf()` method.

At the top of the bill, the company name is printed at (1) with a format string that contains only fixed text. Date and time of day is printed on the same line, with leading zeros at (2). A heading is then printed at (3) by means of the format specification "%-24s" for the left-justified column `Item`, "%8s" for the right-justified columns `Price` and `Sum`, and "%6s" for the right-justified column `Count`.




TABLE 2.2 Format specifications in Java

Parameter value	Format specification	Example value	String printed	Comment
Integer	"%d"	125	" 125 "	Occupies as many character places as needed.
	"%6d"	125	" 125 "	Occupies six character places and is right-justified. The printed string is padded with spaces to the left.
	"%02d"	3	"03 "	Occupies two character places and is padded with leading zeros.
Floating point value	"%f"	16.746	"16.746000"	Occupies as many character places as needed, but always includes six decimal places.
	"%.2f"	16.746	"16.75"	Occupies as many character places as needed, but includes only two decimal places.
	"%8.2f"	16.7466	" 16.75"	Occupies eight character places, including the decimal point, and uses two decimal places.
String	"%s"	"Hi!"	"Hi!"	Occupies as many character places as are needed.
	"%12s"	"Hi Dude!"	" Hi Dude!"	Occupies twelve character places and is right-justified.
	"%-12s"	"Hi Dude!"	"Hi Dude! "	Occupies twelve character places and is left-justified.
Linefeed	"%n"	(none)	(none)	Moves the cursor to the next line in the terminal window.

Underneath the heading, the items purchased are printed at (4)–(5), (6)–(7) and (8)–(9) using the same field widths as the column headings. The item name is printed with the format string "%-24s", resulting in a twenty-four character wide string, left justified. The item price and the total cost for each type of item are printed as floating-point values using the format specification "%8.2f", and the number of items is printed as an integer using the format specification "%6d". The strings are left-justified, while all numbers are right-



PROGRAM 2.5 Printing a simple bill for computer goods

```
// Using the printf() method to prepare a nicely formatted bill.
public class SmallBill {
    public static void main(String[] args) {
        System.out.printf("Easy Data Ltd.                ");
        System.out.printf("%02d/%02d/%04d, %02d:%02d:%02d%n%n",
                           20, 3, 2006, 19, 6, 9); // (1)
        System.out.printf("%-24s %8s %6s %8s%n",
                           "Item", "Price", "Count", "Sum"); // (2)

        int count =2;
        double price = 132.25, sum = count*price, total = sum;
        System.out.printf("%-24s %8.2f %6d %8.2f%n",
                           "Ultraflash, USB 2.0, 1GB", price, count, sum); // (3) // (4)

        count =1;
        price = 355.0; sum = count*price; total = total + sum;
        System.out.printf("%-24s %8.2f %6d %8.2f%n",
                           "Mega HD, 300GB", price, count, sum); // (5) // (6)

        count = 3;
        price = 8.33; sum = count*price; total = total + sum;
        System.out.printf("%-24s %8.2f %6d %8.2f%n",
                           "USB 2.0 cable, 2m", price, count, sum); // (7) // (8)

        System.out.printf("%40s %8.2f%n", "Total:", total); // (9) // (10)
    }
}
```

Program output:

Easy Data Ltd. 20/03/2006, 19:06:09

Item	Price	Count	Sum
Ultraflash, USB 2.0, 1GB	132.25	2	264.50
Mega HD, 300GB	355.00	1	355.00
USB 2.0 cable, 2m	8.33	3	24.99
		Total:	644.49



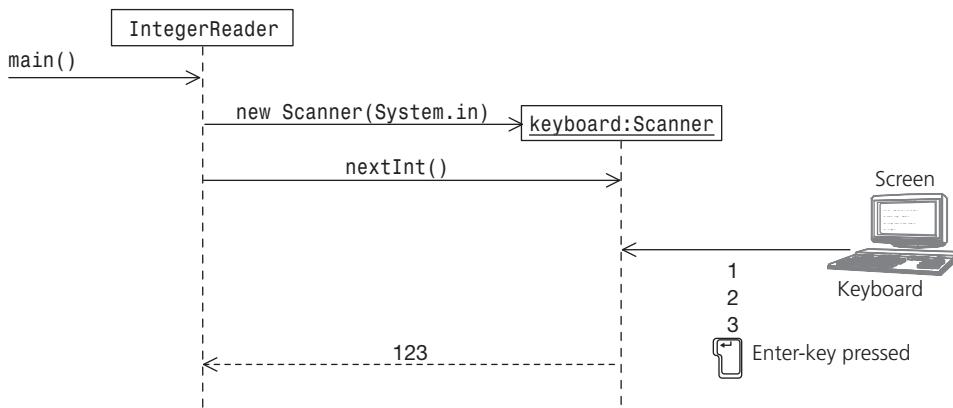
2.6 Reading numbers from the keyboard

The Scanner class

This book uses the class `Scanner` from the Java standard library to read numbers and strings from the keyboard. This class offers two methods, `nextInt()` and `nextDouble()`, for reading integer and floating-point values respectively.

Figure 2.6 shows the method calls involved in reading an integer in Program 2.6. First we have to tell the Java compiler that we want to use the `Scanner` class from the Java standard library in our program. This is done by the `import` statement at (1). In (2) we create a `Scanner` object and connect it to the keyboard through the `System.in` object, which is called the *standard in*. The program then prints a prompt at (3) to let the user know what type of value is required. Reading from the keyboard is done at (4) by calling the `nextInt()` method in the `Scanner` class. The value read is assigned to a variable of the correct type (in this case, `int`). At (5) the program echoes the number read, so that we can check whether reading from the keyboard was successful.

FIGURE 2.6 Reading from the keyboard



PROGRAM 2.6 Reading an integer from the keyboard

```

// Reading an integer from the keyboard using the Scanner class.
import java.util.Scanner; // (1)
public class IntegerReader {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in); // (2)

        System.out.print("Enter an integer: "); // (3)
        int numberRead = keyboard.nextInt(); // (4)

        System.out.printf("You entered the number %d%n", numberRead); // (5)
    }
}

```

```
}
```

Program output:

```
Enter an integer: 123
You entered the number 123
```

2



Reading integers

Values read from the keyboard can be stored in the program and used in computations. Program 2.7 shows how we can let a user enter the length and breadth of a rectangle at the keyboard to calculate the area of the rectangle.

At (1) and (3) the program prompts the user to enter a value. Reading the value is done at (2) and (4), and in both cases an integer value is read by calling the `nextInt()` method. When the area has been calculated at (5), its value is printed at (6). Note that the program uses the `print()` method, rather than the `println()` method, to display the explanatory text in the terminal window, allowing the user to enter a value on the same line as the prompt text.

PROGRAM 2.7 Reading multiple integers from the keyboard

```
// Calculating the area of rectangle whose sides are input from the keyboard.
import java.util.Scanner;
public class IntegerArea {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);

        System.out.print("Enter the rectangle length [integer]: ");      // (1)
        int length = keyboard.nextInt();                                // (2)
        System.out.print("Enter the rectangle breadth [integer]: ");     // (3)
        int breadth = keyboard.nextInt();                               // (4)

        int area = length * breadth;                                    // (5)
        System.out.printf("A rectangle of length %d cm and breadth" +
                           " %d cm has area %d sq. cm.%n",
                           length, breadth, area);                            // (6)
    }
}
```

Program output:

```
Enter the rectangle length [integer]: 6
Enter the rectangle breadth [integer]: 4
A rectangle of length 6 cm and breadth 4 cm has area 24 sq. cm.
```

BEST PRACTICES

Provide meaningful prompts to make the use of the program self-explanatory. The prompts should be short, enabling a user to enter the required value on the same line.

2



Reading floating-point numbers

To make the calculation of the area more flexible, we can modify the program to read floating-point values. The source code using floating-point variables and values is shown in Program 2.8. All variables are now of type `double`, and the program calls the `nextDouble()` method at (2) and (4) to read floating-point values. The program will also read any integer value entered at the keyboard as a floating-point value.

PROGRAM 2.8 Reading floating-point values from the keyboard

```
// Calculating the area of rectangle whose sides are input from the keyboard.  
import java.util.Scanner;  
public class FloatingPointArea {  
    public static void main(String[] args) {  
        Scanner keyboard = new Scanner(System.in);  
  
        System.out.print("Enter the rectangle length [decimal number]: ");// (1)  
        double length = keyboard.nextDouble(); // (2)  
        System.out.print("Enter the rectangle breadth " +  
                        "[decimal number]: "); // (3)  
        double breadth = keyboard.nextDouble(); // (4)  
  
        double area = length * breadth; // (5)  
        System.out.printf(  
            "A rectangle of length %.2f cm and breadth %.2f cm" +  
            " has area %.2f sq. cm.%n",  
            length, breadth, area); // (6)  
    }  
}
```

Program output:

```
Enter the rectangle length [decimal number]: 15.5  
Enter the rectangle breadth [decimal number]: 4.25  
A rectangle of length 15.50 cm and breadth 4.25 cm has area 65.88 sq. cm.
```



Error handling

When entering numbers at the keyboard in Program 2.7 or Program 2.8, a user might unintentionally enter an invalid value. For example, typing a floating-point value when running Program 2.7 will result in the program terminating with the following error message:

```
Enter the rectangle length [integer]: 6.5
Exception in thread "main" java.util.InputMismatchException
  at java.util.Scanner.throwFor(Unknown Source)
  at java.util.Scanner.next(Unknown Source)
  at java.util.Scanner.nextInt(Unknown Source)
  at java.util.Scanner.nextInt(Unknown Source)
  at IntegerArea.main(IntegerArea.java:7)
```

This type of error is called an *exception* and is explained in Chapter 10.

The `Scanner` class will take the localisation into consideration when reading values. For example, if the localisation is for the UK, a dot (.) must be used as the decimal point. Should a user accidentally use a comma (,) as the decimal point, the `nextDouble()` method will print an error message similar to the one above and the execution will be aborted.

For both integer and floating-point values, the syntax of the input value determines whether it is accepted. The `Scanner` class has no way of catching values that are meaningless to the program. For example, a user can enter a negative value for length and/or breadth in Program 2.7 and Program 2.8. Since the program does not check the values, the area will still be computed. Input validation is the responsibility of the calling method, in this case the `main()` method. The `Scanner` class is meant to be flexible, and therefore only offers *syntactic validation* of input data.

Reading multiple values per line

Using the `Scanner` class, we can also read multiple values from the same line in the terminal window, as illustrated by Program 2.9. After the user has entered two integers, the call to the `nextInt()` method at (1) assigns the first value to the variable `number1`, and the call to the same method at (2) assigns the second value to the variable `number2`. The program then adds the two input values and prints the result to the terminal window at (3).

PROGRAM 2.9 Reading multiple values from the same line

```
// Reading multiple values from the same line on the keybouard.
import java.util.Scanner;
public class ReadingMultipleValues {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);

        System.out.print("Enter two numbers [integer integer]: ");
        int number1 = keyboard.nextInt(); // (1)
        int number2 = keyboard.nextInt(); // (2)
```

```
int sum = number1 + number2;
System.out.printf("The sum of integers %d and %d is %d%n",
                  number1, number2, sum); // (3)
}
}
```

Program output:

```
Enter two numbers [integer integer]: 23 347
The sum of integers 23 and 347 is 370
```

2



Skipping the rest of the line when reading values from the keyboard

The previous example showed how to use the `Scanner` class to read multiple values from the same line. If we instead want to read only *one* value per line, the program can read the first value that is entered, then call the `nextLine()` method in the `Scanner` class to skip the rest of the current line.

Program 2.10 calls the `nextDouble()` method at (1) to read the first value that is entered on the current line. The `nextLine()` method is called at (2) to skip any input remaining in the current line. Even if the user enters two or more values on the line, only the first value will be read by the program, and the rest skipped.

PROGRAM 2.10 Skipping the rest of the current line when reading

```
// Calculating the area of rectangle whose sides are input from the keyboard.
import java.util.Scanner;
public class FloatingPointArea2 {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);

        System.out.println("Area calculation - one value per line");
        System.out.print("Enter the rectangle length [decimal number]: ");
        double length = keyboard.nextDouble(); // (1)
        keyboard.nextLine(); // (2) Empty the rest of the line.
        System.out.print("Enter the rectangle breadth [decimal number]: ");
        double breadth = keyboard.nextDouble();

        double area = length * breadth;
        System.out.printf(
            "A rectangle of length %.2f cm and breadth %.2f cm " +
            "has area %.2f sq. cm.%n",
            length, breadth, area);
    }
}
```



Program output:

```
Area calculation - one value per line
Enter the rectangle length [decimal number]: 15.5 3.9
Enter the rectangle breadth [decimal number]: 4.25
A rectangle of length 15.50 cm and breadth 4.25 cm has area 65.88 sq. cm.
```

Program 2.8 does not use the `nextLine()` method. When running that program, the user can enter both numbers on the same line, as illustrated below:

```
Enter the rectangle length [decimal number]: 15.5 4.25
Enter the rectangle breadth [decimal number]: A rectangle of length 15.50 cm
and breadth 4.25 cm has area 65.88 sq. cm.
```

The program will read the second floating-point value on the first line of input, and assign it to the variable `breadth`. The second prompt and the result of the calculation are then printed on the same line in the terminal window. The program reports the correct answer, but the printout is a bit confusing. If we want our program to read multiple values, but only one value per line, the `nextLine()` method can be used, as illustrated in Program 2.10. We empty the current line by calling `nextLine()` method before reading the next value.

2.7 Review questions

1. What is printed to the terminal window by the method calls below?

```
System.out.println("10+10 is " + 20);
System.out.println("10+10 is " + 10 + 10);
System.out.println(10 + 10 + 20);
```

2. What is a variable? What is a local variable?
3. Write the declaration for an integer variable called `numberOfPoints`, and initialize it to the value 35. What do we call a value that is written directly in the source code?
4. What is a constant? Modify the declaration from Question 2.3 to declare a constant.
5. Which of the following variable names are valid in Java? Which variables have meaningful names? Justify your answer.

- a minimum-Price
- b minimumPrice
- c XYZ
- d xCoordinate
- e y2k
- f isDone
- g numberOfDaysInALeapYear
- h JDK_1_6_0

6. What is a data type? What is a primitive data type?



7. Fill in the blanks.

- a An arithmetic expression can consist of _____ and operands.
- b Multiplication (*) requires _____ operands, and is thus a _____ operator.
- c The operator - in the expression -4 is a _____ operator, while the operator - in the expression 5 - 4 is a _____ operator.

8. The operands in an expression in Java are always evaluated from _____ to _____.

If two operators with different _____ are next to each other in an expression, the operator with the _____ will be evaluated first.

9. If two operators with the same precedence are next to each other in an expression, _____ rules are used to determine which operator will be evaluated first.

The operator - (unary minus) is left-associative and groups from _____ to _____, while the operator - (subtraction) is right-associative and groups from _____ to _____.

10. Evaluate the following expressions. Explain in what order the operators are evaluated in each case.

- a $3 + 2 - 1$
- b $2 + 6 * 7$
- c $-5 + 7 - - 6$
- d $2 + 4 / 5$
- e $5 * 2 - - 3 * 4$
- f $10 / 0$
- g $2 + 4.0 / 5$
- h $10 / 0.0$
- i $4.0 / 0.0$
- j $2 * 4 \% 2$

11. Use the `System.out.printf()` method to print the following values:

- a A six-digit integer, including the sign, e.g. 123456 as +123456.
- b The floating-point value 123456789.3837 in scientific notation, i.e. as $1.234567e+08$.
- c The string "We are 100% motivated to learn Java!".
- d The number 1024 as a right-justified eight-digit integer, i.e. as 00001024.

2.8 Programming exercises

1. Write a program that converts the temperature in degrees Celsius to degrees Fahrenheit. In this version of the program, do the conversion for a particular temperature in degrees Celsius, say 25.5°C . Use the following formula for doing the conversion:

$$\text{fahrenheit} = (9.0 \times \text{celsius})/5.0 + 32.0$$



2. Extend the program in Exercise 2.1 to also calculate the corresponding temperature in degrees Kelvin. The following formula calculates the temperature in degrees Kelvin:

$$kelvin = celsius + 273.16$$
3. Extend the program in Exercise 2.2 to read the temperature value from the keyboard. Print a suitable prompt to the terminal window before reading the value, so that the user knows what type of value is expected.
4. Write a program that calculates the volume v of a cube with dimensions l , b and h , all floating-point values, using the following formula:

$$v = l \times b \times h$$

Print a suitable prompt to the terminal window before reading values from the keyboard using the `Scanner` class.

5. Write a program that calculates the area a and circumference c of a circle:

$$a = \pi \times r^2$$

$$c = 2 \times \pi \times r$$

where π is a mathematical constant, and r is the radius of the circle. The square of the radius can be calculated by multiplying r with itself. Assume a value of 3.1415927 for π , which should be defined as a constant in your program.

Print a suitable prompt to the terminal window before reading a value for the radius from the keyboard.

6. A company pays its salesmen based on commissions on their sales. For example, a salesman that has sold products for 5000 GBP receives a payment of 200 GBP plus 9% of 5000 GBP, i.e. 650 GBP.

The price of the different products are:

- Product A: 239.99 GBP.
- Product B: 129.75 GBP.
- Product C: 99.95 GBP.
- Product D: 350.89 GBP.

Write a program that reads the number of sales for each product in the current week by a salesman, and calculates the salary based on these sales. The program should print a formatted salary slip, including the number of units sold and the unit price of each product sold. The user dialogue and program output could look like this:

Salesman's salary calculation

```
Enter units sold of Product A [integer]: 25
Enter units sold of Product B [integer]: 2
Enter units sold of Product C [integer]: 12
Enter units sold of Product D [integer]: 4
```

Enter provision rate in percent [decimal value]: **9.0**

Salary slip

Product A: 20 units x 239.99 GBP = 4799.80 GBP

Product B: 2 units x 129.75 GBP = 259.50 GBP

Product C: 12 units x 99.95 GBP = 1199.40 GBP

Product D: 4 units x 350.89 GBP = 1403.56 GBP

Total sales = 7662.26 GBP

Fixed part of salary = 200.00 GBP

Provision (9% of sales) = 689.03 GBP

Total salary = 889.03 GBP

2



- 7.** Your company provides IT consultancy services to the public sector, and the financial department in each country wants a program to print invoices for their national customers. Write a program that allows an accountant to specify the regular hours worked, the hourly rate, the number of hours worked overtime and the percentage increase in hourly rate for the overtime. Here is an example of a dialogue between the program and the user:

Invoice preparation for IT ConsultPro Inc.

Your national currency is GBP

Enter the client name [string]: **Orange County**

Enter the regular working hours [decimal number]: **120.0**

Enter the hourly rate [decimal number]: **60.0**

Enter the number of hours worked overtime [decimal number]: **40.0**

Enter the percentage increase in hourly rate [decimal number]: **25.0**

Based on this information, the program should print a formatted invoice:

IT ConsultPro Inc. - Invoice for Orange County

Regular working hours : 120 hours a 60 GBP: 7200 GBP

Overtime work : 40 hours a 75 GBP: 3000 GBP

Total : 10200 GBP

The national currency used should be defined as a constant in the program.

The name of the client company can be read by the following code:

```
Scanner keyboard = new Scanner(System.in);  
String clientName = keyboard.nextLine();
```

Any input on the current line will be read as one string and assigned to the `clientName` variable. This code allows text with multiple words to be read as one string, for example "Orange County".



8. Your multinational company has just started offering IT consultancy to governmental organisations in the European Union, and must provide invoices in Euros (€). Modify the program in Exercise 2.7 to allow the national currency to be used, as well as the exchange rate between the national currency and the Euro.
9. Extend the program in Exercise 2.5 to read the units measurement for the radius, e.g. cm, m or km, as a string. You can use the `nextLine()` method for this purpose, as explained in Exercise 2.7.

Run the program to calculate the circumference of the Earth at equator, when the equatorial radius is 6378.135 km.

10. Write a program that calculates how much an amount deposited in a bank account at $r\%$ annual interest has grown to after one, two and three years. The general formula for computing the amount including interest after n years is:

$$K_n = K_0 \times (1 + r / 100)^n$$

when K_0 is the principle amount. The user dialogue and output from the program could look like this:

Calculation of deposits with interests

```
Enter the initial amount deposited [decimal value]: 10000.00
Enter the interest rate [decimal value]: 4.5
Enter the currency used [string]: USD
```

```
Amount with interest after 1 year : 10450.00 USD
Amount with interest after 2 years: 10920.25 USD
Amount with interest after 3 years: 11411.66 USD
```

Program control flow

LEARNING OBJECTIVES

By the end of this chapter you will understand the following:

- Representing truth (Boolean) values in Java.
- Comparing values using relational operators.
- Evaluation of expressions with logical operators.
- Selecting statements to execute: **if**, **if-else**.
- Executing statements repeatedly using loops: **while**, **do-while**.
- Verifying expected program properties with assertions.

INTRODUCTION

Until now we have written programs where the statements are executed sequentially, i.e. one at a time in the order in which they are written in the source code. This is not very flexible, as we would like to control the order in which statements are executed, allowing us to express more complex logic in our programs.

This chapter introduces selection statements and loops for controlling the flow of execution. Selection statements enable us to define multiple actions, where the selection of an action to be executed is dependent on a condition being satisfied. Loops enable us to execute the same statements repeatedly, dependent on a condition being satisfied.

Conditions controlling the execution flow are specified as Boolean expressions, which are constructed using relational and logical operators. Such expressions are also used to define assertions, which allow us to check program properties during execution.

3.1 Boolean expressions

Boolean primitive data type

The primitive data type `boolean` in Java defines two *Boolean values*, represented by the literals `true` and `false`. As with other primitive data types, we can declare variables of the type `boolean` and assign Boolean values to them:

```
boolean itemIsOnSale = true;
```

By testing the value in the Boolean variable `itemIsOnSale`, we can determine whether to give a discount on an item or not. As we shall see, Boolean values facilitate decision making over the actions that should be executed during program execution.

Relational operators

3



A *relational operator* enables us to compare values. Table 3.1 gives an overview of the relational operators found in Java. The following expression in Java is a *Boolean expression*:

```
numHours >= 37.5
```

It evaluates to `true` if the value of the variable `numHours` is greater than or equal to `37.5`, otherwise it evaluates to `false`.

Since a Boolean expression evaluates to a Boolean value, we can assign its value to a Boolean variable:

```
boolean workedOvertime = numHours >= 37.5;
```

Since relational operators have *higher* precedence than the assignment operator `=`, the statement is executed as one would expect: first the Boolean expression and then the assignment.

The operands of a relational operator can be any arithmetic expression:

```
heads + tails == tosses
```

Since arithmetic operators have *higher* precedence than relational operators, the Boolean expression executes as follows:

```
((heads + tails) == tosses) // test for equality
```

I.e. we compare the sum of two variables on the left-hand side with the value of variable on the right-hand side. The `==` operator compares for equality, i.e. if both operands have the same value. Note that this operator is written as `==`, with no space in between. It is a common mistake to mix the equality operator with the assignment operator (`=`):

```
((heads + tails) = tosses) // ERROR!
```

In this case, the compiler reports an error:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:  
  The left-hand side of an assignment must be a variable  
  at TestBooleanError.main(TestBooleanError.java:8)
```

TABLE 3.1 Relational operators in Java

Operator	Meaning
<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code><</code>	Less than
<code>></code>	Greater than
<code><=</code>	Less than or equal to
<code>>=</code>	Greater than or equal to

Understanding relational operators

We can calculate a series of Boolean expressions and print their values. We can also print what we expect the values to be. Program 3.1 shows a program that does this comparison.

3



PROGRAM 3.1 Calculating Boolean expressions with relational operators

```
// Testing relational operators in boolean expressions.
public class TestRelationalOperators {
    public static void main(String[] args) {
        // Tests for integers.
        System.out.printf("Expression      Expected      Calculated%n");
        System.out.printf("%-20s%-12s%-12s%n", "3 == 3", true, (3 == 3));
        System.out.printf("%-20s%-12s%-12s%n", "3 != 3", false, (3 != 3));
        System.out.printf("%-20s%-12s%-12s%n", "7 > 4", true, (7 > 4));
        System.out.printf("%-20s%-12s%-12s%n", "7 < 4", false, (7 < 4));
        System.out.printf("%-20s%-12s%-12s%n", "6 <= 6", true, (6 <= 6));
        System.out.printf("%-20s%-12s%-12s%n", "6 >= 6", true, (6 >= 6));
    }
}
```

Program output:

Expression	Expected	Calculated
<code>3 == 3</code>	true	true
<code>3 != 3</code>	false	false
<code>7 > 4</code>	true	true
<code>7 < 4</code>	false	false
<code>6 <= 6</code>	true	true
<code>6 >= 6</code>	true	true

The printout from the program allows us to compare the calculated and expected values for the Boolean expressions. The print methods of the `System.out` object are useful for printing results to test that our program behaves as expected.

BEST PRACTICES

Do not confuse the assignment operator (`=`) with the equality operator (`==`).

Logical operators

We can combine Boolean expressions by means of *logical operators*, creating new Boolean expressions:

`boolean payBonus = workedOvertime || salesAboveAverage;`

The expression above will be evaluated to `true` if the employee has worked overtime or sold more than the average sales during the week, or if both conditions are satisfied. Table 3.2 shows the logical operators defined in Java.

TABLE 3.2 Logical operators in Java

Operator	Meaning
<code>!</code>	Negation, results in inverting the truth value of the operand, i.e. <code>!true</code> evaluates to <code>false</code> , and <code>!false</code> evaluates to <code>true</code> .
<code>&&</code>	Conditional And, evaluates to <code>true</code> if both operands have the value <code>true</code> , and <code>false</code> otherwise.
<code> </code>	Conditional Or, evaluates to <code>true</code> if one or both operands have the value <code>true</code> , and <code>false</code> otherwise.

BEST PRACTICES

Avoid using Boolean variables that must be negated to serve as conditions in selection statements and loops. An expression such as `payBonus` is easier to understand than `!noBonusPaid`.

Precedence for logical operators

The *precedence of the logical operators* is as follows: the negation operator `!` has higher precedence than the conditional And operator `&&`, which in turn has higher precedence than the conditional Or operator `||`. Given three Boolean variables `b1`, `b2` and `b3`, the following expression:

`b1 || !b2 && b3`

will be interpreted as:

```
(b1 || (!b2) && b3))
```

The *associativity rules for logical operators* are as follows: The negation operator ! associates from right to left. The conditional And operator && and the conditional Or operator || both associate from left to right.

As a consequence of the evaluation rules, the variable b2 above will be associated with the negation operator !, and the expression (!b2) forms one of the operands to the conditional And operator &&. Because of the precedence rules, the conditional And operator && is applied before the conditional Or operator ||.

We can quickly verify this by writing a short program containing the following statements:

```
boolean b1 = false, b2 = false, b3 = true;  
System.out.printf("b1 || !b2 && b3 evaluates to %s%n", b1 || !b2 && b3);  
System.out.printf("(b1 || (!b2) && b3)) to %s%n", (b1 || (!b2) && b3));
```

3



and checking that the same Boolean value is printed by each `printf()` method call.

The logical operators have *lower precedence* than the relational operators. Thus, in the expression

```
weekday >= 6 || weekday == 3
```

the simple Boolean expression (`weekday >= 6`) is evaluated first, followed by the evaluation of the second simple expression (`weekday == 3`), and finally the logical operator || is applied to the resulting Boolean values of the two expressions. Assuming that the variable `weekday` represents the number of a weekday (1 for Monday, 2 for Tuesday, and so on), this test will return the value `true` for the weekdays Wednesday, Saturday and Sunday, and `false` otherwise.

The source code below allows us to verify that this holds:

```
int weekday = 4;  
System.out.printf("Weekday number is %d%n", weekday);  
System.out.printf("Wednesday, Saturday or Sunday: %s%n",  
    weekday >= 6 || weekday == 3);
```

The printout from these lines will be:

```
Weekday number is 4  
Wednesday, Saturday or Sunday: false
```

Combining the tests of precedence shown in this subsection into a workable program is left as an exercise.

Short-circuit evaluation

The operands in a Boolean expression are normally evaluated from left to right. However, the evaluation of a Boolean expression involving the logical operators will end as soon as the value of the expression can be determined. This is called *short-circuit evaluation*. For example, the expression:

`(4 == 3) && (3 < 4)`

will first evaluate to:

`false && (3 < 4)`

As the conditional And operator `&&` will always return `false` if one of its operands has the value `false`, the final value of the expression can be determined without having to evaluate the second operand. Similarly, the Boolean expression:

`(4 > 3) || (5 < 4)`

first evaluates to `true || (5 < 4)`. Thereafter, the evaluation stops, because the final value of the Boolean expression must be `true`, regardless of the value of the second operand to the conditional Or operator `||`.

De Morgan's laws

3



In some cases, complicated expressions can be simplified by means of De Morgan's laws:

- 1 `!(b1 && b2)` is equivalent to `(!b1 || !b2)`
- 2 `!(b1 || b2)` is equivalent to `(!b1 && !b2)`

when `b1` and `b2` both are Boolean expressions.

If we, for example, want to determine whether an integer value `t` falls inside a given interval with lower limit `a` and upper limit `b`, we can write this as `t >= a && t <= b`, or as `!(t < a || t > b)`, using De Morgan's second law. Both of these expressions will be evaluated to the same Boolean value for all values of `a`, `b` and `t`. Thus, we can choose the form we find easier to understand.

For example, whether a purchase is neither cheap nor good can be formulated as `(!cheap || !good)` or as `!(cheap && good)`. According to De Morgans first law, these two Boolean expressions are equivalent.

Using Boolean expressions to control flow of execution

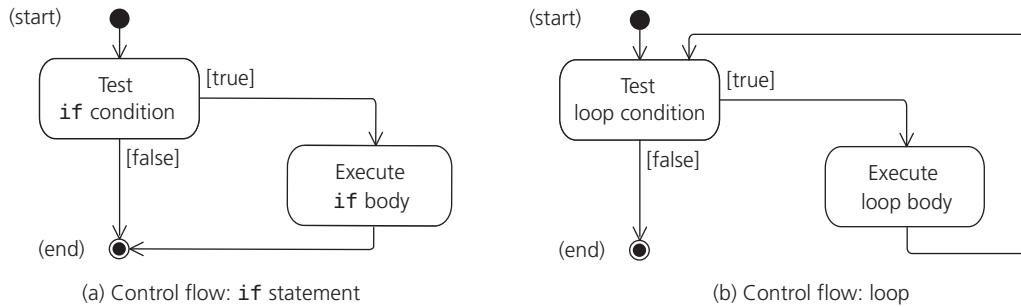
We often want to perform different actions depending on whether a given condition is satisfied. This condition can be formulated as a Boolean expression, and its value can then be used to determine which actions should be executed.

Let's look at how we can calculate the salary for an employee, given the number of hours the employee has worked during the week. Many employees will receive a fixed salary as long as they work the normal number of hours during a week, with additional payment for any overtime. We need a statement that allows us to specify a condition that will determine any overtime, and a corresponding action to compute the additional salary. This type of statement is called a *selection statement*.

Other types of problems require that certain actions be executed repeatedly. For example, to compute the salary for *all* employees in a company, we need repeatedly to execute the actions for calculating the salary of an employee. This type of problem can be solved by a *loop* statement, allowing us to repeat actions a certain number of times based on the value of a condition.

Selection and repetition statements allow us to control which actions are performed by the program. These statements therefore determine the *control flow* of the program. Figure 3.1a shows the control flow determined by a selection statement called the *if* statement. This statement executes an action, the *if body*, if the given condition is satisfied. Figure 3.1b illustrates the control flow determined by a type of loop called the *pre-test loop*, where the condition is tested before the *loop body* is executed. While the *if* statement only allows its body to be executed at most once, the pre-test loop allows the loop body to be executed multiple times as long as the *loop condition* is satisfied.

FIGURE 3.1 Control flow in selection statements and loops

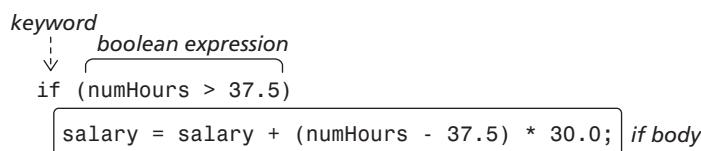


3.2 Control flow: selection statements

Simple selection statement: *if*

A *simple selection statement* performs an action if a given condition is satisfied. The condition is written as a Boolean expression, as illustrated in Figure 3.2. If the Boolean expression evaluates to *true*, i.e. the condition is satisfied, the action in the *if body* is executed. On the other hand, if the expression evaluates to *false*, the action in the *if body* is skipped, and the execution continues after the *if* statement.

FIGURE 3.2 Simple selection statement



Note that in Figure 3.2 we have used indentation to indicate which statement constitutes the *if body*. This is a useful programming practice, but has no impact on how the compiler translates the *if* statement. The compiler will treat the first statement after the Boolean expression as the *if body*, regardless of how the source code is formatted. Consistent indentation, however, makes the source code easy to read.



Program 3.2 uses the simple selection statement to calculate the weekly salary for an employee. A person working thirty-seven and a half hours (or less) receives a basic salary of 750 USD. If they work overtime, they receive an additional 30 USD per hour for the extra time worked.

At (1), the variable `salary` is initialized to the basic salary. The condition to determine whether there is any overtime, `numHours > 37.5`, is evaluated at (2). If the condition is satisfied, salary for overtime is calculated and added at (3). Finally, at (4), the weekly salary is printed to the terminal window.

PROGRAM 3.2 Calculating salary using a simple selection statement

```
// Calculating weekly salary, version 1.  
import java.util.Scanner;  
public class Salary1 {  
    public static void main(String[] args) {  
        final double NORMAL_WORKWEEK = 37.5;  
  
        // Read the number of hours worked this week.  
        Scanner keyboard = new Scanner(System.in);  
        System.out.print("Enter the number of hours worked [decimal number]: ");  
        double numHours = keyboard.nextDouble();  
  
        // Calculate the weekly salary and print it to the terminal window.  
        double salary = 750.0; // (1) weekly salary  
        if (numHours > NORMAL_WORKWEEK) // (2)  
            salary = salary + (numHours - NORMAL_WORKWEEK) * 30.0; // (3)  
        System.out.printf("Salary for %.1f hours is %.2f USD\n",  
                         numHours, salary); // (4)  
    }  
}
```

Program output:

```
Enter the number of hours worked [decimal number]: 39.5  
Salary for 39.5 hours is 810.00 USD
```

Blocks of statements: { ... }

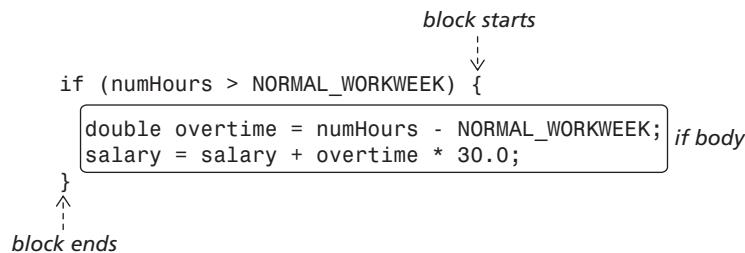
A sequence of statements can be enclosed in curly brackets, {}, to specify a *block of statements*. Such a block is called a *compound statement*, and can be used anywhere that a single statement can be used.

The use of such a block is illustrated in Figure 3.3, where the `if` body is specified as a block, allowing us to execute more than one action when the `if` condition is satisfied. If we had left out the block notation, only the first statement after the Boolean expression, i.e.

```
double overtime = numHours - NORMAL_WORKWEEK;
```

would constitute the `if` body. However, the compiler will report an error in the next statement, as the `overtime` variable is not accessible outside the `if` body.

FIGURE 3.3 A block of statements



Local variables in a block

3



We can define new variables inside a block, as shown in Figure 3.3. Such local variables can only be accessed inside the block.

The part of the program where such a variable can be accessed is called its *scope*. The scope of a variable defined within a block is from its declaration in the block to the end of the block. When a variable is no longer accessible, we say that it is *out of scope*. The compiler will produce an error message if we try to access local variables outside their scope. This is the case for the local variable `overtime` at (1) in Program 3.3, if we were to remove the comment designator `//` at the beginning of the line.

PROGRAM 3.3 Variables defined inside a block

```
// Calculating weekly salary, version 1b.  
import java.util.Scanner;  
public class Salary1b {  
    public static void main(String[] args) {  
        final double NORMAL_WORKWEEK = 37.5;  
  
        // Read the number of hours worked this week.  
        Scanner keyboard = new Scanner(System.in);  
        System.out.print("Enter the number of hours worked [decimal number]: ");  
        double numHours = keyboard.nextDouble();  
  
        // Calculate the weekly salary and print it to the terminal window.  
        double salary = 750.0; // (1) weekly salary  
        if (numHours > NORMAL_WORKWEEK) { // if body is a block  
            double overtime = numHours - NORMAL_WORKWEEK; // local variable  
            salary = salary + overtime * 30.0;  
        }  
        System.out.printf("Salary for %.1f hours is %.2f USD%\n",  
                         numHours, salary);  
        // System.out.printf("Number of hours overtime: %.1f%n", overtime); // (1)
```

```
    }  
}
```

Program output:

```
Enter the number of hours worked [decimal number]: 39.5  
Salary for 39.5 hours is 810.00 USD
```

3



Selection statement with two choices: **if-else**

We often need to choose between two alternative actions, where the program will execute one alternative if a given condition is satisfied, and the other if the condition is not satisfied. Java offers an **if-else** statement for this purpose. Figure 3.4 shows an example of this type of selection statement.

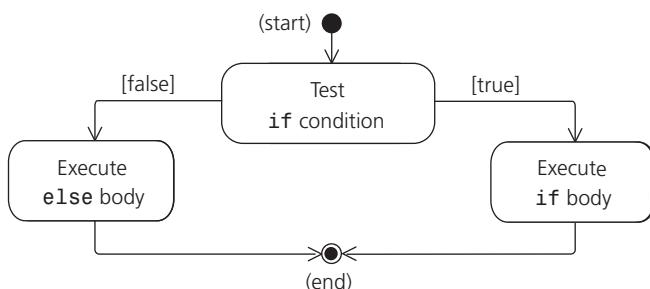
FIGURE 3.4 A selection statement with two alternatives

```
keyword  
↓  
boolean expression  
if (numHours <= NORMAL_WORKWEEK) {  
    salary = FIXED_SALARY;  
} else {  
    salary = FIXED_SALARY + (numHours - NORMAL_WORKWEEK) * 30.0;  
}  
else body
```

The first part of an **if-else** statement is the same as a simple **if** statement. The *if body* is executed if the Boolean expression evaluates to **true**, and the *else body* is executed otherwise. The keyword **else** starts the *else body*.

Figure 3.5 illustrates the execution of an **if-else** statement. Depending on the value of the Boolean expression, one of the two alternative actions is executed.

FIGURE 3.5 Executing a selection statement with two alternatives



Program 3.4 solves the same problem as Program 3.2, but using an **if-else** statement, where the *if body* calculates the basic salary for an employee who has not worked

overtime, while the *else body* calculates the basic salary plus overtime compensation, for an employee who has worked overtime.

The condition for payment of basic salary, i.e. that the employee has worked 37.5 hours or less, is defined at (1). If this condition is satisfied, the variable `salary` is assigned the basic payment at (2). The keyword `else` at (3) marks the start of the alternative action, that calculates the weekly salary for an employee who has worked overtime, as shown at (4).

PROGRAM 3.4 Selecting alternative actions

```
// Calculating weekly salary, version 2.  
import java.util.Scanner;  
public class Salary2 {  
    public static void main(String[] args) {  
        final double NORMAL_WORKWEEK = 37.5;  
        final double FIXED_SALARY = 750.0;  
  
        // Read the number of hours worked this week.  
        Scanner keyboard = new Scanner(System.in);  
        System.out.print("Enter the number of hours worked [decimal number]: ");  
        double numHours = keyboard.nextDouble();  
  
        // Calculate the weekly salary and print it to the terminal window.  
        double salary = 0.0; // weekly salary  
        if (numHours <= NORMAL_WORKWEEK) { // (1)  
            salary = FIXED_SALARY; // (2) if body  
        } else { // (3)  
            salary = FIXED_SALARY +  
                (numHours - NORMAL_WORKWEEK) * 30.0; // (4) else body  
        }  
        System.out.printf("Salary for %.1f hours is %.2f USD\n",  
                         numHours, salary);  
    }  
}
```

3



Running Program 3.3 with the same input as Program 3.1 yields:

```
Enter the number of hours worked [decimal number]: 39.5  
Salary for 39.5 hours is 810.00 USD
```

Program output for an employee who has worked overtime:

```
Enter the number of hours worked [decimal number]: 42.5  
Salary for 42.5 hours is 900.00 USD
```

Nested selection statements

In Program 3.5 the selection statement at (1) has a new selection statement at (5) nested in its *else body*. It is only executed if the employee has worked overtime. Note that

Program 3.5 gives the same result as Program 3.6 when run with the same input.

PROGRAM 3.5 Calculating payment with a nested selection statement

```
// Calculating weekly salary, version 3.  
import java.util.Scanner;  
public class Salary3 {  
    public static void main(String[] args) {  
        final double NORMAL_WORKWEEK = 37.5;  
        final double FIXED_SALARY = 750.0;  
  
        // Read the number of hours worked this week.  
        Scanner keyboard = new Scanner(System.in);  
        System.out.print("Enter the number of hours worked [decimal number]: ");  
        double numHours = keyboard.nextDouble();  
  
        // Calculate the weekly salary and print it to the terminal window.  
        double salary = 0.0;  
        if (numHours <= NORMAL_WORKWEEK) {           // (1) if statement  
            salary = FIXED_SALARY;                     // (2) if body  
        } else {                                     // (3) else body  
            salary = FIXED_SALARY + (numHours - NORMAL_WORKWEEK) * 30.0; // (4)  
            if (numHours > 42.0) {                   // (5) nested if statement  
                salary = salary + 100.0;             // (6)  
            }  
        }                                              // (7)  
        System.out.printf("Salary for %.1f hours is %.2f USD%n",  
                           numHours, salary);  
    }  
}
```

Program output for Program 3.2:

```
Enter the number of hours worked [decimal number]: 35.5  
Salary for 35.5 hours is 750.00 USD
```

Program output for Program 3.4:

```
Enter the number of hours worked [decimal number]: 39.5  
Salary for 39.5 hours is 810.00 USD
```

Program output for Program 3.5:

```
Enter the number of hours worked [decimal number]: 42.5  
Salary for 42.5 hours is 1000.00 USD
```

When nesting selection statements, we must be careful to ensure that the program logic is correct. Using block statements may be necessary explicitly to specify which actions are to be executed when selection statements are nested. In the following code, the *else part* is associated with the *if part at (2)*:

```

if (condition1)          // (1)
    if (condition1)      // (2)
        System.out.println("nested if");
    else                 // Associated with (2)
        System.out.println("else body");

```

In the following code, the `else` part is associated with the `if` part at (1):

```

if (condition1) {           // (1)
    if (condition1)         // (2)
        System.out.println("nested if");
} else                      // Associated with (1)
    System.out.println("else body");

```

Note that the `else body` is always attached to the *nearest unattached if* part.

BEST PRACTICES

Always enclose both the `if` body and the `else` body of an `if-else` statement in a block, even if they just contain a single statement. This makes the program easier to understand, and reduces the chance for errors should the `if` statement be modified.

3



Chaining if-else statements

The `else body` in an `if-else` statement can be another `if-else` statement, as shown in Figure 3.6. Here, the second `if-else` statement will be executed as *one* statement if the first Boolean expression, `numHours <= 37.5`, evaluates to `false`. If this is the case, the second Boolean expression, `numHours <= 42.0`, will be evaluated, and one of the alternative actions of the second selection statement will be executed.

FIGURE 3.6 Chaining if-else statements

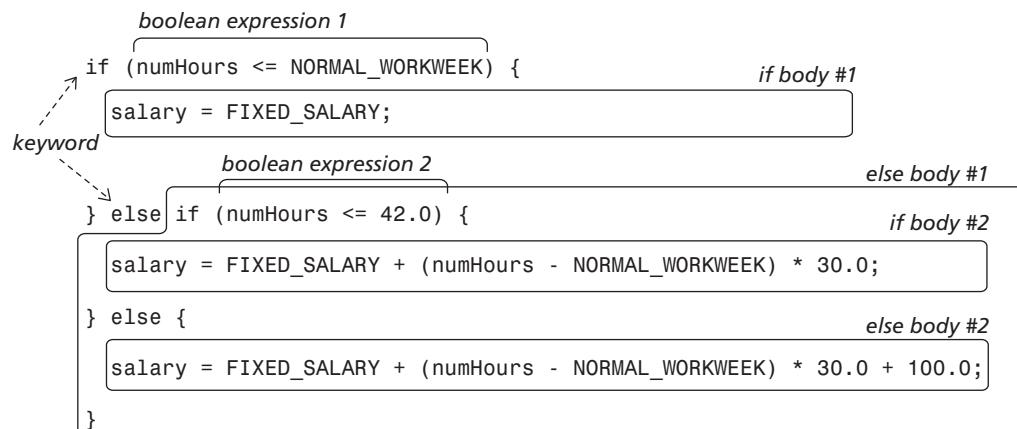
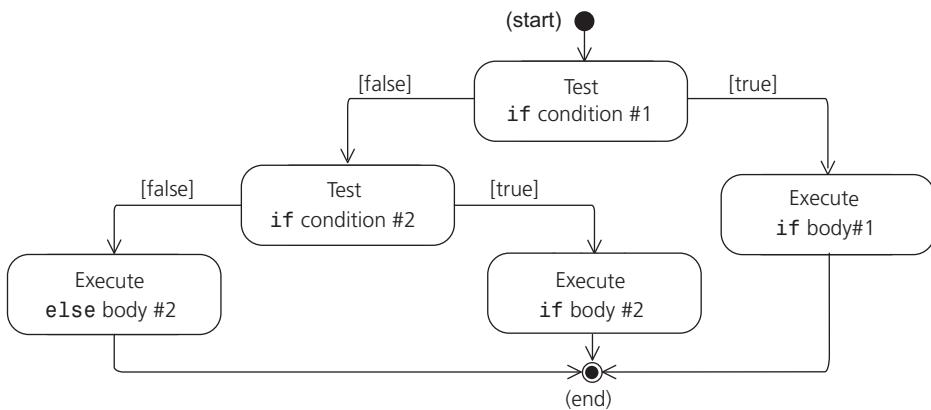


Figure 3.7 shows the execution of chained `if-else` statements. We can repeatedly extend the chain by replacing the last `else` body with another `if-else` statement.

FIGURE 3.7 Execution of chained `if-else` statements



3



Program 3.6 uses a chain of `if-else` statements to extends the salary calculation from the previous examples, by allocating an additional payment of 100 USD for employees who have worked more than forty-two hours a week.

First, the Boolean expression (`numHours <= 37.5`) at (1) is evaluated. If this condition is satisfied, the statement at (2) is executed. On the other hand, if this condition is not satisfied, the Boolean expression (`numHours <= 42.0`) at (3) is evaluated. If this condition evaluates to true, the statement at (4) is executed. Otherwise, the statement at (5), in the last `else` body, is executed.

PROGRAM 3.6 Calculating salary with a chain of `if-else` statements

```
// Calculating weekly salary, version 4.  
import java.util.Scanner;  
public class Salary4 {  
    public static void main(String[] args) {  
        final double NORMAL_WORKWEEK = 37.5;  
        final double FIXED_SALARY = 750.0;  
  
        // Read the number of hours worked this week.  
        Scanner keyboard = new Scanner(System.in);  
        System.out.print("Enter the number of hours worked [decimal number]: ");  
        double numHours = keyboard.nextDouble();  
  
        // Calculate the weekly salary and print it to the terminal window.  
        double salary = 0.0;  
        if (numHours <= NORMAL_WORKWEEK) { // (1)  
            salary = FIXED_SALARY; // (2)  
        } else if (numHours <= 42.0) { // (3)
```

```

        salary = FIXED_SALARY + (numHours - NORMAL_WORKWEEK) * 30.0; // (4)
    } else {
        salary = FIXED_SALARY +
            (numHours - NORMAL_WORKWEEK) * 30.0 + 100.0;           // (5)
    }
    System.out.printf("Salary for %.1f hours is %.2f USD\n",
                      numHours, salary);
}
}

```

Testing alternative 1:

Enter the number of hours worked [decimal number]: 35.5
 Salary for 35.5 hours is 750.00 USD

Testing alternative 2:

Enter the number of hours worked [decimal number]: 39.5
 Salary for 39.5 hours is 810.00 USD

Testing alternative 3:

Enter the number of hours worked [decimal number]: 42.5
 Salary for 42.5 hours is 1000.00 USD

3



3.3 Control flow: loops

A loop statement can be used to execute an action repeatedly. The action is specified in the *loop body*. The action can consist of zero or more statements. Each execution of the loop body is called an *iteration*.

The loop executes the loop body for as long as a given loop condition is satisfied. The condition is specified as a Boolean expression.

Loop test before loop body: **while**

In a while statement the loop condition is tested *before* the loop body is executed. That is why this kind of loop called a *pre-test loop*. Figure 3.8 illustrates the syntax of while loop.

FIGURE 3.8 Example of a **while** loop

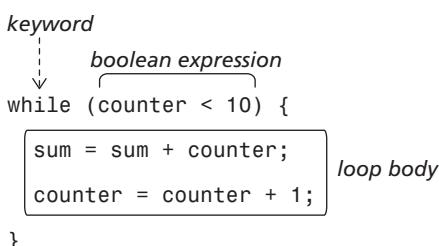
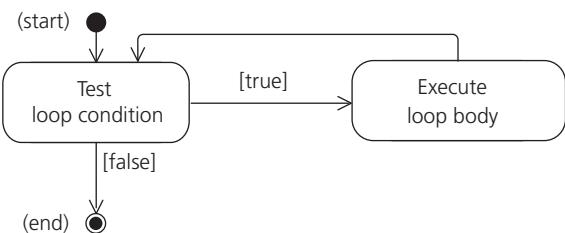


Figure 3.9 shows how a `while` loop is executed. First the loop condition, which is a Boolean expression, is evaluated. If the expression evaluates to `true`, the loop body is executed. After each iteration, the loop condition is tested. The execution of the loop body repeats until the loop condition evaluates to `false`. If the condition is not satisfied when control enters the loop for the first time, the loop body is skipped altogether.

FIGURE 3.9 Execution of a pre-test loop



3



Loop test after loop body: `do-while`

A `do-while` loop evaluates the loop condition *after* the loop body has been executed. That is why it is often called a *post-test loop*. The syntax of this loop is illustrated in Figure 3.10. Since this is a post-test loop, the loop body of a `do-while` loop is executed *at least once*, as is evident from Figure 3.11.

FIGURE 3.10 Example of a `do-while` loop

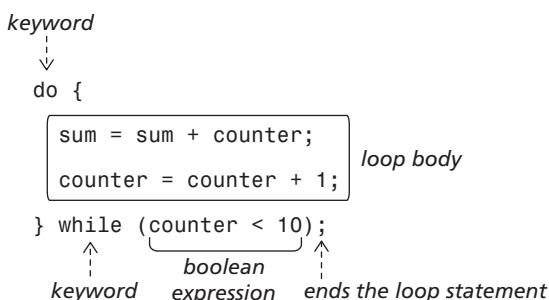
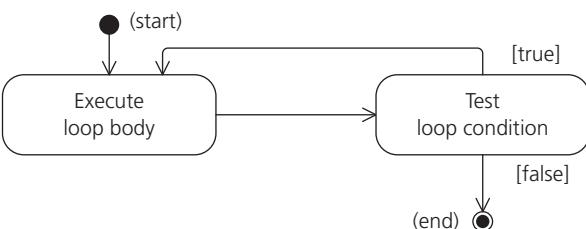


FIGURE 3.11 Execution of a post-test loop





Infinite loops

The execution of a loop body must at some point affect the loop condition such that it evaluates to `false`, otherwise the loop will never terminate. A loop that never terminates is called an *infinite loop*. Should an infinite loop occur in a program, the program will need to be terminated explicitly. On most platforms pressing the key combination `Ctrl-C` terminates program execution and thus the infinite loop.

Some examples of infinite loops are given in Section 5.5 on page 110.

Using loops to implement user dialogue

Loops are often used to implement dialogue between the user and the program, in which the program repeatedly asks the user to enter values. The number of values that will be entered is either read first, or a specific value, called the *sentinel value*, is used to mark the end of input. This allows the user to control the number of loop iterations. Figure 3.8 and Figure 3.10 are both examples in which the user enters a value controlling the number of iterations of the loop, while Exercise 3.5 on page 69 shows an example that uses a sentinel value to control the loop.

Program 3.7 calculates the sum of integers entered by the user. Again, we use the `Scanner` class to read the user input. A `Scanner` object is created, and connected to the keyboard with the following statement:

```
Scanner keyboard = new Scanner(System.in);
```

At (1) in Program 3.7 the number of values to add is then read from the keyboard. The program uses this value to control the execution of the loop at (2). Each new number is added to the accumulated sum of the numbers read so far. The reading of numbers is done by the `while` loop at (2), where the condition (`numberCounter < totalNumbers`) controls the execution of this loop. If the condition evaluates to `true`, the loop body is executed, reading the next number to add, updating the sum so far, and incrementing the counter for the number of values read so far. After each iteration, the condition is tested again. The testing of the loop condition and execution of the loop body is repeated until the condition evaluates to `false`. The loop is guaranteed to terminate, as the value of the number counter increases after each iteration, and will eventually become equal to the total numbers of values to read. After the loop terminates, the sum of the numbers read is printed to the terminal window at (4).

PROGRAM 3.7 Adding a sequence of integers using loops

```
// Adding a series of integers read from the keyboard.
import java.util.Scanner;
public class IntegerAddition {
    public static void main(String[] args) {

        Scanner keyboard = new Scanner(System.in);
        System.out.print("Enter the number of integers to add [integer]: ");
        int totalNumbers = keyboard.nextInt();      // (1) No. of integers to add
        keyboard.nextLine();                      // Skip rest of input
```

```

int numberCounter = 0;                                // Numbers read so far
int sum = 0;                                         // Sum of numbers so far

while (numberCounter < totalNumbers) {                // (2)
    System.out.print("Enter the next number [integer]: ");
    int nextInteger = keyboard.nextInt(); // Read the next number
    keyboard.nextLine();
    sum = sum + nextInteger;
    numberCounter = numberCounter + 1;
}                                                       // (3)

System.out.printf("The sum of %d integers is %d%n",
                  numberCounter, sum);                   // (4)
}
}

```

3



Program output:

```

Enter the number of integers to add [integer]: 4
Enter the next number [integer]: 12
Enter the next number [integer]: 34
Enter the next number [integer]: 4
Enter the next number [integer]: -2
The sum of 4 integers is 48

```

Choosing the right loop

Which form of the loop to choose depends on the problem. A rule of the thumb is that, if the loop body needs to be executed at least once, a `do-while` loop is preferred. A `while` loop is the best choice if the loop body should not be executed for some input cases. Both forms can often be used to provide a solution, and choosing between the two forms becomes more a matter of taste.

Some problems lend themselves more naturally to one form of loop than the other. For example, in Program 3.7 we could have used a `do-while` loop to ensure that the user did not enter a negative value for the number of values to read:

```

int totalNumbers = keyboard.nextInt();                      // (1)
keyboard.nextLine(); // skip rest of input line

while (totalNumbers < 0) { // read again until a valid count is given
    totalNumbers = keyboard.nextInt();
    keyboard.nextLine();
}

int numberCounter = 0;                                     // Numbers read so far
int sum = 0;                                              // Sum of numbers so far

```

Nested loops

A loop body can contain another loop as one of its statements. We then have *nested loops*: an *outer loop* and an *inner loop*. Program 3.8 prints a multiplication table from 1 to 10 using two nested `while` loops. We note that in the program, for each iteration of the outer loop, the inner loop executes 10 iterations.

PROGRAM 3.8 Nested loops

```
// Printing a multiplication table using nested loops.  
public class NestedLoops {  
    public static void main(String[] args) {  
        int number = 1, limit = 10;  
        while (number <= limit) { // Outer loop  
            int times = 1;  
            while (times <= limit) { // Inner loop  
                int product = number * times;  
                System.out.println(number + " x " + times + " = " + product);  
                times = times + 1;  
            }  
            number = number + 1;  
        }  
    }  
}
```

Partial program output:

```
1 x 1 = 1  
1 x 2 = 2  
1 x 3 = 3  
...  
1 x 10 = 10  
2 x 1 = 2  
2 x 2 = 4  
2 x 3 = 6  
...  
10 x 8 = 80  
10 x 9 = 90  
10 x 10 = 100
```



BEST PRACTICES

Use print statements inside loops to check that the variables controlling loop termination are being changed as expected. Using print statements to verify the state of the program during execution is a useful debugging technique.

3.4 Assertions

Making assertions

3



We can make sure that a program satisfies a certain assumption about its properties at a given point in its source code by defining an *assertion*. The assert statement allows us to specify an assertion about the program's behaviour as a Boolean expression. This expression is evaluated during program execution. If the expression evaluates to false, an error message is generated and execution is aborted. If the expression evaluates to true, execution continues after the assert statement. We can, for example, check that the variable `width` holds a positive value by formulating the following assertion:

```
assert width > 0.0;
```

This assertion will lead to program termination if the variable `width` does not hold a positive value when the `assert` statement is executed.

To provide more information about the cause of assertion failure, explanatory text can be specified after the Boolean expression:

```
assert width > 0.0 : "The width of the rectangle must be > 0.0";
```

If the variable `width` holds a non-positive value when the `assert` statement is executed, the program will terminate with an error message similar to the following:

```
> javac FloatingPointArea3.java
> java -ea FloatingPointArea3
Enter the rectangle length [decimal number]: 12.5
Enter the rectangle width [decimal number]: -2.5
Exception in thread "main" java.lang.AssertionError:
    The width of the rectangle must be > 0.0
        at FloatingPointArea3.main(FloatingPointArea3.java:16)
```

The printout gives information about where the assertion is defined (the `main()` method in the `FloatingPointArea3` class), the name of the source code file (`FloatingPointArea3.java`) and the line number in this file (16). Note that the flag “`-ea`” must be specified on the command line when the program is run. This option turns on the validation of assertions in the program during execution (see Appendix G).

Assertions as a testing technique

Program 3.9 uses assert statements at (1) and (2) to *validate user input* before the values are used in calculations. The area is only calculated if both the length and the width are positive values.

The assertions at (1) and (2) in Program 3.9 can be combined as follows:

```
assert width > 0.0 && length > 0.0;
```

We can also specify an explanatory text to be printed if the assertion fails:

```
assert width > 0.0 && length > 0.0 : "Rectangle width and length must be > 0.0";
```

In previous code examples we have only controlled the value of variables. However, the Boolean expression in an assertion can be more complicated, but it must not have *side-effects*, meaning that its execution must not, for example, change the value of variables in the program. If it did, the execution of the program would depend on the assertions, and that would be inappropriate, as assert statements are only executed if the program is run with the “-ea” option.

3



Assertions provide a useful testing technique that can help us detect errors early, and well before a program is delivered to the user. Assertions can be turned on when running the program for test purposes, and turned off when the program is shipped to the user. Should problems occur after delivery, the assertions can be turned on again by means of the “-ea” flag, enabling problems to be detected when running in the user environment.

PROGRAM 3.9 Using assertions to validate user input and computed results

```
// Using assertions to verify user input and calculated values.
import java.util.Scanner;
public class FloatingPointArea3 {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);

        // Read rectangle dimensions
        System.out.print("Enter the rectangle length [decimal number]: ");
        double length = keyboard.nextDouble();
        keyboard.nextLine();
        System.out.print("Enter the rectangle width [decimal number]: ");
        double width = keyboard.nextDouble();

        // Validate user input
        assert length > 0.0 : "The length of the rectangle must be > 0.0";// (1)
        assert width > 0.0 : "The width of the rectangle must be > 0.0"; // (2)

        double area = length * width; // Calculate area of the rectangle

        // Print the correct answer
        System.out.printf(
            "A rectangle of length %.2f cm. and width %.2f cm. has" +
            " area %.2f sq. cm.%n",
            length, width, area);
    }
}
```

```
    length, width, area);  
}  
}
```

BEST PRACTICES

Use assertions to check that variables do not contain unexpected values, before they are used.

3



3.5 Review questions

1. A Boolean expression can only evaluate to one of two values, _____ or _____. The value of such an expression can be assigned to a _____ variable.
2. Name the two types of operators that can be used in Boolean expressions.
3. Which of the following assignments are allowed in Java, when the variables `test1`, `test2` and `test3` all are of data type `boolean`? Compute the value of all valid expressions.
 - a `test1 = 2 < 3;`
 - b `test1 = 2 + 3 < 1 - - 5;`
 - c `test1 = "true";`
 - d `test2 = false;`
4. What is the value of the expression `b1 && !b2 || !b1 == b3`, when the Boolean variables `b1`, `b2` and `b3` have the values `true`, `false` and `true`, respectively?
5. Selection and loop statements offer two mechanisms for controlling the _____. Such statements enable us to _____ alternative actions, or to _____ an action a _____ or _____ number of times.
6. What is the value of the variable `test` at (1) to (4)? Are all assignments valid in Java?

```
boolean test;  
int numMen = 12;  
int numWomen = 7;  
test = numMen < numWomen;           // (1)  
test = numMen = numWomen;          // (2)  
test = numMen * numWomen >= 90; // (3)  
test = 2 * numWomen < numMen;   // (4)
```

7. The Java operators _____ (negation), _____ (conditional And), and _____ (conditional Or) expect operands of type _____.
8. To what values are the following Boolean expressions evaluated, when the value of the Boolean variables `b1`, `b2` and `b3`, are `true`, `false` and `true`, respectively?



- a** `b3 || 3 * 0.5 < 2 + 3 * -1`
- b** `4 + 1 == 8 - 3 && b`
- c** `5 < 4 || !b2`
- d** `b1 && !b2 || b3`
- e** `(b2 || 2 < 3) && (!b3 || b1)`
- f** `! ((b1 && b3) || (b2 != b1))`

9. Use De Morgan's laws to simplify the following Boolean expressions, when `b1` and `b2` are Boolean variables:

- a** `!(b1 || b2)`
- b** `!(b1 == b2 && b2)`
- c** `!(!b2 || b1 == true)`
- d** `(!b1 == b2 || b2)`
- e** `(!(b1 == b2) || !(b2 == b1))`

- 10.** Write a selection statement that prints a message explaining whether the integer variable `numPoints` contains a value that is an even or an odd number. (Even numbers can be divided by two without a remainder).
- 11.** A repetition statement is often called a _____. The condition for repeating the _____ is specified as a _____ expression.
- 12.** Explain the difference between the `while` and `do-while` loops.
- 13.** An `assert` statement specifies a _____ expression, and (optionally) a _____. The _____ is printed to the terminal window if the expression evaluates to _____, and the execution is _____.
- 14.** Which of these statements are true?
- a** Assertions are always executed when the program is run.
 - b** If the Boolean expression in the `assert` statement evaluates to `true` at runtime, the execution aborts.
 - c** If the Boolean expression in the `assert` statement evaluates to `false` at runtime, the execution aborts.

3.6 Programming exercises

1. Write a program that reads the duration of a time interval in seconds, and then prints the corresponding number of hours, minutes and seconds to the terminal window. For example, if the user enters the number 3603, the program should output:

```
3603 seconds = 1 hour, 0 minutes and 3 seconds
```

Ensure that the program prints the time units grammatically, i.e. 0 hours, 1 hour, 2 hours, etc. Tip: Use a selection statement to determine whether the time units should be written in singular or plural form.



Use assertions to check that the values for each time unit are within expected limits.

- 2.** Write a program that determines whether a user-supplied value is inside or outside an interval. For example, the value 6 is inside the interval from 2 to 8, while the value 8 is outside the interval ranging from 1 to 5 (cf. figure below).

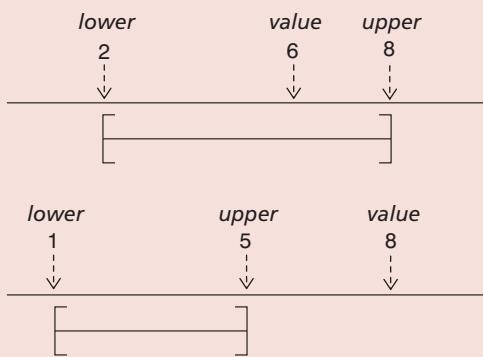
Write a program that:

1. Reads the lower and upper limits of an integer interval.
2. Reads a value.
3. Reports back if the value is inside or outside the given interval.

The program output should look this:

```
Enter the lower limit of interval [integer]: 2
Enter the upper limit of interval [integer]: 8
The value 6 lies within the interval [2, 8].
```

Print an appropriate report if the input value lies outside the interval.



- 3.** Write a program that determines in which quadrant of a Cartesian coordinate system a given point (x, y) lies. For example, if x has the value 2 and y has the value 3, the program should print:

The point $(x=2,y=3)$ is placed in the upper, right quadrant.

If x has the value -3 and y has the value -1, the program should print:

The point $(x=-3,y=-1)$ is placed in the lower, left quadrant.

Let the program read an x - and a y -coordinate, and report the quadrant in which the point (x, y) lies.

- 4.** Write a program that reads n integers from the keyboard and calculates their sum, using a do-while loop. Your program shall have the same functionality as Program 3.7.



5. Write a program that reads a sequence of positive integers from the keyboard and prints their average. Use a negative integer as a sentinel value. If the user enters the following sequence:

```
23  
42  
53  
3  
54  
82  
37  
43  
29  
44  
-1
```

the program should output as follows:

```
The sum of the integers is 410.  
The average of the 10 numbers is 41.
```

6. Assume that the user also wants to calculate the average value of negative numbers, and that the numbers may be floating-point values. Modify the program in Exercise 3.5 so that it accepts a user-supplied sequence of values, and prints the average of these values. The number of values in the sequence is read first, followed by the specified number of values:

```
7  
-3.25  
-4.44  
2.14  
3.66  
3.78  
3.67  
-4.33
```

The program should print a report similar to the one in Exercise 3.5:

```
The sum of the values is 1.01  
The average of the 7 numbers is 0.14
```

Print an explanatory text before asking the user for the number of values to read. You may assume that the user will always enter at least one value.

7. Write a program that assigns the grades A–F based on the number of points a student has scored in an exam. The intervals for the grades are: A for 91–100 points, B for 81–90 points, C for 61–80 points, D for 31–60 points and E for 15–30 points.

If a student gets less than 15 points, the grade F is assigned.

The program should read the student ID and number of points, where the ID is an even integer between 2 and 200, and permissible values for points are from 0.0 to 100.0.



Output from the program may look like this:

```
Enter the student id [integer]: 20
Enter the number of points [decimal number]: 87.5
Student with id 20 gets grade B
```

Output from the program when the student ID is valid, but the number of points is not, should be in the form:

```
Input student id [integer]: 24
Input number of points [decimal number]: 102.3
Exception in thread "main" java.lang.AssertionError:
  Invalid number of points for the exam: 102.3
  Allowed values range from 0.0 to 100.0
  at Grade.main(Grade.java:26)
```

- 8.** Write a program that calculates the volume of different solids:

1 Cube: $v = \text{length} \times \text{length} \times \text{length}$

2 Cylinder: $v = \pi \times r^2 \times \text{height}$

3 Sphere: $v = \frac{4}{3} \pi \times r^3$

where *length* is the length of each side of the cube, *height* is the height of the cylinder, *r* is the radius on the cylinder base or of the sphere, and *p* is a mathematical constant. Use the constant `Math.PI` as the value for *p*.

Use the `Scanner` class to create a user dialogue where the user can select the type of solid.

Sample output from the program:

```
Calculation of volume for solid objects
```

```
Enter 1 for cube, 2 for cylinder or 3 for sphere: 2
Enter the radius of cylinder [decimal value]: 2.5
Enter the height of cylinder [decimal value]: 1.5
Enter the unit for radius and height [text]: cm
```

```
The volume of a cylinder with radius 2.50 cm and height 1.50 cm
is 9.62 cubic cm.
```

- 9.** Extend the program in Exercise 3.7 to ask the user whether another volume should be calculated or whether the program should terminate.
- 10.** Write a program that calculates how much the amount deposited in a bank account has grown at the annual interest rate of $r\%$ after n years. The general formula for computing the amount, including interest, after n years is:

$$K_n = K_0 \times (1 + r / 100)^n$$

when K_0 is the principle amount.



Sample user dialogue and output from the program:

Calculation of Compound Interest

```
Enter the initial amount deposited [decimal value]: 10000.00
Enter the interest rate [decimal value]: 4.5
Enter the number of years [integer]: 10
Enter the currency used [string]: GBP
```

Amount with interest after 10 years: 15529.69 GBP

Write assertions to verify both the input data and the computed amount.

- 11.** Modify the program in Exercise 3.10 to calculate how much an amount has grown in n years for an interval of interest rates from $r_{min}\%$ to $r_{max}\%$. Let the user provide the minimum and maximum interest rates, and calculate the amount obtained after n years for each 0.5% increment in the interest rate inside this interval.

Sample user dialogue and output from the program:

Calculation of deposits with interests for min - max rates

```
Enter the initial amount deposited [decimal value]: 10000.00
Enter the minimum interest rate [decimal value]: 4.5
Enter the maximum interest rate [decimal value]: 8.25
Enter the number of years [integer]: 5
Enter the currency used [string]: GBP
```

Amounts will be printed with 0.5% increment in interest rates.

Amount with interest after 5 years with rate 4.5%: 12461.82 GBP

Amount with interest after 5 years with rate 5.0%: 12762.82 GBP

...

Amount with interest after 5 years with rate 7.5%: 14356.29 GBP

Amount with interest after 5 years with rate 8.0%: 14693.28 GBP

Write assertion statements to verify your assumptions about input data and the computed amount.





P A R T T W O

Object-based Programming

Using objects

LEARNING OBJECTIVES

By the end of this chapter you will understand the following:

- The relationship between a class and its objects.
- Representing the properties and behaviour of an object with fields and instance methods respectively.
- Creating objects using the `new` operator.
- Manipulating objects by reference variables.
- Calling methods on objects and accessing fields in objects.
- Representing characters in the computer.
- Using methods from the `String` class to create and manipulate strings.
- Distinguishing between reference equality and value equality for objects.
- Using primitive values as objects.

INTRODUCTION

In this chapter we will create objects from pre-defined classes, and manage them using references. Strings are used frequently in programming, and Java offers extensive support for handling such objects. We also discuss how primitive values can be used as objects.

4.1 Introduction to the object model

In any field, we need to master the terminology to understand the subject matter. This is also true for programming. The goal in this section is to introduce some important

concepts without going into too much detail. The details will be filled in as we build our conceptual apparatus.

Abstractions, classes and objects

We use abstractions to handle the diversity that surrounds us in everyday life. An *abstraction* represents the relevant properties of an object required to solve the problem at hand, so that we can distinguish it from other types of objects. Both a Volvo S40 and a Toyota Avensis will be perceived as vehicles. Our ability to create an abstraction of what we perceive as a vehicle helps us with this task. The colour or the model is not relevant, the deciding factor is that it can be driven.

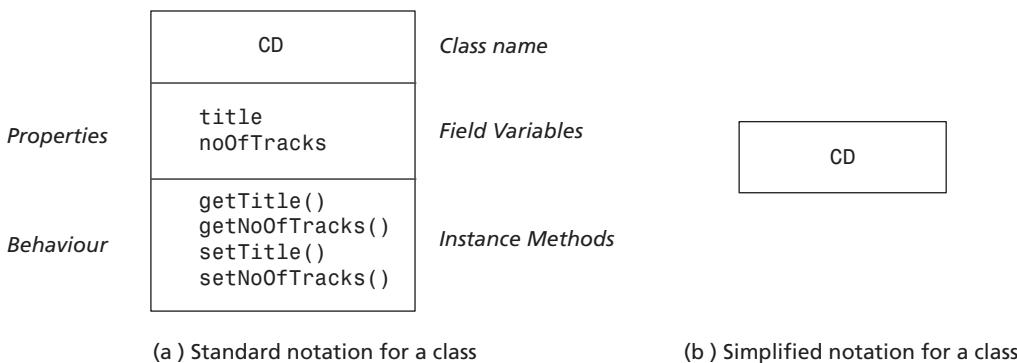
When we write programs that process information about abstractions from reality, it is helpful to represent the *properties* and *behaviour* of these abstractions in the computer. In Java, abstractions can be represented by classes. A *class* describes objects of a particular type – what information they can contain and what they can do, i.e. it specifies the properties and behaviour of these objects.

We will illustrate the relationship between a class and the objects it represents by specifying a simple class for music CDs. Such a CD has certain properties: a title and a number of music tracks. (Let's presume that the name of the artist or the title of the tracks is not important in the present context.) For a CD it should be possible to determine its title and how many tracks there are on it. An abstraction that we use in the computer need not be an exact model of a CD in real life. In our abstraction of a CD, we will change the title and number of tracks on the CD, but this is not usually the case with a real CD. To keep the example simple, we are also not concerned about other behaviours of a CD, for example being able to read the names of all the tracks on the CD.

4



FIGURE 4.1 Class notation in UML



A *class declaration* contains a number of *declarations* that define the properties and behaviour of its objects. Figure 4.1 shows the class `CD` in UML notation (Appendix H). The corresponding class in Java is shown in Figure 4.2. A class declaration begins with the keyword `class` followed by the name of the class. Declarations in a class are always enclosed in curly brackets, `{}`.

FIGURE 4.2 Class declaration in Java

```
Class name  
↓  
class CD {  
  
// Declaration of field variables          Properties  
String title;  
int noOfTracks;  
  
// Declaration of instance methods          Behaviour  
String getTitle() { return title; }  
int getNoOfTracks() { return noOfTracks; }  
void setTitle(String newTitle) { title = newTitle; }  
void setNoOfTracks(int nTracks) { noOfTracks = nTracks; }  
  
}
```

The properties of the objects of a class are specified with the help of *field variables* (also just called *fields*). The declaration of the class **CD** has two field variables, **title** and **noOfTracks**, which store information about the title and number of tracks on a CD respectively. The behaviour of the objects is specified by *instance methods*. The class **CD** has four such methods, with the names **getTitle**, **getNoOfTracks**, **setTitle** and **setNoOfTracks**. These methods are used to look up and change information about the title and the number of tracks on a CD. A method contains operations that exhibit the desired behaviour in an object when the method is executed.

4



Objects, reference values and reference variables

A class is a “blueprint” for creating objects that have properties and behaviour defined by the class. In the literature, the term *instance* is often used as a synonym for an object. There is only one **CD** class, but we can create several **CD** objects. When we create an object from a class, we get a *reference value* for the newly-created object. Each object of a class is unique (i.e. has a unique identity indicated by the reference value), even though all the objects of the class represent the same properties and behaviour.

A *reference variable* (often shortened to just *reference*) is a variable that can store a *reference value* of an object. References are analogous to variables of primitive data types that store values of primitive data types, and store reference values of objects. A reference thus *refers to* the object identified by the reference value stored in the reference. We manipulate an object via a reference that holds the reference value of the object.

A *reference variable declaration* is used to declare a reference variable. It specifies the *name* of the reference and its *reference type*. A class is a reference type. References can only refer to objects that have this type, i.e. that are objects of the specified class. The following declaration will result in memory being allocated for the reference **favouriteAlbum** that can store the reference value of a **CD** object:

```
CD favouriteAlbum;
```

The new operator

No object is created as a result of declaring a reference. The following expression statement is responsible for creating an object of the class CD:

```
new CD();
```

This language construct consists of two parts: the operator `new` and a *constructor call*, `CD()`. The operator `new` creates an object of class `CD`, whose name is specified in the constructor call, and returns the reference value of the new object. The constructor call also specifies *a list of parameter values* (empty in the constructor call above) that can be used to initialize the field variables in the new object that was created from the declared class.

Often we combine the declaration of a reference and the creation of an object whose reference value is assigned to the reference, as shown in Figure 4.3.

FIGURE 4.3 Object creation

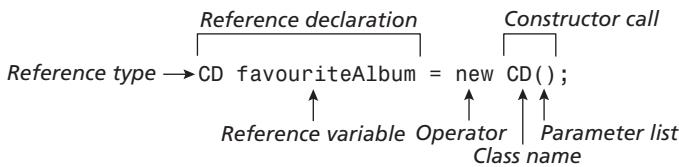


Figure 4.4a shows the result of executing the declaration statement above in UML notation, with explicit specification of a reference to the newly-created object. We see that the fields `title` and `noOfTracks` have the values `null` and `0` respectively. This will always be the case for each `CD` object we create using the `new` operator, because of the way we have defined the `CD` class.

The notation in Figure 4.4a might seem cumbersome. In that case, we can use alternative notations shown in Figure 4.4b and Figure 4.4c. In UML, objects are typically designated with both the reference name and the class name, in the following format: *referenceName: className*.

Using objects

After an object has been created, a reference that refers to the object can be used to send messages to the object. Messages take the form of a *method call* in Java (see Figure 4.5a).

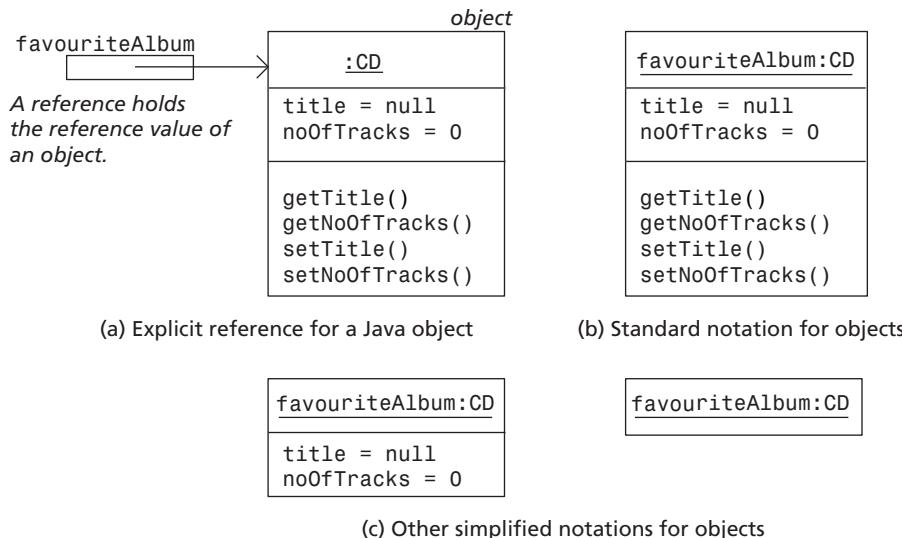
A method call to an object specifies:

- the reference to the receiving object
- the name of the method that is to be executed
- any other information (in a parameter list) that the method needs to execute its statements (see Method calls and actual parameter expressions in Chapter 7).

Note the dot (`.`) between the reference and the method name. The class of the referred object must define the method that is called. In Figure 4.5a, the method call will result in the method `setTitle()` in the class `CD` being executed. From Figure 4.2 we see that this method assigns the value it receives in its parameter list to the field `title`, i.e. the field

`title` in the object referred by the reference `favouriteAlbum` will be assigned the reference value of the `String` object "Java Jam Hits".

FIGURE 4.4 Object notation in UML

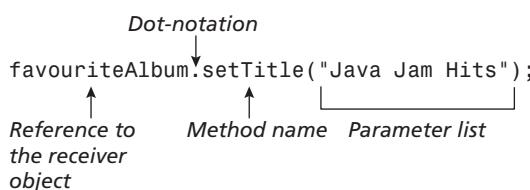


4

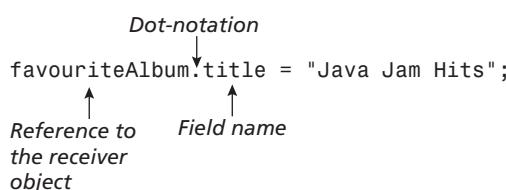


We can use the dot-notation, together with a reference, to access the field variables in an object. An example is shown in Figure 4.5b. The code line in Figure 4.5b will give the same result as the method call in Figure 4.5a. If a computation is required to set the value of a field variable, it is generally a good idea to call an instance method in the object, rather than repeating the code for the computation whenever the field value is set. This helps to ensure that the field is updated properly and consistently.

FIGURE 4.5 Dot notation



(a) Calling a Method in an Object



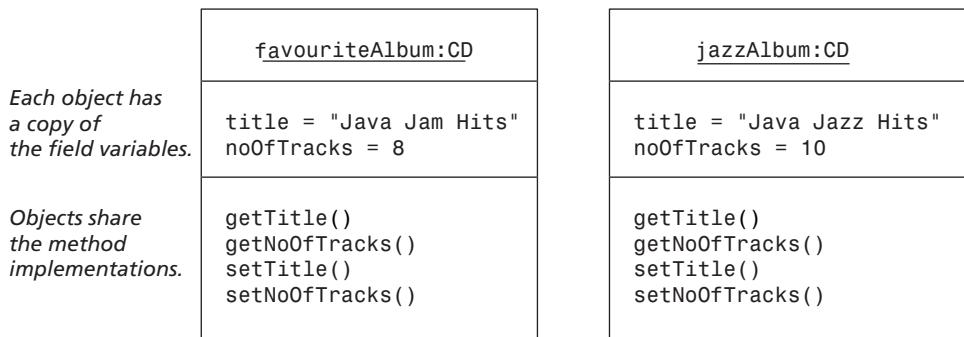
(b) Referring to a Field in an Object

Object state

Each object has its own copy of field variables. The fields of different objects of class `CD` can therefore have different values, as shown in Figure 4.6. The values of the fields in an object at any given time constitute the *state* of the object. The behaviour of an object is given by the instance methods. Coding of a method, i.e. the code that constitutes a method declaration, is called a *method implementation*, and objects of the same class share method implementations (see Method execution and the `return` statement in Chapter 7).

Program 4.1 declares a class `CDSampler` that uses objects of the class `CD`. It is a useful exercise to identify the concepts we have introduced so far in this chapter by examining this code. Information that the program prints corresponds to the state of the `CD` objects illustrated in Figure 4.6.

FIGURE 4.6 Object state



4



PROGRAM 4.1 Objects

```
// Using CD-objects
public class CDSampler {
    public static void main(String[] args) {
        // Create 2 CDs.
        CD favouriteAlbum = new CD();
        CD jazzAlbum = new CD();

        // Set state of the CDs.
        favouriteAlbum.setTitle("Java Jam Hits");
        favouriteAlbum.setNoOfTracks(8);
        jazzAlbum.setTitle("Java Jazz Hits");
        jazzAlbum.setNoOfTracks(10);

        // Print state of the CDs.
        System.out.println("Title of favourite album: " +
                           favouriteAlbum.getTitle());
        System.out.println("Number of tracks on favourite album: " +
                           favouriteAlbum.getNoOfTracks());
```

```
        System.out.println("Title of jazz album: " + jazzAlbum.getTitle());
        System.out.println("Number of tracks on jazz album: " +
                           jazzAlbum.getNoOfTracks());
    }
}
```

Program output:

```
Title of favourite album: Java Jam Hits
Number of tracks on favourite album: 8
Title of jazz album: Java Jazz Hits
Number of tracks on jazz album: 10
```

BEST PRACTICES

A class declaration should only declare properties and behaviour that are relevant for the problem at hand.

4.2 Strings

4



Text is a common medium for representing information. Text consists of *characters*. In programming languages, text is usually *a sequence of characters*, and is called a *text string*, or just *string*. Java provides a primitive data type, `char`, and a pre-defined class, `String`, that can be used to handle characters and strings respectively.

Characters and strings

In the computer each character is represented by an integer value called the *code number*. This is true for all characters, including letters, digits and other special characters. Java uses a standard called *Unicode* to represent characters. This standard assigns a unique code number for each character.

The primitive data type `char` represents the code number of each character as a 16-bit integer value, so that it is possible to represent 65 536 (2^{16}) different characters in the data type. This is large enough to represent the characters found in most of the languages in the world. The Unicode values (i.e. the code numbers in the Unicode standard) are usually specified as hexadecimal numbers. For example, the letter 'a' has the Unicode value `\u0061`, the digit '0' has the Unicode value `\u0030`, and the character '€' has the Unicode value `\u20ac`. The prefix `\u` indicates that the value is a code number for a character in the Unicode standard.

Character literals

To write a character as a `char` value in a Java program, we enclose the character or its Unicode value in single quotes (''). For example, the letter a can be written as '`'a'`' or



'\u0061'. This notation defines a character literal that is the code number of a particular character. Without the single quotes, the character a alone will be interpreted as a one-letter name in the program. Some examples of character literals are given in Table 4.1.

As we can see from Table 4.1, we must precede a single quote ('') by a back slash (\), and both the characters (\') must be enclosed in single quotes (''') to specify the character literal for a single quote. The backslash character (\) is used to "escape" the special meaning of a character. Some special characters, like those that do not have a visible representation but have a special meaning (for example, newline with the Unicode value \u000a), have predefined literals. For example, the newline literal is '\n'.

TABLE 4.1 Characters, Unicode representation and character literals

Character	Decimal value	Unicode value	Character literal
0 (zero)	48	\u0030	'0'
a	97	\u0061	'a'
A	65	\u0041	'A'
?	63	\u003f	'?'
single quote: '	39	\u0027	'\''
double quote: "	34	\u0022	'\"'
backslash: \	92	\u005c	'\\'
newline	10	\u000a	'\n'
tab	9	\u0009	'\t'
space	32	\u0020	' '

Character variables and arithmetic expressions

A character literal has the data type char. We can declare variables that can store characters, meaning that the code numbers these variables store are interpreted as characters:

```
char newline = '\n', tab = '\u0009';
char char1, char2, char3, char4;
char1 = char4 = 'a'; char2 = char3 = 'b';
```

Since a character is represented by an integer value, a character can be an integer operand in an arithmetic expression:

```
int sumCodeNumbers = char1 + char2 + char3 + char4; // 97+98+98+97 => 390
int number = '5' - '0'; // 53 - 48 => 5
```

The code numbers of lowercase letters (a to z), uppercase letters (A to Z) and digits (0 to 9) are numbered consecutively in the Unicode standard, with lowercase letters from

\u0061 (a) to \u007a (z), uppercase letters from \u0041 (A) to \u005a (Z), and the digits from \u0030 (0) to \u0039 (9) (Appendix D). We can compare characters, and it is the code numbers that are actually compared:

```
boolean test1 = char1 == char4; // true
boolean test2 = char1 > char2; // false,
                           // since 'a' (\u0061) < 'b' (\u0062)
```

String literals

Analogous to character literals, we can define string literals by enclosing a sequence of characters in double quotes (""). In contrast to character literals that are integer values, string literals are objects of the class `String`. For example, the string literal "abba" is a `String` object. This object stores the characters 'a', 'b', 'b' and 'a' as a sequence. Other examples of string literals are given in Table 4.2. The table also shows the result of printing string literals, for example by calling the `System.out.println()` method. Note that any double quotes ("") that actually occur in a string must be escaped with a backslash (\), and that string literals cannot span more than one line in the source code.

TABLE 4.2 String literals

String literal	Printout
"Welcome to Forevereverland"	Welcome to Forevereverland
""	The empty string has no visible representation.
"!"	!
"\"Move it!\"", said the teacher."	"Move it!", said the teacher.
"A string cannot span more than one line."	Compile time error.
"Wrap a long string\n with a newline literal."	Wrap a long string with a newline literal.

4



String concatenation

As with other reference variables, we can declare variables of class `String` that can refer to string literals.

```
String firstName = "Leif", lastName = "Eriksen";
```

The character sequence in a `String` object cannot be modified, i.e. a `String` object is *immutable*. Calling methods in the `String` class that seemingly modify the string in a `String` object actually result in a new `String` object with the modified string. We will see several examples of string immutability later in this section.



A useful operation on strings is to join two strings, referred to as *string concatenation*. A new string is created, whose contents are the characters from the first string followed by the characters from the second string. The binary operator +, which also performs arithmetic addition, is used for concatenating two strings:

```
String fullName = firstName + " " + lastName; // "Leif Eriksen"
```

If one of the operands of the + operator is a string and the other is not, the other operand is automatically converted to its string representation before the concatenation is performed. The concatenation operator is left-associative, i.e. the concatenation is performed from left to right resulting in a new String object with the final string. The operator returns the reference value of the new String object. After execution of the statement above, the reference `fullName` will refer to a String object containing the string "Leif Eriksen".

Program 4.2 illustrates string concatenation. After execution of the statement at (1), the reference `course` will refer to the string "Introductory course in programming". Note the result of the concatenation in the statements at (2) and (3). In (2), the string literals "C" and "S" are concatenated with the int variable `courseNumber`, resulting in the string "CS100", as the value 100 in the variable `courseNumber` is converted to the string "100" before concatenation. In (3), the character literals 'C' (int value 67) and 'S' (int value 83) are added before addition to the int variable `courseNumber` (int value 100), resulting in the int value 250. This value is converted to a string and concatenated with the String variable `course`.

PROGRAM 4.2 String concatenation

```
// Illustrating string concatenation
public class StringConcatenation {
    public static void main(String[] args) {
        String course = "programming";
        course = "Introductory course in " + course; // (1)
        System.out.println("course: " + course);
        int courseNumber = 100;
        String course1 = "C" + "S" + courseNumber + ": " + course; // (2)
        String course2 = 'C' + 'S' + courseNumber + ": " + course; // (3)
        System.out.println("course1: " + course1);
        System.out.println("course2: " + course2);
        System.out.println((int)'C');
        System.out.println((int)'S');
    }
}
```

Program output:

```
course: Introductory course in programming
course1: CS100: Introductory course in programming
course2: 250: Introductory course in programming
```



Creating string objects

Earlier in this section we saw how a variable can be assigned a string literal. Specification of a string literal in the program implies creation of a `String` object that contains the string, and the reference value of this object can be assigned to a `String` reference variable:

```
String star = "madonna";
```

If several reference variables are subsequently assigned the same string literal, they are aliases: they will refer to the same `String` object as the string literal (see Figure 4.7):

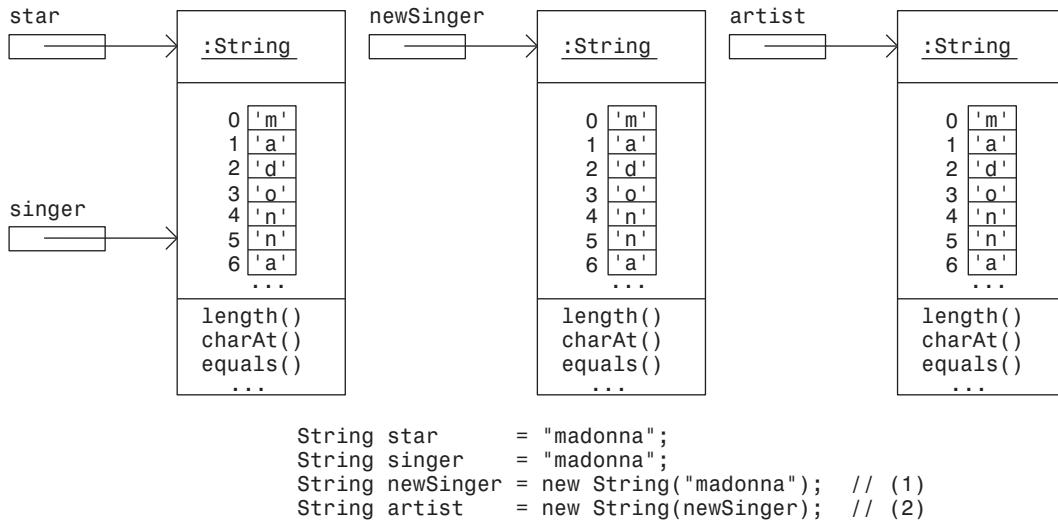
```
String singer = "madonna"; // The reference singer refers to the same
                           // String-object as the reference star.
```

Another way of creating `String` objects is by using the `new` operator together with a `String` constructor call:

```
String newSinger = new String("madonna"); // (1)
String artist    = new String(newSinger); // (2)
```

Use of the `new` operator implies creation of a new `String` object. In (1) a new `String` object is created based on the string literal "madonna", and in (2) a new `String` object is created based on the `String` object referred to by the reference `newSinger`. The reference variables `star`, `newSinger` and `artist` refer to three different `String` objects that have the same state (see Figure 4.7).

FIGURE 4.7 String creation



String comparison

Comparison of strings is based on *lexicographical order*, i.e. characters in corresponding positions in the two strings are compared based on their Unicode values. Entries in a telephone directory or words in a dictionary are listed according to lexicographical order. Based on lexicographical order, the string "abba" is *less* than the string "aha", since the



Unicode value of second character, 'b', in string "abba" is less than the Unicode value of the second character, 'h', in the string "aha".

The method `compareTo()` in the `String` class can be used to compare strings. We call this method on one string and send the second string as a parameter in the method call. The return value from the call to the method indicates the result of the comparison, as explained in Table 4.3 on page 87. In the program, the return value can be used to determine further action:

```
int result1 = star.compareTo(singer);      // == 0
int result2 = star.compareTo(newSinger);    // == 0

String group1 = "abba", group2 = "aha";
int result3 = group2.compareTo(group1);     // > 0
int result4 = group1.compareTo(group2);     // < 0
if (result4 < 0) { // true in this case.
    System.out.println(group1 + " is smaller!"); // Prints: abba is smaller!
}
```

In Table 4.3 we see that the methods `compareTo()` and `equals()` require a parameter of class `Object`. Such a parameter makes it possible to send any object as parameter to the method. The compiler will accept this, but during execution a runtime error can occur if the object passed is not of the right data type for the comparison.

Equivalence between two strings means that both `String` objects have the same state, i.e. the strings in the two objects have identical character sequences. This is called *value equality*. If we only want to compare two strings for value equality, we can use the `equals()` method rather than the `compareTo()` method:

```
boolean flagA = star.equals(singer);          // true
boolean flagB = star.equals(newSinger);        // true
```

The equality operator `==` cannot be used for comparing objects for value equality. This operator compares two *references* to determine whether they are aliases. This is called *reference equality*:

```
boolean flag1 = (star == singer)           // true
boolean flag2 = (star == newSinger)         // false
```

Converting primitive values to strings

It is often necessary to convert primitive values to their string representations. The static method `valueOf()` in the `String` class will convert a primitive value to a `String` object that contains the string representation of its value.

```
String numberStr = String.valueOf(3.14);      // "3.14"
String boolStr  = String.valueOf(true);         // "true"
```

If a string is a string representation of a primitive value and we wish to convert it to its corresponding primitive value, it is convenient to use a suitable `parseType()` method from the wrapper classes (see *Primitive values as objects* on page 92).



If a string contains the string representation of several primitive values, we can use the `java.util.Scanner` class to read the string and convert the characters to primitive values:

TABLE 4.3 Selected methods from the `String` class

<code>java.lang.String</code>	
<code>int compareTo(Object s2)</code>	C.compares two strings. For example, given the code line: <code>int result = s1.compareTo(s2);</code> where <code>s1</code> and <code>s2</code> are strings, we can conclude the following, depending on the value of the <code>result</code> variable: If <code>result < 0</code> , string <code>s1</code> is less than string <code>s2</code> . If <code>result == 0</code> , string <code>s1</code> is equal to string <code>s2</code> . If <code>result > 0</code> , string <code>s1</code> is greater than string <code>s2</code> .
<code>boolean equals(Object s2)</code>	C.compares two strings for equality, i.e. whether the respective strings have identical sequences of characters, and returns <code>true</code> if that is the case. Otherwise the method returns <code>false</code> .
<code>int length()</code>	Returns the number of characters in the string, i.e. the <i>length</i> of the string.
<code>static String valueOf(T t)</code>	Depending on the type <code>T</code> , returns a string representation of the value in <code>t</code> . For example, type <code>T</code> can be <code>boolean</code> , <code>char</code> , <code>double</code> , <code>float</code> , <code>int</code> or <code>long</code> .
<code>char charAt(int index)</code>	Returns the character at the <code>index</code> in the string. The first character is at <code>index 0</code> . Invalid index values will result in an <code>IndexOutOfBoundsException</code> .
<code>int indexOf(int charValue)</code> <code>int indexOf(String subString)</code> <code>int indexOf(int charValue, int startIndex)</code> <code>int indexOf(String subString, int startIndex)</code>	Returns the index of the <code>charValue</code> or index of the start of the <code>subString</code> in the string, otherwise returns -1. Argument <code>startIndex</code> can be used to start the search from a particular index, otherwise the search starts at index 0.
<code>String substring(int startIndex, int endIndex)</code>	Returns a new string consisting of the sequence of characters from <code>startIndex</code> to <code>(endIndex-1)</code> . The returned string has length <code>(endIndex-startIndex)</code> . Invalid index values will result in an <code>IndexOutOfBoundsException</code> .
<code>String toLowerCase()</code> <code>String toUpperCase()</code>	Returns a new string in which all characters that are letters in the original string are converted to either lowercase or uppercase, respectively.
<code>String trim()</code>	Returns a new string where invisible characters at the start and end of the original string are deleted. These invisible characters can be, for example, space, tab or newline.

```

Scanner input = new Scanner("101 25.0 2006");           // (1)
int code      = input.nextInt();                      // 101
double salesTax = input.nextDouble();                // 25.0
long year     = input.nextLong();                     // 2006L

```

Note that the string representation of a floating number is according to the rules of the *locale* for a country. For example, in Norway the decimal character is a comma (,) and not a period (.).

Other useful methods for strings

Table 4.3 on page 87 shows a selection of methods from the `String` class. The class has many useful methods for creating new `String` objects, for comparing strings, searching for characters in strings and converting `String` objects.

If the character index passed to any of the methods in the `String` class is not a valid index in the interval $[0, n-1]$, where n is the length of the string, an `IndexOutOfBoundsException` is thrown. Exceptions signal a runtime error, and are discussed in Chapter 11. Program 4.3 illustrates some of the methods. The statement in (1) is commented out, as a `String` object can obviously not be compared with an `Integer` object.

4

PROGRAM 4.3 Use of miscellaneous `String` methods

```

// Illustrating misc. String methods
public class MiscStringMethods {
    public static void main(String[] args) {
        String group1 = "abba", group2 = "aha";
        int result3 = group2.compareTo(group1);           // > 0
//      int result4 = group2.compareTo(new Integer(10)); // (1) Error!
        if (result3 > 0) // True in this case.
            // "aha" is greater lexicographically.
            System.out.println(group2 + " is greater lexicographically!");
        if (group1.length() > group2.length())           // 4 > 3
            // "abba" is greater in length.
            System.out.println(group1 + " is greater in length!");

        String star = "madonna";
        int strLength = star.length();                  // 7
        System.out.println(star.charAt(strLength-4));    // 0 (index: 3,
                                                       //       i.e. 4th. char)
        System.out.println(star.indexOf('n'));           // 4
        System.out.println(star.substring(0,3));          // "mad"
    }
}

```

Program output:

```
aha is greater lexicographically!  
abba is greater in length!  
0  
4  
mad
```

BEST PRACTICES

Remember that `String` objects are immutable. Operations that seemingly modify a `String` object actually return a new `String` object. Consider using the `StringBuilder` class if string content is modified frequently.

4.3 Manipulating references

4



Reference types and variables

The primitive data types define which values are legal and which operations can be performed on these values. Analogous to a primitive data type, a class defines a data type called a *reference type*. Later in the book we will meet other kinds of reference types.

Assignment to reference variables

Just as an `int` variable can only store values of the `int` data type, a reference variable of a specific reference type can only store reference values of objects of that reference type. Analogous to changing the value in an `int` variable, we can change the reference value stored in a reference variable. In this way, a reference variable can refer to different objects of the same reference type at different times.

Aliases: one object, several references

The same reference value can be assigned to several reference variables. Assignment never copies any object, only the reference value. When a reference value is assigned to several reference variables, these variables are called *aliases* for the object identified by the reference value stored in them. An object can be manipulated by any of its aliases. Figure 4.7 showed that the reference variables `star` and `singer` are aliases for the same `String` object.

The `null` literal

The literal `null` is a special reference value that can be assigned to any reference variable. The reference value `null` indicates that the reference variable does not refer to any object. After assignment of `null` to a reference, the object previously referred to by the reference will no longer be available via this reference. Program 4.4 shows that a runtime error (`NullPointerException`) can occur if we use a reference that has the value `null`.

Comparing objects

In the subsection *String comparison* on page 85 we distinguished between value equality and reference equality for `String` objects. We can now generalize the discussion about strings to other objects. What does it mean if we say that two cars are equal, or two beers are equal? To compare objects for value equality, the class must provide its own implementation of the `equals()` method. This method has a special position in Java, and is used for comparing two objects for value equality.

The `equals()` method must check that it is meaningful to compare the two objects for value equality. For example, there is no obvious reason to compare a car with a beer for value equality, or for that matter, compare a `String` object with a `CD` object. As we have seen, the class `String` implements its own `equals()` methods for comparing two strings for value equality.

The equality operator `==` can be used to determine whether two references are aliases, i.e. the operator only compares the reference values stored in the references. See *Aliases: one object, several references* on page 89 for examples of reference equality.

Program 4.4 shows an example of how we can swap reference values in reference variables. Figure 4.8 illustrates the process. To swap values, we need to use an extra reference variable. Note that assignment copies reference values, not objects. The figure also shows which aliases are created as the program executes.

The output from Program 4.4 shows that a reference variable (`groupName`) with the value `null` has the string representation "null". During execution the program terminates at (6) with the message that an `NullPointerException` has occurred. The cause of the error is that we are using the reference `groupName`, which has the value `null`, to call the method `length()` on a non-existent `String` object.

BEST PRACTICES

Make sure that a reference variable refers to an object before the reference is used. Any alias of the object can be used to manipulate the object.

PROGRAM 4.4 Swapping reference values

```
// Illustrating aliases
public class ReferenceValueSwapping {
    public static void main(String[] args) {
```

```

String group1 = "abba", group2 = "aha", groupName;      // (1)
groupName = group1;                                     // (2)
group1 = group2;                                       // (3)
group2 = groupName;                                     // (4)
groupName = null;                                      // (5)
System.out.println("group1 refers to: " + group1);
System.out.println("group2 refers to: " + group2);
System.out.println("groupName refers to: " + groupName);
System.out.println(groupName.length());                 // (6)
}
}

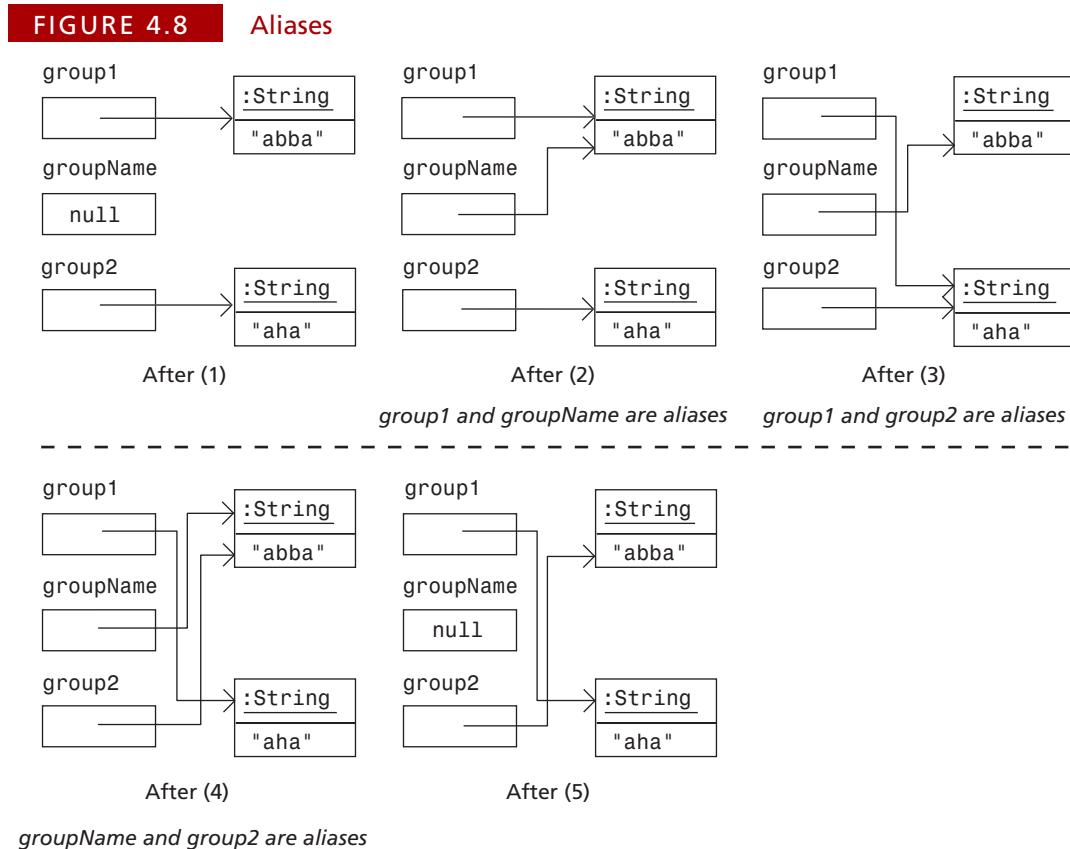
```

Program output:

```

group1 refers to: aha
group2 refers to: abba
groupName refers to: null
Exception in thread "main" java.lang.NullPointerException
at ReferenceSwapping.main(ReferenceSwapping.java:12)

```



4.4 Primitive values as objects

In Java, primitive values are not objects. Java offers *wrapper* classes so that values of primitive data types such as `int` and `double` can be treated as objects. Table 4.4 shows the wrapper classes that can be used to encapsulate primitive values. There is a wrapper class for each primitive data type, and their names make the correspondence between the primitive data types and the wrapper classes clear. Note that the wrapper class for `int` values is called `Integer`.

TABLE 4.4 Wrapper classes for primitive values

Primitive data type	Corresponding wrapper class
<code>boolean</code>	<code>Boolean</code>
<code>char</code>	<code>Character</code>
<code>byte</code>	<code>Byte</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>

4



Auto-boxing

Auto-boxing is the process of automatic conversion from a primitive value to a corresponding wrapper object. Converting from primitive values to objects of the corresponding wrapper class is simple: we specify the primitive value where we want to use the corresponding wrapper object:

```
Integer iRef = 10; // (1) Auto-boxing, 10 is encapsulated in an Integer object.
```

The right-hand side of the assignment operator can be any `int` expression. Its value will be evaluated and automatically encapsulated in an `Integer` object, whose reference value will be assigned to the reference variable on the left-hand side of the declaration.

Auto-unboxing

Auto-unboxing is the process of automatic conversion from a wrapper object to the corresponding primitive value. This process reverses the encapsulating process. We specify the reference to the wrapper object where we want to use the primitive value encapsulated in the wrapper object:

```
int j = iRef; // (2) Auto-unboxing, j gets the value 10.
```

The right-hand side of the assignment operator can be any expression that evaluates to a reference value of an `Integer` object. The `int` value encapsulated in the `Integer` object will be assigned to the variable on the left-hand side.

Auto-boxing and unboxing work similarly for the other primitive types.

Explicit boxing and unboxing

We can also do explicit conversion between primitive data values and wrapper objects. Wrapper classes have constructors that take a primitive value for encapsulation, and methods to read the value in the wrapper object.

```
Integer iRef = new Integer(10);    // (1) Explicit boxing
int j = iRef.intValue();          // (2) Explicit unboxing
```

The method `intValue()` in the class `Integer` returns the value in the wrapper object as an `int` value. Program 4.5 shows examples of boxing and unboxing.

PROGRAM 4.5 Conversion between a primitive value and an object

```
// Conversions: wrapper <--> primitive value
public class PrimitiveValueWrapper {
    public static void main(String[] args) {
        // A primitive value.
        int valueIn = 2006;

        // Two ways of creating an object from a primitive value:
        Integer valueObject;
        valueObject = new Integer(valueIn);
        valueObject = valueIn;           // Simple variant

        // Two ways of creating a primitive value from an object:
        int valueOut;
        valueOut = valueObject.intValue();
        valueOut = valueObject;         // Simple variant
        assert(valueIn == valueOut);    // Assert: same primitive value
        System.out.println("valueIn: " + valueIn + ", valueOut: " + valueOut);
    }
}
```

Program output:

```
valueIn: 2006, valueOut: 2006
```

4



Useful methods in the wrapper classes

A selection of methods from the `Integer` class is shown in Table 4.5. The other numerical wrapper classes offer similar methods. Note that wrapper objects are immutable, i.e. we cannot change the primitive value in an wrapper object, only read it.

TABLE 4.5 Selected methods from the `Integer` class

<code>java.lang.Integer</code>	
<code>int intValue()</code>	Returns the value in the wrapper object as an <code>int</code> .
<code>String toString()</code>	Conversion from wrapper object to string. Returns a string representation of the primitive value in the wrapper object.
<code>static String toString(int i)</code>	Conversion from wrapper object to string. Returns a string representation of the <code>int</code> value passed as argument.
<code>static int parseInt(String s)</code>	Conversion from string to primitive value. Interprets a string as an <code>int</code> value. This method accepts strings containing digits and the minus operator (-) only. It throws a <code>NumberFormatException</code> (see Chapter 11) if the string does not represent an <code>int</code> value.

Figure 4.9 shows conversions between an `int` value, a string and an `Integer` object. Program 4.6 tests the transitions in Figure 4.9. Case A shows conversion of a string to an `Integer` object that is further converted to a primitive value, and this value is finally converted back to a string. We have used methods from Table 4.5. During execution of the program, the assertion at (1) tests that the final string is equivalent to the original string. Case B shows the conversion of a string that represents a floating-point number to a primitive value first, followed by a conversion to a wrapper object, and then back to a string. During execution of the program, the assertion at (2) tests that the final string is equivalent to the original string.

PROGRAM 4.6 Conversions between strings, primitive values and wrappers

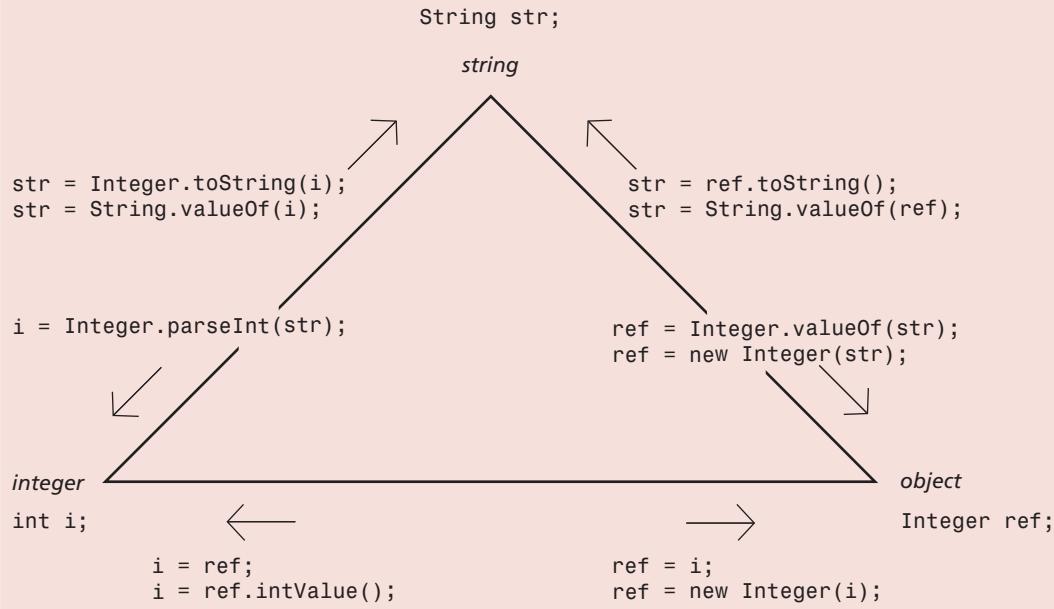
```
// Conversions: string --> wrapper --> primitive
public class PrimitiveValueRepresentation {
    public static void main(String[] args) {
        String string1, string2;

        // Case A: string --> wrapper --> primitive --> string
        string1 = "2005";
        Integer iWrapper = new Integer(string1);
        int iPrimitive = iWrapper;
        string2 = Integer.toString(iPrimitive);
        assert(string1.equals(string2)); // (1)

        // Case B: string --> primitive --> wrapper --> string
        string1 = "12.5";
        double dPrimitive = Double.parseDouble(string1);
        Double dWrapper = dPrimitive;
        string2 = dWrapper.toString();
        assert(string1.equals(string2)); // (2)
    }
}
```

}

FIGURE 4.9 Conversions between strings, primitive values and wrappers



4.5 Review questions

1. A class specifies the _____ and _____ of objects that can be created from the class.
2. Instance variables represent _____, and instance methods represent the _____ of objects that can be created from a class.
3. Which statements are true?
 - a An object is created from a class.
 - b A class creates an object.
 - c A reference stores the reference value of an object.
 - d A reference stores the reference value of a class.
 - e Two objects can be aliases.
 - f References that refer to the same object are aliases.
 - g References `ref1` and `ref2` are aliases if `(ref1 == ref2)` is true.
4. The following code line will create an object of class `CD` referred to by the reference `myCD`. True or false?
`CD myCD;`



5. Given the class CD, rewrite the following code lines so that there are no compile time errors.

```
CD cd1 = CD;  
CD cd2 = CD();  
CD cd3 = new CD;  
CD cd4 = new;  
new CD cd5;
```

6. How many objects of the class CD exist after the execution of the following statements?

```
CD cd1 = new CD();
```

7. Cd cd2 = cd1; Given a reference to an object, what notation can we use to call a method or access a field in an object?

8. Given the reference myCD, that refers to a CD object, which expressions are valid?

- a myCD.getTitle()
- b CD.getTitle()
- c myCD.getTitle
- d myCD.title
- e CD.title

9. What is the difference between 'z' and "z"?

10. Assume that we have two references, str1 and str2, that refer to String objects. If (str1 == str2) is true, then str1.equals(str2) is always true. But the converse is not always true. Explain why.

11. What will the following statements print?

```
System.out.println(2000 + 2);      // (1)  
System.out.println(2000 + '2');    // (2)  
  
System.out.println(2000 + "2");    // (3)
```

12. Given the following code, which statements will read the double value correctly?

```
Scanner source = new Scanner("3.14");  
  
a double pi = source.getDouble();  
b double pi = source.nextDouble();  
c double pi = source.readDouble();  
d double pi = source.parseDouble();
```

13. Given the following declarations, which statements will not compile?

```
int iVal;  
double dVal;  
Integer iRef;  
Double dRef;
```

```
a iRef = 12;  
b iVal = iRef;  
c dVal = iRef;  
d dVal = 12;  
e dVal = 12.5;  
f dRef = 12;  
g dRef = 6 * 2.0;
```

4.6 Programming exercises

1. Write a program that prints the following pattern:

```
#####
#
#
#
#
# #
```

2. Write a program that asks the user for relevant information (indicated by the uppercase words below) in order to print the following horoscope for the user:

Expect bad news, NAME.
Your manager, MANAGER, is not happy with WHATEVERYOUDO,
and is planning to move you to the DEPARTMENTNAME department.
Your salary will be reduced by PERCENT%.
But your manager also has a surprise coming: SURPRISE.

3. Write a program that reads a line of text from the keyboard, and prints an acronym formed from the first letter of each word.

Example:

```
Enter a line of text: Just another valuable artifact  
JAVA
```

4. Write a program that reads a line of text from the keyboard, and prints it back after replacing each occurrence of any four-letter word in the text with the word "x***", where x is the first character in the four-letter word.

Example:

```
Enter a line of text: I love Java very much  
I l*** J*** v*** m***
```

5. Write a program that reads a string from the keyboard in which substrings are separated by the character literal '|'. The program should print the substrings in a suitable format. For example, given the string "DOT-COM|IT Place 1|5020|Bergen", the program prints:

```
DOT-COM
```



IT Place 1

5020

Bergen

4



More on control structures

LEARNING OBJECTIVES

By the end of this chapter you will understand the following:

- Using extended assignment operators to compute the new value of a variable based on its previous value.
- Using increment (++) and decrement (--) operators to increment and decrement the value of a variable by 1, respectively.
- Writing counter-controlled loops using the `for(;;)` statement.
- Changing the control flow in a loop using `continue` and `break` statements.
- Avoiding common pitfalls when writing loops.
- Using the multiple selection `switch` statement.
- Changing the control flow in a `switch` statement using the `break` statement.

INTRODUCTION

Loops can be used to execute actions repeatedly. The `for(;;)` statement allows great flexibility in writing loops, and counter-controlled loops in particular. We discuss how the `continue` and the `break` statements change the control flow in a loop, but also caution on their use. These two statements are considered harmful if they are not used with care, as they foster what is colloquially known as *spaghetti code*, i.e. code with many control flow paths that is difficult to comprehend. We also discuss common pitfalls when writing loops, such as unintentional infinite loops and one-off errors.

The simple `if` statement and the `if-else` statement limit the number of selections. We discuss the `switch` statement, which allows multiple selections to be specified. We give examples of different scenarios of control flow through the `switch` statement. In particular, we discuss how the `break` statement affects the control flow in a `switch` statement.

While the break statement is best avoided in loops, it allows us to control the program execution in a switch statement.

5.1 The extended assignment operators

A very common operation is computing the new value for a variable based on its old value, as in the following assignment:

```
x = x + (y); // (1)
```

The variable `x` occurs as an operand for a binary operator (in this case `+`) on the right-hand side of the assignment operator (`=`), where `(y)` is any valid expression, and the result of the binary operation is assigned to the variable `x`. In such a case, we can use the *extended assignment operator*:

```
x += (y); // (2)
```

The assignment statement (2) is equivalent to assignment statement (1), where `+=` is an extended assignment operator. Note that there is no space between the `+` operator and the `=` operator.

Java provides a number of such extended assignment operators that combine arithmetic operations with assignment. Program 5.1 illustrates the use of such operators.

PROGRAM 5.1 Using extended assignment operators

5



```
// Illustrating use of extended assignment operators
public class ExtendedAssignmentOperators {

    public static void main(String[] args) {
        int i = 5, j = 10;
        double x = 5.0, y = 10.5, z = 10.0;
        i += j;           // i = i + (j)
        x -= y + 30.5;   // x = x - (y + 30.5)
        j *= i + 20;     // j = j * (i + 20)
        y /= 5.0;         // y = y / (5.0)
        z %= 3.0;         // z = z % (3.0)
        System.out.println("i : " + i);
        System.out.println("x : " + x);
        System.out.println("j : " + j);
        System.out.println("y : " + y);
        System.out.println("z : " + z);
    }
}
```

Program output:

```
i : 15
x : -36.0
j : 350
```

```
y : 2.1  
z : 1.0
```

5.2 The increment and decrement operators

Very often we need to increment the value in a variable by 1. Both the following assignments can be used for this purpose:

```
x = x + 1;           // (1)  
x += 1;             // (2)
```

To increment by 1, we can also use the *increment operator*:

```
x++;                // (3) The value in x is incremented by 1.
```

The statement in (3) achieves the same result as the statements in (1) and (2): the value in *x* is incremented by 1.

We can use the *decrement operator* to decrement the value in a variable by 1:

```
x--;                // (4) The value in x is decremented by 1.
```

The increment and decrement operators come in two flavours, as *pre* (`++x` and `--x`) and as *post* (`x++` and `x--`) *operators*, i.e. as prefix and postfix to the operand. It does not matter which form we use in the examples above but, as Program 5.2 shows, one should be careful when expressions such as `x++` and `x--` are used as operands in other expressions.

In Program 5.2 the value of the variable *i* is incremented by 1 in (1) and (2), but the value that is assigned to the variable *j* is different. The reason for this difference is that the pre-increment operator `++` in the expression `(++i)` adds one to the value of *i*, and returns the *new* value as the value of the expression `(++i)`. On the other hand, the post-increment operator `++` in the expression `(i++)` returns the *current* value of *i* as the value of the expression `(i++)`, but it also increments the value of *i* by 1.

The side-effect of both pre- and post-increment operators is to increment the value in its operand by 1. The pre- and post-decrement operators behave analogously, but decrement the value of the operand by 1. If we are only interested in the side-effect of these operators, as in (3) and (4) above, then it is irrelevant whether we use the pre- or the post-form.

PROGRAM 5.2 Using increment and decrement operators

```
// Illustrating use of increment and decrement operators  
public class IncrAndDecrOperators {  
  
    public static void main(String[] args) {  
        int i = 10, j;  
        j = i++; // (1) Postfix: j gets the value 10, and i gets the value 11  
        System.out.println("j : " + j + " and i: " + i);  
    }  
}
```



```
i = 10;  
j = ++i; // (2) Prefix: j gets the value 11, and i gets the value 11  
System.out.println("j : " + j + " and i: " + i);  
}  
}
```

Program output:

```
j : 10 and i: 11  
j : 11 and i: 11
```

BEST PRACTICES

As the increment and the decrement operators have the side-effect of updating their operand, they should not be used in combination with other operators.

5.3 Counter-controlled loops

The `while` statement is suitable for implementing *conditional loops* in which the number of times the loop should iterate is not known. However, there are situations in which the number of iterations is known beforehand. Such loops are called *counter-controlled loops*, as we use a counter to repeat the loop an exact number of times. In Java, the `for(;;)` statement is specifically designed to implement counter-controlled loops.

5



Figure 5.1a illustrates the syntax of the `for(;;)` loop. The loop consists of a *loop header* and a *loop body*. Inside the loop header we specify the three parts: *initialization*, *loop condition* and *updating*. In Figure 5.1a, the loop body comprises a *statement block*, but it can also be a single statement without the block notation, {}.

The initialization part declares a loop variable (`i`) and assigns a *start value* (1) to it. The loop variable `i` acts as the counter for the number of times the loop is executed. As can be expected, the initialization is only executed once, to declare and initialize the loop variable. The value of the loop variable `i` is compared to a *final value* (5) in the loop condition (`i <= 5`). If the condition is true, the loop body is executed, otherwise the loop terminates and execution continues after the `for(;;)` loop statement. Immediately after the execution of the loop body, the loop variable is incremented in the updating part (`i++`) to keep track of the number of iterations executed so far. After updating the loop variable, the loop condition is always tested.

Updating the loop variable means that at some point the loop condition will become `false`. As long as the loop condition is `true`, the loop body is executed one more time. The execution of the `for(;;)` loop is illustrated in Figure 5.2.

FIGURE 5.1

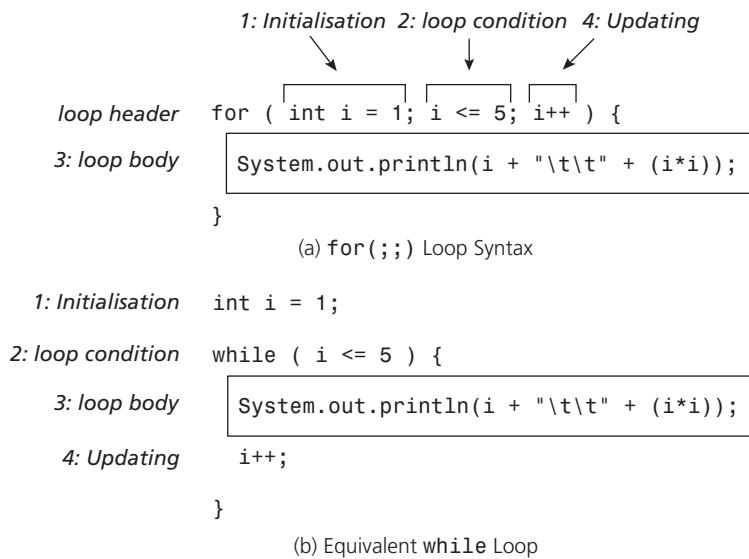
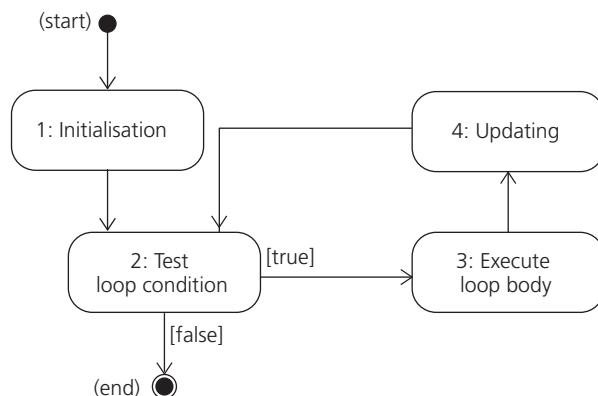
The `for(;;)` loop syntax

FIGURE 5.2

Executing the `for(;;)` loop

5



The `for(;;)` loop in Figure 5.1a is executed five times, printing the squares of the numbers from 1 to 5 in the terminal window:

1	1
2	4
3	9
4	16
5	25

After five iterations, the loop variable `i` has the value 6, and the loop condition evaluates to `false`. The loop thus terminates after the desired number of iterations.

The `for(;;)` loop in Figure 5.1a is shown in Figure 5.1b coded as a `while` loop. Note that in Figure 5.1b the initialization is executed only once before the `while` statement. The updating of the loop variable is always done *after* the execution of the loop body. The loop

body is skipped entirely if the loop condition is `false` at the start of the loop. That is the reason why the `for(;;)` loop is a *pretest loop*.

If the loop condition is not formulated properly, we can risk it never becoming `false` and the loop never terminating (called *infinite loop*). Care must be taken not to change the value of the loop variable in the loop body, as this might result in the loop not iterating the correct number of times.

Program 5.3 uses a `for(;;)` loop to spell out a phone number in words. The telephone number is read as a string from the keyboard at (1). Each character in the string (`inputPhoneString`) represents a digit in the phone number.

A `for(;;)` statement at (2) is used to iterate over the input string `inputPhoneString`. The loop body is executed for values from 0 to $n-1$, where n is the length of the input string. The loop condition (`i < inputPhoneString.length()`) ensures that the loop terminates when the value in the loop variable `i` is equal to the value returned by the method call `inputPhoneString.length()`, i.e. n .

For each iteration of the loop, an `if` statement translates one character to the corresponding word. The words are concatenated into a string (`outputPhoneString`) as the input string is processed, and the resulting string is printed out after the loop terminates. Note that the variable `outputPhoneString` will refer to a new `String` object after each concatenation operation.

PROGRAM 5.3 Use of the `for(;;)` statement

5



```
import java.util.Scanner;
// Program spells out a telephone number: if statement version
public class Telephone {
    public static void main(String[] args) {
        System.out.print("Enter the telephone number (as 55584152): ");
        Scanner keyboard = new Scanner(System.in);
        String inputPhoneString = keyboard.next(); // (1)
        String outputPhoneString = "";
        for (int i = 0; i < inputPhoneString.length(); i++) { // (2)
            char aChar = inputPhoneString.charAt(i);
            if (aChar == '1') {
                outputPhoneString += "one ";
            } else if (aChar == '2') {
                outputPhoneString += "two ";
            } else if (aChar == '3') {
                outputPhoneString += "three ";
            } else if (aChar == '4') {
                outputPhoneString += "four ";
            } else if (aChar == '5') {
                outputPhoneString += "five ";
            } else if (aChar == '6') {
                outputPhoneString += "six ";
            } else if (aChar == '7') {
                outputPhoneString += "seven ";
            }
        }
        System.out.println(outputPhoneString);
    }
}
```

```

        } else if (aChar == '8') {
            outputPhoneString += "eight ";
        } else if (aChar == '9') {
            outputPhoneString += "nine ";
        } else if (aChar == '0') {
            outputPhoneString += "zero ";
        } else {
            System.out.println(aChar + " is not a digit!");
        }
    }
    System.out.println(outputPhoneString);
}
}

```

Program output:

```

Enter the telephone number (as 55584152): 55584152
five five five eight four one five two

```

Local variables in the `for(;;)` loop

We have described a simple form of the `for(;;)` statement for counter-controlled loops, in which a loop variable is first declared and initialized, and subsequently updated, so that the loop condition eventually becomes `false`, resulting in the loop being terminated. The initialization in Figure 5.1a contains a *declaration* for the variable `i`. A variable that is declared in the initialization part of the loop header or in the loop body can only be used in the `for(;;)` loop, as they are local variables and therefore not accessible outside the loop. The code below shows that the local variables `j` and `m` are not accessible outside the `for(;;)` loop:

```

for (int j = 1; j <= 5; j++) {
    int m = 20;
    System.out.println(j + "\t" + (j*m));
}
System.out.println("j has the value: " + j); // Compile time error!
System.out.println("m has the value: " + m); // Compile time error!

```

If we need to access the values of the variables `j` and `m` outside the loop, we can declare them before the loop, as shown at (1) in the code below. Note that the initialization part of the loop header at (2) is now an assignment to initialize the variable `j` to the start value. The type declaration is omitted, as the loop variable is already declared outside the loop. Note also that the value of the loop variable `j` is 6 (and not 5) after the termination of the loop.

```

int j, m;                                // (1)
for (j = 1; j <= 5; j++) {                  // (2)
    m = 20;
    System.out.println(j + "\t" + (j*m));
}
System.out.println("j has the value: " + j); // Prints "6"
System.out.println("m has the value: " + m); // Prints "20"

```



Other increments in `for(;;)` loops

We are not limited to increments of 1 in the updating part of the `for(;;)` loop. We can specify other increments using extended assignment operators. The code at (2) in Program 5.4 will only print the squares of odd numbers from 1 to 5. The loop variable `i` is initialized to the start value 1, and incremented by 2 after each iteration of the loop. The loop terminates when the value of the loop variable `i` is 7. Care must be taken not only to initialize and increment the loop variable correctly, but also to ensure that the loop condition is formulated correctly.

Counting backwards with `for(;;)` loops

We can also write `for(;;)` loops that count backwards. The code at (3) in Program 5.4 illustrates how to write such a loop. The start value (10) is greater than the final value (0), so that the loop condition (`i > 0`) becomes false when the loop variable `i` gets the value 0. We use the decrement operator `--` to decrease the value of the loop variable `i` by 1 after each iteration of the loop.

Nested `for(;;)` loops

The loop body can also be a `for(;;)` loop, as it is a statement. We then have *nested for(;;)* loops. The code at (4) in Program 5.4 uses two nested `for(;;)` loops to print the multiplication tables from 1 to 5. The outer loop runs from 1 to 5, and the inner loop runs from 1 to 10. This means that for each iteration of the outer loop, the inner loop is executed ten times, resulting in the inner loop being executed altogether fifty times (5×10). This also means that the value of the loop variable in the inner loop changes more frequently than the value of the loop variable in the outer loop.

5

PROGRAM 5.4 Several examples using the `for(;;)` statement



```
// Illustrating various uses of the for(;;) loop
public class ForExamples {
    public static void main(String[] args) {

        // (1) Simple for loop
        // Print the square of numbers from 1 to 5.
        System.out.println("Square of numbers from 1 to 5");
        for (int i = 1; i <= 5; i++) {
            System.out.println(i + "\t\t" + (i*i));
        }

        // (2) Increment by other than 1
        // Print the square of odd numbers from 1 to 5.
        System.out.println("Square of odd numbers from 1 to 5");
        for (int i = 1; i <= 5; i += 2) {
            System.out.println(i + "\t\t" + (i*i));
        }

        // (3) "Backwards" for loop
        // Print the square of numbers from 5 to 1.
    }
}
```

```

System.out.println("Square of numbers from 5 to 1");
for (int i = 5; i > 0; i--) {
    System.out.println(i + "\t\t" + (i * i));
}

// (4) Nested for loops
// Print the multiplication tables from 1 to 5.
for (int i = 1; i <= 5; i++) {
    System.out.println("Multiplication table for " + i);
    for (int j = 1; j <= 10 ; j++) {
        System.out.printf("%2d x %2d = %2d\n", i, j, (i*j));
    }
}
}
}

```

Program output:

```

Square of numbers from 1 to 5
1   1
2   4
3   9
4   16
5   25
Square of odd numbers from 1 to 5
1   1
3   9
5   25
Square of numbers from 5 to 1
5   25
4   16
3   9
2   4
1   1
Multiplication table for 1
1 x  1 =  1
...
Multiplication table for 2
2 x  1 =  2
...
Multiplication table for 3
3 x  1 =  3
...
Multiplication table for 4
4 x  1 =  4
...
Multiplication table for 5
5 x  1 =  5
...

```



5.4 Changing control flow in loops

The `break` and the `continue` statements in Java have special significance when they are executed in any loop (`while`, `do-while` and `for` loops). These statements change the normal execution of a loop that we have discussed so far.

The `break` statement

Execution of the `break` statement in a loop body results in the rest of the loop body being skipped and the loop terminated: execution continues after the loop. Program 5.5 demonstrates the execution of a `break` statement in the body of the `for(;;)` loop at (1). When the value of the loop variable `i` becomes 4, the `break` statement at (3) in the `if` statement at (2) is executed. The remaining statement (4) in the loop body is skipped and the loop terminated, and execution continues at (5), after the loop.

A loop terminates when the loop condition becomes `false`. Using a `break` statement in a loop introduces another exit from the loop, and can make the program *unstructured*, i.e. difficult to understand, with all the problems that can entail. The code below shows a simple technique that can be used to ensure that exit is always from one place in the loop: through the testing of the loop condition. It introduces a flag variable (`done`) that is initialized to `false` (1) and always tested *first* in the loop condition (2), effectively making sure that the loop is not executed if the flag variable is `false`. An `if-else` statement (3) is used to set the flag variable and to control which statements are executed in each iteration of the loop body.

```
boolean done = false;                                // (1)
for (int i = 1; !done && i <= 5; i++) {           // (2)
    if (i == 4) {                                  // (3)
        System.out.println("Terminates the loop when i is equal to " + i);
        done = true;
    } else
        System.out.println(i + "\t\t" + (i * i));
}
```

The `continue` statement

Execution of the `continue` statement in a loop body results in the rest of the loop body being skipped. What happens next depends on the loop statement. In a `while`-loop execution will continue with the testing of the loop condition, but in a `for(;;)` loop execution will continue with the updating of the loop variable.

Program 5.5 demonstrates the execution of a `continue` statement in the body of the `for(;;)` loop at (6). When the value of the loop variable `i` becomes 3, the `continue` statement at (8) in the `if` statement at (7) is executed. The remaining statement (9) in the loop



body is skipped, and execution continues normally with updating of the loop variable at (6).

PROGRAM 5.5 Using **break** and **continue** statements in a **for(;;)** loop

```
// Illustrating use of break and continue statements
public class BreakAndContinue {
    public static void main(String[] args) {

        // for loop with a break statement.
        System.out.println("Square of numbers from 1 to 5");
        for (int i = 1; i <= 5; i++) {                                // (1)
            if (i == 4) {                                         // (2)
                System.out.println("Terminates the loop when i is equal to " + i);
                break;                                            // (3)
            }
            // Never executed when i == 4, as the loop terminates.
            System.out.println(i + "\t\t" + (i * i));           // (4)
        }

        // for loop with a continue statement.
        System.out.println("Square of numbers from 1 to 5");      // (5)
        for (int i = 1; i <= 5; i++) {                            // (6)
            if (i == 3) {                                         // (7)
                System.out.println("Skips rest of the loop body" +
                    " when i is equal to " + i);
                continue;                                         // (8)
            }
            // Skipped when i == 3, otherwise executed.
            System.out.println(i + "\t\t" + (i * i));           // (9)
        }
    }
}
```

5



Program output:

```
Square of numbers from 1 to 5
1   1
2   4
3   9
Terminates the loop when i is equal to 4
Square of numbers from 1 to 5
1   1
2   4
Skips rest of the loop body when i is equal to 3
4   16
5   25
```

BEST PRACTICES

The `break` and `continue` statements should be used with care in loops, as they can make the logic of the program difficult to understand.

5.5 Common pitfalls in loops

Infinite loop: `for(;;)`

Initialization and updating can be omitted in a `for(;;)` loop. The `for(;;)` loop then executes like a `while` loop, as only the loop condition is specified in the loop header. However, the loop condition can also be omitted, but this can have major consequences for the execution of the loop. Omitting the loop condition in a `for(;;)` loop means that the test to execute the loop body is always `true`, effectively creating an infinite loop:

```
for (;;) { // Always true
    System.out.println("Stop! I want to get off!");
}
```

A `break` statement can seem to offer a solution for terminating a loop that omits the loop condition, but as we have shown in the previous section, the program logic can be expressed effectively without the `break` statement, by formulating an appropriate loop condition for the loop.

5



One-off errors

A common type of programming error is a *one-off error*, i.e. the loop body is executed either one more or one less time than the correct number of iterations. Say we wish to add all the even numbers up to and including 10. In the code below we have a one-off error: the loop executes one less time than required. The start value of the variable `counter` is 2 and it is incremented by 2, therefore the value in the variable `counter` is always an even number. When the variable `counter` reaches 10, the loop condition (`counter < 10`) becomes `false`, and the loop terminates without adding the value 10. Changing the loop condition to (`counter <= 10`) remedies the situation.

```
int result = 0;
for (int counter = 2; counter < 10; counter += 2) {
    result += counter;
}
```

Errors in initialization

The loop variable is often not initialized correctly. In the code below, the variable `result` is initialized to 0. Its value does not change after the execution of the loop body. The loop condition remains `true` in the `do-while` loop. Most probably the variable should have been initialized to 1.

```
result = 0;
do {
    result *= 2;           // Always 0
} while (result < 100); // Infinite loop
```

Errors in the loop condition

It is important to make sure that the loop condition is formulated correctly. In the code below, since the start value is one and the variable counter is incremented by 2, the final value of counter is never 100, so that the loop can terminate. The loop condition (counter ≤ 100) would at least terminate the loop.

```
result = 0;
for (int counter = 1; counter != 100; counter += 2) {
    result += counter;
} // Infinite loop!
```

Optimizing loops

A loop body can be executed repeatedly. To a large extent, program execution time is dependent on how many times the statements in a loop body are executed. If we can make loop execution more effective, we can improve the execution time of the program. One source for such *optimisations* is computations in the loop that do not change value during execution, i.e. they have a constant value. Such computations are called *loop invariants*. By moving such computations out of the loop, we can reduce the execution time of the program.

```
double pi = 3.14;
for (double r = 0.0; r < 100.0; r++) {
    System.out.println("Radius: " + r +
                       ", Circumference: " + (2.0 * pi * r));
    // (2.0 * pi) does not change in the loop, and can be moved out of the loop
}
```

5



In the code above, the expression $2.0 * \pi$ is calculated during each iteration, even though its value does not change during the execution of the loop. The expression $2.0 * \pi$ is thus a loop invariant. We can move its computation out of the loop, so that we compute it just once and store it in a new variable (`factor`), as shown below. This variable can be used in the loop instead of the loop invariant.

```
double pi = 3.14;
double factor = 2.0 * pi;
for (double r = 0.0; r < 100.0; r++) {
    System.out.println("Radius: " + r + ", Circumference: " + (factor * r));
}
```

BEST PRACTICES

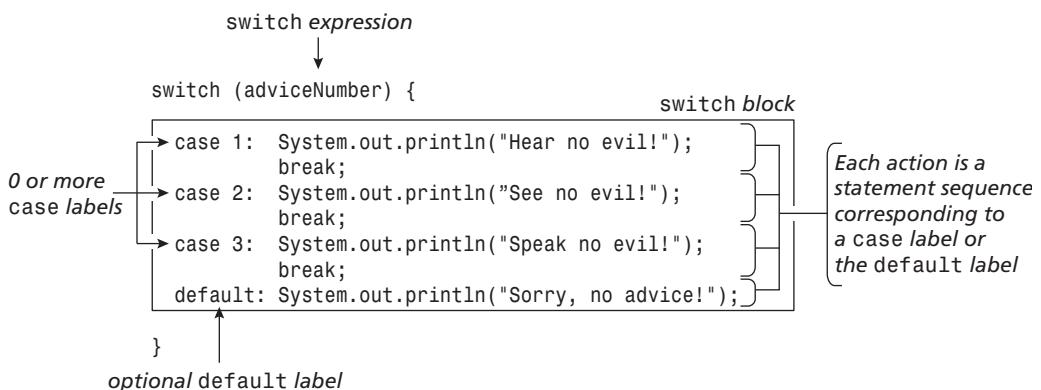
It is important to hand-simulate the loop so that the execution of the first and the last iterations are correct, to avoid one-off errors and unintentional infinite loops. This saves time that would otherwise be spent on *debugging* the program, i.e. modifying, compiling, executing and finding programming errors.

5.6 Multiple-selection statements

An `if` statement allows a single action to be selected and an `if-else` statement allows one of two actions to be selected, based on the value of a boolean expression. The `switch` statement allows one of many actions to be selected, based on the value of an expression. Figure 5.3 illustrates the form of a `switch` statement. The `switch` header specifies the `switch` expression, which in Figure 5.3 is the variable `adviceNumber`. The `switch` block specifies a list of `case` labels. Figure 5.3 shows three `case` labels. Each `case` label has a unique value and a corresponding action. Each action can be a sequence of statements. In Figure 5.3 the values of the three `case` labels are 1, 2 and 3.

During execution, the value of the `switch` expression is compared to the value of each `case` label in turn. If the value of the `switch` expression is equal to the value of a `case` label, the corresponding action is executed. If the value of the `adviceNumber` variable is 2 in Figure 5.3, the action corresponding to the `case` label with the value 2 is executed. The action in this case is a sequence of statements comprising a call to the method `println()` and a `break` statement. The call prints the string "See no evil!" in the terminal window, and the `break` statement transfers program control to *after* the `switch` statement. A `break` statement in a `switch` statement always skips the rest of the `switch` statement, terminating the execution of the `switch` statement.

FIGURE 5.3 Multiple selection statement: `switch`



The `default` label

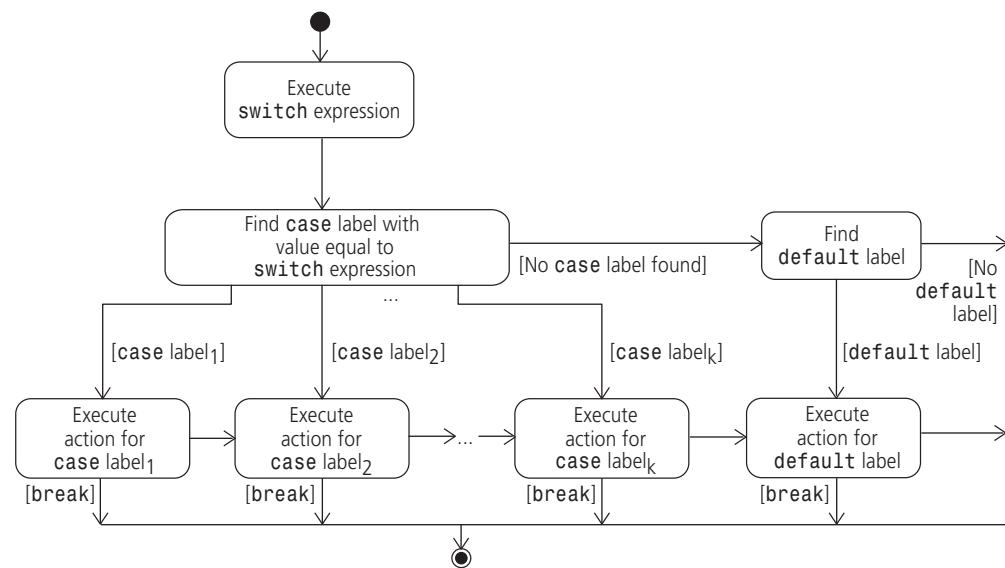
Figure 5.4 illustrates the execution of a `switch` statement. If no case label has the same value as the switch expression, the action corresponding to the `default` label is executed. The `default` label is optional in a `switch` statement. If no case label value is found and no `default` label is specified, the `switch` statement is skipped. If the value of the `adviceNumber` variable is 4 in Figure 5.3, the action corresponding to the `default` label is executed. This action is a call to the method `println()` to print the string "Sorry, no advice!". As this is the last statement in the `switch` block, program execution continues after the `switch` statement.

The `default` label and any other case labels can be specified in any order in a `switch` statement. The convention is to specify the case labels in increasing order of their values, and to specify the `default` label last.

BEST PRACTICES

In a `switch` statement, the convention is to specify the case labels in increasing order of their values, and to specify the `default` label last. Always include the `default` label, as it can help to catch errors in the program.

FIGURE 5.4 Executing `switch` statement



The `case` label values

The value of a case label is a *constant expression* that the compiler can evaluate. The values for the case labels must also be unique. The type of the value must be integer, including `char`, but not `long`. Boolean and floating-point values are also not permitted for case labels,

and neither are string literals. In a later chapter we will see how constant values defined by enumerated types can be used as case label values (see *Enumerated types* on page 192).

The type of the case label value must be compatible with the type of the switch expression for their values to be compared for *equality*. This is in contrast to an if statement, in which the condition is an arbitrary boolean expression.

Program 5.6 is a new version of Program 5.3 to spell out a phone number, in which a switch statement is used instead of an if statement. The type of the case label values is char. Each case label translates a digit. The corresponding action ends with a break statement to terminate the switch statement after a digit is processed.

Variables declared in a switch block cannot be used outside the block. If you need to access a local variable outside the block, the variable must be declared before the switch statement. In Program 5.6, the variable outputPhoneString is an example of such a variable.

PROGRAM 5.6 Using a switch statement

```
import java.util.Scanner;
// Program spells out a telephone number: switch statement version
public class TelephoneII {
    public static void main(String[] args) {
        System.out.print("Enter the telephone number (as 55584152): ");
        Scanner keyboard = new Scanner(System.in);
        String inputPhoneString = keyboard.next();
        String outputPhoneString = "";
        for (int i = 0; i < inputPhoneString.length(); i++) {
            char aChar = inputPhoneString.charAt(i);
            switch (aChar) {
                case '1':
                    outputPhoneString += "one ";
                    break;
                case '2':
                    outputPhoneString += "two ";
                    break;
                case '3':
                    outputPhoneString += "three ";
                    break;
                case '4':
                    outputPhoneString += "four ";
                    break;
                case '5':
                    outputPhoneString += "five ";
                    break;
                case '6':
                    outputPhoneString += "six ";
                    break;
                case '7':
                    outputPhoneString += "seven ";
```



```

        break;
    case '8':
        outputPhoneString += "eight ";
        break;
    case '9':
        outputPhoneString += "nine ";
        break;
    case '0':
        outputPhoneString += "zero ";
        break;
    default:
        System.out.println(aChar + " is not a digit!");
    } // end switch
} // end for
System.out.println(outputPhoneString);
}
}

```

Program output:

```
Enter the telephone number (as 55584152): 55584152
five five five eight four one five two
```

Falling through case labels

We need not use the block notation, [], to enclose the sequence of statements for an action corresponding to a case label. Enclosing these statements in a block does *not* affect the control flow in a `switch` statement. However, the execution of a `break` statement as part of an action has a profound effect on the control flow in a `switch` statement: as we have seen, it terminates the `switch` statement. If a `break` statement is not the last statement that is executed for an action, execution can continue with the next action that is specified. In other words, unless a statement transfers control elsewhere, control will “fall through” to subsequent statements, regardless of whether they have a label or not. Note also that a `break` statement is not part of the syntax for a `switch` statement.

```

final int YES = 1;
final int NO = 0;
int whatToDo;
...
switch(whatToDo) {
    case YES:
        System.out.print("Falling ");
    case NO:
        System.out.print("through. ");
    default:
        System.out.println("Come on down.");
}

```

In the code above, if the value of the variable `whatToDo` is 1, the following is printed:

Falling through. Come on down.



If the value of the variable `whatToDo` is 0, the following is printed:

through. Come on down.

For any other integer value in the variable `whatToDo`, only the string "Come on down." is printed.

A consequence of falling through case labels is that several case labels can share the same action. Program 5.7 illustrates this special case. The program prints the season depending on the month of the year. Since it is winter during December, January and February (at least in some parts of the World), case labels with corresponding month numbers 12, 1 and 2 have the same action: print the string "It's winter. ".

PROGRAM 5.7 Common action for case labels in a `switch` statement

```
import java.util.Scanner;
public class Seasons {
    public static void main(String[] args) {
        System.out.print("Enter a month number [1-12]: ");
        Scanner keyboard = new Scanner(System.in);
        int monthNumber = keyboard.nextInt();
        switch (monthNumber) {
            case 12: case 1: case 2:// Common action for several case labels
                System.out.println("It's winter!");
                break;
            case 3: case 4: case 5:
                System.out.println("It's spring!");
                break;
            case 6: case 7: case 8:
                System.out.println("It's summer!");
                break;
            case 9: case 10: case 11:
                System.out.println("It's autumn!");
                break;
            default:
                System.out.println(monthNumber + " is not a valid month number!");
        } // end switch
    }
}
```

Program output:

```
Enter a month number [1-12]: 4
It's spring!
```



BEST PRACTICES

Judicious use of the `break` statement to terminate the actions for a `case` label helps to prevent unintentional fall through in a `switch` statement.

5.7 Review questions

1. What are the four statements in Java that we can use to increment the value in an integer variable `i` by 1?

2. What will the following code print?

```
int counter = 4;
System.out.println(counter++);
System.out.println(++counter);
System.out.println(counter--);
System.out.println(--counter);
System.out.println(counter);
```

3. If the number of iterations is known in advance, would you choose a `for(;;)` loop or a `while` loop?

4. Which part (initialization, loop condition, updating) in the `for(;;)` loop header can be omitted?

5. Which statements are true about the `for(;;)` loop? Assume that the loop body does not execute any statement that transfers control out of the loop.

- a Initialization is done once.
- b Loop condition is tested at least once.
- c Loop body is executed at least once.
- d If the loop body is executed, the updating and the loop condition are always executed.

6. Rewrite the following code using a `for(;;)` loop. How many times is the loop body executed?

```
int i = 10, sum = 0;
while (i >= 5) {
    sum += i;
    i -= 2;
}
```

7. What is the value of the variable `i` after the execution of the `for(;;)` loop? How many times does the loop body execute?

- a `int i;`
`for (i = 0; i < 10; i += 2)`



- System.out.println(i);
- b for (int i = 0; i < 10; i += 3)
8. System.out.println(i); Which for(;;) loops can be used to print the string "Move it!" three times in the terminal window?
- a for (int i = 1; i <= 3; i++);
System.out.println("Move it!");
- b for (int i = 1; i <= 3; i++)
9. System.out.println("Move it!"); Which statements are true about nested for(;;) loops?
- a The outer loop completes all its iterations for every iteration of the inner loop.
- b The inner loop completes all its iterations for every iteration of the outer loop.
- c The total number of times the inner loop executes is the product of the number of times each outer loop is executed.
10. The total number of times the inner loop executes is the sum of the number of times each outer loop is executed. Which statements are true about the switch statement?
- a A default label must always be included in a switch statement.
- b Boolean values and floating-point values cannot be specified as case labels.
- c A string literal can be specified as a case label.
11. A break statement is part of the switch statement. Rewrite the following code using a switch statement:

```
if (i == 10 || i == 20)
    System.out.println("10 or 20");
else if (i == 15)
    System.out.println("15");
else
    System.out.println("Not valid");
```

12. Which switch statements will not compile? If so, explain why not. Assume the following declarations:

```
int result = 100, finalValue = 12;
char letter = 'k';
boolean flag = false;
String msg = "";
```

a switch(result) {
 case 100:
 msg = "Falling ";
 case 200:
 msg += "through";
}



```
b switch(letter) {  
    case '0': case 'o':  
        System.out.println("0");  
        break;  
    case 'K': case 'k':  
        System.out.println("K");  
        break;  
}  
  
c switch(finalValue) {  
    default:  
        System.out.println("Not ok");  
        break;  
    case 100:  
        System.out.println("Ok");  
}  
  
d switch(flag) {  
    case true:  
        System.out.println("It's true.");  
        break;  
    case false:  
        System.out.println("It's false.");  
        break;  
    default:  
        assert false: "Can never happen.";  
}  
  
e switch(finalValue % 2) {  
    case 1:  
        System.out.println("Odd");  
        break;  
    case 0:  
        System.out.println("Even");  
        break;  
    default:  
        assert false: "Can never happen.";  
}  
  
f switch(finalValue / 2.0) {  
    case 5:  
        System.out.println("Accepted");  
        break;  
    default:  
        System.out.println("Rejected");  
}  
  
g switch(result > 0) {  
    case 5: case 10:  
        System.out.println("Too low");  
        break;  
    case 30: case 5:
```

```
        System.out.println("Too high");
    }
```

5.8 Programming exercises

1. Write a program that prints the even numbers from 1 to 100. Use a `for(;;)` loop.
2. Write a program that prints the even numbers from 1 to 100 backwards. Use a `for(;;)` loop that counts backwards.
3. Write a program that asks the user for a string and then prints how many times the first character occurs in the string. Use a `for(;;)` loop to solve the problem.

Example:

```
Enter a string: hahaha
The letter 'h' occurs 3 times.
```

4. Write a program that reads a sentence (i.e. a line of input) from the keyboard and reports the number of lowercase and uppercase letters in the input.

Example:

```
Enter your sentence: BIG MOUTH, small feet.
Number of lowercase characters: 9
Number of uppercase characters: 8
Number of lowercase characters is greater than the number of
uppercase characters.
```

5. A *palindrome* is a string that has identical character sequence regardless of whether you read it backwards or forwards. For example, "aha", "abba" and "00700" are all palindromes according to this definition. The string can have several words, and you can choose to ignore the spaces between the words if you wish. For example, "roma amor" is a palindrome. Write a program that reports whether a line of input read from the keyboard is a palindrome. Use a `for(;;)` loop for this purpose.
6. Write a program that reads two values from the keyboard. The numbers represent an interval on the Celsius temperature scale. The program prints the values in the corresponding interval on the Fahrenheit temperature scale. Use the following formula, where `fTemp` and `cTemp` represent the temperature in degrees Fahrenheit and Celsius, respectively:

$$fTemp = (9.0/5.0) * cTemp + 32.0;$$

Example:

```
Enter the interval for degrees Celsius: 10.0 100.0
Celsius      Fahrenheit
10.0          50.0
11.0          51.8
...
99.0          210.2
```

100.0

212.0

7. Write a program that computes the *factorial* of an integer, i.e. $n! = n \times (n-1)! = 1 \times 2 \times 3 \times 4 \times \dots \times (n-1) \times n$, where $n > 0$. What is the largest value of n that your program works correctly for?
8. Embed the `switch` statement from Figure 5.3 in a loop to give advice to the user. The program reads the advice number from the keyboard.
9. Rewrite the `switch` statement in Program 5.7, using an `if` statement.
10. Write a program that gives change in notes and coins. UN currency comprises UN dollars (UND) and cents. This fictitious currency has notes for the following denominations: 1000 UND, 500 UND, 200 UND, 100 UND, and 50 UND. There are coins for 20 UND, 10 UND, 5 UND, 1 UND, and 50 cents (100 cents equals 1 UND).

An amount that is less than 25 cents is rounded down to 0 cents. An amount greater than or equal to 25 cents, but less than 75 cents, is rounded to 50 cents. Lastly, an amount greater than 75 cents, but less than 1 UND, is rounded up to 1 UND.

The program always tries to give change such that notes and coins delivered always have the highest possible denomination. For example, 5732.60 UND give the following change:

```
Number of 1000 UND notes: 1
Number of 500 UND notes: 1
Number of 200 UND notes: 1
Number of 20 UND coins: 1
Number of 10 UND coins: 1
Number of 1 UND coins: 2
Number of 50 cents coins: 1
```





Arrays

LEARNING OBJECTIVES

By the end of this chapter you will understand the following:

- Using arrays to organize a collection of values.
- Declaring array references, creating arrays and using these references to store values in and retrieve values from arrays.
- Initializing an array with specific values when it is created.
- Iterating over an array to access each value successively.
- Creating and using multidimensional arrays.
- Writing code for common operations on arrays.
- Generating pseudo-random numbers using the `Random` class.

INTRODUCTION

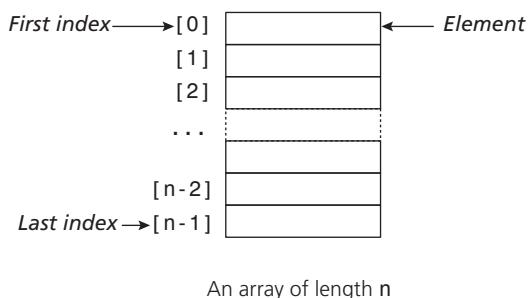
We often need to organize values so that they can be processed as a group, a unit or a collection. Such an organisation of values is called a *data structure*. An array is a simple form of data structure. For example, we can create an array to store the number of text messages sent each day of the week from a mobile phone. This array would collectively represent the weekly figures for the number of text messages sent from a mobile phone. Such an array would hold seven integers, one for each day of the week, indicating the number of text messages sent on a particular day. How would one find in such an array, for example, the number of messages sent on Wednesday for a given week? Or the total of text messages sent during the week? To answer such questions, we first have to look at the properties of arrays.

Finally in this chapter we discuss pseudo-random numbers. As an example we simulate the throwing of two dice, and storing the frequency of each throw value in an array.

6.1 Arrays as data structures

An *array* is a fixed-length sequence of elements in which all elements have the same type. An array is thus a data structure that has a fixed *length*, indicating the number of values that can be stored in the array. Each *array element* can store a value. The type of the elements is called the (*array*) *element type*. Since an array is a sequence, it means that the elements are numbered. Their position in the array is given by a position number, called the *index*. Positions in an array are always numbered from 0 upwards. The lowest index, 0, indicates the position of the first element, and if the length of the array is n , the last element in the array has index $n-1$. Figure 6.1 illustrates an array.

FIGURE 6.1 An array



An array of length n

In our example of an array for recording the number of text messages sent from a mobile phone for each day of the week, the array length is 7 as there are seven days in a week, and the element type is `int`, as we want to store an integer representing the number of text messages sent from a mobile phone. If we decide that index 0 indicates the element with the text messages sent on the first day of the week (for example Monday), then index 2 indicates the element with the number of text messages sent on Wednesday.

An array with integers that represent the number of text messages sent for each day of the week is an example of an array with values of a primitive data type, in this case `int`. We can create arrays of integers, floating-point numbers, characters and Boolean values. The element type in each case is a specific primitive data type.

We can also create *arrays of objects*. Elements in such an array are thus references to objects, and the element type is a reference type (for example, a specific class). Note that the array does not actually contain the objects, only references to objects. Arrays are themselves objects in Java, and since array elements can be references to objects, we can create *arrays of arrays*, i.e. arrays in which the elements are references to other arrays.

In the rest of the chapter we look at how arrays can be *declared*, *created*, *initialized* and *used* in Java.



6.2 Creating and using arrays

Declaring array reference variables

An *array reference variable* is a reference that can only refer to objects that are arrays. In a reference declaration we have to specify the reference type. For arrays, the reference type is specified using brackets, `[]`, and is called the *array type*. For example, we can declare the following array reference variable `noOfTextMessages` that can refer to an array of integers:

```
int[] noOfTextMessages;
```

Note that without the brackets, the name `noOfTextMessages` would be declared as a variable of type `int`, and not a variable of an array type.

The declaration above does *not* create any array. It only declares that the reference `noOfTextMessages` can refer to an array. The array type is `int[]`, and the element type is `int`. No array length is specified either. The declaration says that the reference `noOfTextMessages` can refer to an arbitrary array as long as its element type is `int`, regardless of the length of the array.

Creating arrays

We used the `new` operator, together with a constructor call, to create objects from classes. We also use the `new` operator to create an array, but we must specify the element type and the desired array length:

```
noOfTextMessages = new int[7]; // An array to store 7 int values.
```

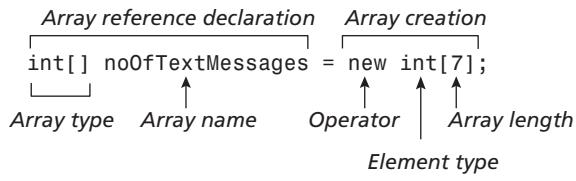
The expression on the right-hand side creates an array that can store seven `int` values. The array length can be any expression that evaluates to a non-negative integer value. The length of the array cannot be changed after the array has been created. After the execution of this declaration statement, the reference `noOfTextMessages` will refer to the array and can be used to manipulate the array (see Figure 6.2b).

Default initialization

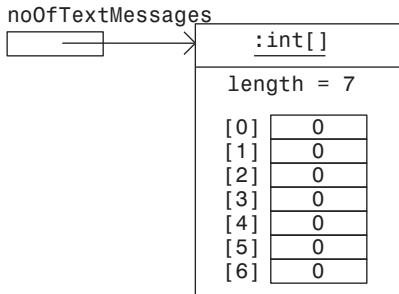
The array creation expression in the previous section says nothing about which seven `int` values are stored in the array, so all elements in the array will be assigned a value of 0. The rule is that when an array is created as shown, the elements are automatically initialized to the *default value* for the element type. The default value for the `int` type is 0 (see Figure 6.2b). Table 6.1 shows the default values for the primitive and reference types. If we create an array of `boolean` values, i.e. `boolean[]`, all the elements will be assigned the value `false`.



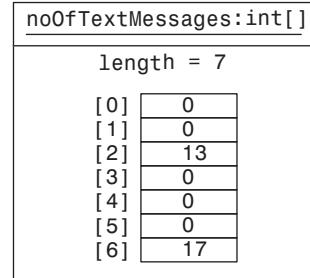
FIGURE 6.2 Declaration, creation and default initialization of arrays



(a) Array reference declaration and creation



After executing:
`noOfTextMessages = new int[7];`



After executing:
`noOfTextMessages[2] = 13;`
`noOfTextMessages[6] = noOfTextMessages[2] + 4;`

(b) Notation showing explicit array reference

(c) More compact notation for an array

TABLE 6.1 Default values for types

Type	Default value
boolean	false
char	'\u0000'
Integer (int, long)	0
Floating-point (double)	+0.0d
All reference types	null

6



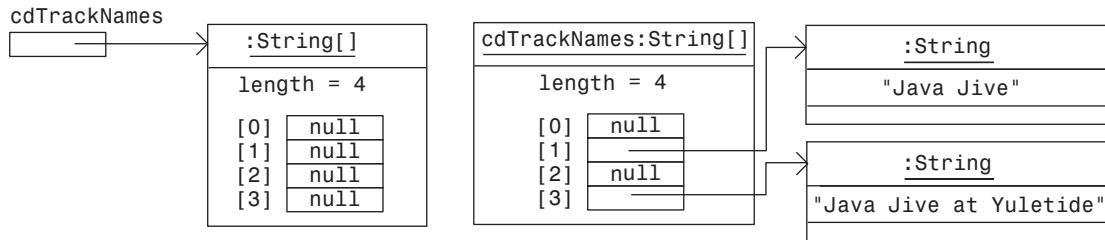
Arrays of objects

In the same way as creating arrays of primitive types, we can create arrays of objects. The following declaration statement combines the declaration of the array reference variable and the creation of the array:

```
String[] cdTrackNames = new String[4]; // An array of 4 String references.
```

After execution of the above statement, the reference `cdTrackNames` will refer to an array of length four, whose elements can refer to `String` objects (see Figure 6.3a). The elements are implicitly initialized to the default value for reference types, i.e. the value `null` (see Table 6.1). The array type is `String[]`, and the element type is `String`.

FIGURE 6.3 Array of objects



After executing:

```
String[] cdTrackNames = new String[4];
```

(a) Creating an array of `Strings`

After executing:

```
cdTrackNames[1] = "Java Jive";
cdTrackNames[3] = cdTrackNames[1] + " at Yuletide";
```

(b) Initialising an array of `Strings`

The `length` field

Each array has a field called `length` whose value is the array length. The value is set when the array is created, and cannot be changed. The value of this field can be accessed using the dot notation:

```
System.out.println(noOfTextMessages.length); // Prints 7.
System.out.println(cdTrackNames.length); // Prints 4.
```

As opposed to the method `length()` in the `String` class, the name `length` is a field in every array.

Accessing an array element

To access an element in an array we need to specify both the array reference and the index of the element in the array. We can access the value in the element that holds the number of text messages sent on Wednesday using the expression `noOfTextMessages[2]`. Similarly, the expression `noOfTextMessages[6]` indicates the number of text messages sent on Sunday. The index is always specified between brackets.

We can change the element values using the assignment operator:

```
noOfTextMessages[2] = 13; // Index 2 for Wednesday.
```

After execution of this assignment statement, the number of text messages sent on Wednesday is now thirteen. If the number of text messages sent on Sunday is four more than the number sent on Wednesday, we can update the number of text messages sent on Sunday as follows:

```
noOfTextMessages[6] = noOfTextMessages[2] + 4; // Index 6 for Sunday.
```

Figure 6.2c illustrates the array after execution of these two statements. The expression `noOfTextMessages[2]` behaves exactly like a variable. As the expression specifies an array reference and an index, we call it an *indexed variable*.

We can imagine the array referenced by the array reference `noOfTextMessages` as consisting of seven integer variables. One might be tempted to declare seven different integer variables to maintain the seven values indicating the number of text messages sent each day of



the week, but that is not a good idea. What if we need to maintain several thousand elements in an array?

We can access elements in an array of objects similarly. The expression `cdTrackNames[1]` indicates the element with index 1 in an array of `String` objects. Each such indexed variable is like any other reference, and has either the value `null`, or refers to an object of the element type. The following statements show examples of accessing elements of an array of `String` objects:

```
cdTrackNames[1] = "Java Jive";
cdTrackNames[3] = cdTrackNames[1] + " at Yuletide"; // "Java Jive at Yuletide"
```

Figure 6.3b illustrates the array after execution of these two assignment statements.

Array bounds

So far we have used an integer literal as an index, but in fact the index can be any arbitrary expression that evaluates to an `int` value. The index value must be in the range of valid index values for an array, i.e. the index value must be greater than or equal to 0 and less than the array length ($0 \leq$ index value $<$ array length). At runtime, the index value is always checked before accessing the array. An invalid index results in an out-of-bounds error: the program is terminated after reporting an exception of type `ArrayIndexOutOfBoundsException`. See the output from Program 6.1.

Array aliases

As arrays are objects, we can create aliases to arrays:

```
int[] messageCounters = noOfTextMessages;
String[] trackTitles = cdTrackNames;
```

The references `noOfTextMessages` and `messageCounters` are aliases to the same array. The same applies to the references `cdTrackNames` and `trackTitles`. Any alias to an array can be used to manipulate the array. Therefore the expressions `noOfTextMessages[2]` and `messageCounters[2]` both refer to the same element in the array, i.e. the number of text messages sent on Wednesday.

Note that array variables are references, and assigning one reference to another does *not* copy the elements of one array to another. In Section 6.7 we will see how we can copy element values from one array to another.

Alternate notation for array declaration

There is an alternate notation for declaring array references in which the brackets are placed after the array reference variable:

```
int noOfTextMessages[]; // Alternate notation for array reference declaration
```

Both forms declare the reference `noOfTextMessages` to have the type `int[]`, i.e. an array of `int` values. The difference between the two forms becomes apparent when a list of variables are specified in the declaration:

```
int[] arrayA, arrayB, arrayC; // (1)
```

```
int arrayA[], arrayB[], arrayC[]; // (2)
int arrayA[], arrayB, arrayC[]; // (3)
```

(1) and (2) are equivalent: all three references have the type `int[]`. In (3) only references `arrayA` and `arrayC` have the type `int[]`, whereas the variable `arrayB` is of type `int`. The standard convention is the form in (1), which we follow in this book.

BEST PRACTICES

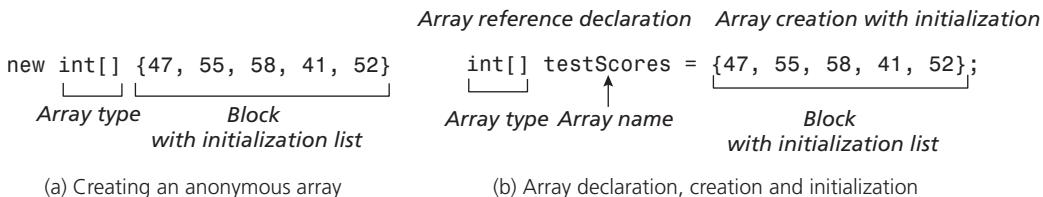
The prevailing practice is to declare array reference variables using the following notation:

```
int[] arrayRefName;
```

6.3 Initializing arrays

We saw that when we create an array using the `new` operator and specifying the array length, the elements of the array are initialized to the default value for the element type. If we wish to store other values in the elements when we create the array, we must explicitly specify the values. This is called (*explicit*) *array initialization*. Figure 6.4 shows two language constructs that initialize an array when it is created. These constructs are practical when we wish to create small-size arrays with specific values.

FIGURE 6.4 Array creation with initialization



The form in Figure 6.4a is an expression that creates an *anonymous array* of the specified array type, that is, one initialized with the values specified within the block. The length of the array is implicitly given by the number of values in the block. The values in the block all have the same element type (`int`), and these values are stored in the array with the first value at index 0, the second value at index 1 and so on. This form is called an “anonymous array” because it does not require the reference value of the array be assigned to an array variable. A typical use for an anonymous array is when we need to create a small array “on the fly”, for example, as an actual parameter in a method call.

Like any other reference value, the reference value returned by the array creation expression can be assigned to an array variable:

```
int[] testScores; // (1) Only declaration, no array creation.
// Array creation, explicit initialization and assignment
```



```
testScores = new int[] {47, 55, 58, 41, 52};           // (2) 5 initial values
```

Of course we can combine the declaration (1) and the assignment (2) as before:

```
// Array declaration, creation, explicit initialization and assignment  
int[] testScores = new int[] {47, 55, 58, 41, 52}; // (3) 5 initial values
```

In (3) the array type (`int[]`) is repeated on both sides of the assignment. This is superfluous. In this particular case, (3) can be simplified to the form shown in Figure 6.4b:

```
int[] testScores = {47, 55, 58, 41, 52};           // (4) Simplified form
```

The `new` operator and the array type are omitted on the right-hand side in (4). The form in Figure 6.4b can only be used when we *declare* an array variable, whereas the form in Figure 6.4a can be used anywhere an array reference is allowed. For example, the form in Figure 6.4b cannot be used in an assignment statement (compare with (2) above):

```
testScores = {47, 55, 58, 41, 52}; // (5) Compile time error!
```

Below we give some more examples of array initialization for different kinds of arrays:

```
// (1) An integer array with 8 elements  
int[] intArray = {1, 3, 49, 55, 58, 41, 52, 3146};  
// (2) A boolean array with 4 elements  
boolean[] booleanArray = new boolean[] {true, false, false, true};  
// (3) A character array with 4 characters  
char[] charArray = {'J', 'a', 'v', 'a'};  
// (4) An array with 4 floating-point values  
double fpArray = new double[] {25.0, 3.14, 1.5, 0.75};  
// (5) An array with 4 strings  
String[] pets = {"crocodiles", "elephants", "crocophants", "eleddiles"};  
// (6) An array with 3 strings,  
//      but only 2 elements are initialized with non-null values.  
pets = new String[] {"cat", null, "dog"};
```

Note the use of the `null` value in the initialization list in (6). Element `pets[1]` does not refer to an object. It's a common programming error to use such a reference to refer to an object.

Program 6.1 demonstrates array declaration, creation, initialization and assignment discussed so far. The assertions in the program show the assumptions we can make about arrays.

PROGRAM 6.1 Array declaration, creation, initialization and assignment

```
// Array initialisation  
public class ArrayInitialisation {  
    public static void main(String[] args) {  
  
        final int NO_OF_TESTS = 5;  
  
        // Array declaration:  
        int[] testScores; // (1) Only declaration, no array creation
```

```

// Array creation, default initialisation and assignment:
testScores = new int[NO_OF_TESTS];           // (2) Array length specified
assert(testScores != null);
assert(testScores.length == NO_OF_TESTS);
assert(testScores[0] == 0);                   // First value
assert(testScores[NO_OF_TESTS - 1] == 0); // Last value
// and the other elements are also initialised to the default value 0.

// Combined (1) and (2).
// Array declaration, creation, default initialisation and assignment:
int[] testScoresII = new int[NO_OF_TESTS];

// Array declaration:
int[] testScoresIII; // (3) Only declaration, no array creation
// Array creation, explicit initialisation and assignment:
testScoresIII = new int[] {47, 55, 58, 41, 52}; // (4) Anonymous array
assert(testScoresIII.length == NO_OF_TESTS);
assert(testScoresIII[0] == 47);                // First value
assert(testScoresIII[NO_OF_TESTS - 1] == 52); // Last value
// and the other elements are also explicitly initialised accordingly.

// Combined (3) and (4)
// Array declaration, creation, explicit initialisation
// and assignment:
int[] testScoresIV = new int[] {47, 55, 58, 41, 52};
int[] testScoresV = {47, 55, 58, 41, 52}; // Simplified form
// testScoresV = {47, 55, 58, 41, 52};      // Compile time error!

System.out.println(testScoresV[NO_OF_TESTS]); // Out-of-bounds error
}
}

```

Program output:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
at ArrayInitialization.main(ArrayInitialization.java:37)
```

6

BEST PRACTICES

Array elements are initialized to the default value of their type if no explicit initialization is specified when the array is created. As the default type of a reference type is `null`, elements of an array of objects must explicitly be initialized before use to avoid a `NullPointerException`.

6.4 Iterating over an array

A common task in programming is accessing elements in an array successively to perform the same operation on each element of the array. For example, to calculate the total number of text messages sent during the week in our example, we would have to add each element of the array that represents the number of text messages sent each day. Accessing elements of an array successively is called *iteration* over the array.

To iterate over an array with n elements, we start with the first element at index 0, continue successively with the elements at index 1, 2 and so on until the last element at index $n-1$ has been processed.

It should not come as a surprise that a counter-controlled `for(;;)` loop is convenient for iterating over arrays. Each iteration of the loop accesses a new element of the array. The index, represented by the loop variable, is initialized to 0 in the initialization part of the `for(;;)` loop. After each iteration the index is incremented by 1 in the update part of the `for(;;)` loop. But what should the loop test be? As long as the index is *less* than n , not all the elements have been accessed. The loop body accesses the current element using the current value of the index:

```
// Code pattern for iterating over an array.  
for (int index = 0; index < array.length; index++) {  
    // ... current element given by array[index] ...  
}
```

The code pattern above guarantees that the loop will iterate through all the elements of the array successively: the index value will always be valid, and using it to access the designated element in the array will not result in an out-of-bounds error.

Program 6.2 illustrates the use of arrays by looking at four simple problems related to the array `noOfTextMessages`.

Comparing all values in an array with a given value

6



Problem (1) in Program 6.2 deals with finding how many days have a number of text messages equal to or greater than a specified lower bound. The lower bound is read from the keyboard. We use a counter (`noOfDays`) to count the number of days that satisfy this criterion. During each iteration of the loop we compare whether the current day (given by the loop variable `index`) has a number of text messages (given by `noOfTextMessages[index]`) equal to or greater than the lower bound. If this is the case, we increment the counter.

Finding the lowest value in an array

Problem (2) in Program 6.2 deals with finding the lowest number of text messages sent during the week. We use a variable (`lowestNoOfTextMessages`) to keep track of the lowest number of text messages sent on a day as we iterate over the array. We initialize the variable `lowestNoOfTextMessages` with the value of the first day of the week (`noOfTextMessages[0]`). Assuming that the first day has the lowest number of text messages sent during the week, we begin the loop iteration with an index of 1. In this case we cannot use a value

that is lower than any of the values in the array, because the lowest value has to be one of the values in the array. Alternatively, we can use a value that is greater than any of the values in the array, for example `Integer.MAX_VALUE`, and begin the loop iteration with an index of 0.

During each iteration of the loop we test whether the number of text messages for the current day (given by `noOfTextMessages[index]`) is less than the current lower bound (given by the variable `lowestNoOfTextMessages`). If this is the case, then the current day has the lowest number of text messages so far, and we must update the variable `lowestNoOfTextMessages` to reflect the new lower bound.

Finding the highest value in an array

Problem (3) in Program 6.2 deals with finding the highest number of text messages sent during the week, and the days on which that number of messages were sent. We first find the highest number of text messages sent during the week, similar to problem (2). To find the days with the highest number of text messages, we iterate over the array again, printing the name of the day if the number of text messages sent on this day equals the maximum number of text messages sent on a specific day during the week.

Adding the values in an array

Problem (4) in Program 6.2 deals with finding the total number of messages sent during the week. We use a counter (`totalNoOfTextMessages`) to accumulate the value for each day. After each iteration of the loop, the variable `totalNoOfTextMessages` has the current total.

PROGRAM 6.2 Iterating over an array

```
import java.util.Scanner;
public class ArrayIteration1 {

    public static void main(String[] args) {
        // Array with the names of week days
        String[] daysOfTheWeek = {"Monday", "Tuesday", "Wednesday",
                                  "Thursday", "Friday", "Saturday", "Sunday"};
        // Array with the no. of text messages sent during a day.
        int[] noOfTextMessages = new int[7];

        // Explicit initialisation
        noOfTextMessages[0] = 20;                                // Monday
        noOfTextMessages[1] = 12;                                // Tuesday
        noOfTextMessages[2] = 13;                                // Wednesday
        noOfTextMessages[3] = noOfTextMessages[1];                // Thursday
        noOfTextMessages[4] = 10;                                // Friday
        noOfTextMessages[5] = noOfTextMessages[0];                // Saturday
        noOfTextMessages[6] = noOfTextMessages[2] + 4;            // Sunday

        // Setup to read from the terminal window.
```



```
Scanner keyboard = new Scanner(System.in);

// Problem (1) Find how many days have their number of text messages
// equal to or greater than a specified lower bound.
System.out.print("Enter the lower bound for " +
                 "the no. of text messages: ");
int lowerBound = keyboard.nextInt();
int noOfDays = 0;
for (int index = 0; index < noOfTextMessages.length; index++) {
    if (noOfTextMessages[index] >= lowerBound) {
        noOfDays++;
    }
}
System.out.println("No. of days with more than " + lowerBound +
                   " text messages: " + noOfDays);

// Problem (2) Find the lowest number of messages sent during the week.
int lowestNoOfTextMessages = noOfTextMessages[0];
for (int index = 1; index < noOfTextMessages.length; index++) {
    if (lowestNoOfTextMessages > noOfTextMessages[index]) {
        lowestNoOfTextMessages = noOfTextMessages[index];
    }
}
System.out.println("Lowest no. of text messages: " +
                   lowestNoOfTextMessages);

// Problem (3) Find the highest no. of text messages sent during
// the week and the days on which that number of messages were sent.
// Find the highest no. of text messages sent during a day.
int highestNoOfTextMessages = 0;
for (int index = 0; index < noOfTextMessages.length; index++) {
    if (highestNoOfTextMessages < noOfTextMessages[index]) {
        highestNoOfTextMessages = noOfTextMessages[index];
    }
}
System.out.println("Highest no. of text messages: " +
                   highestNoOfTextMessages);
// Print all days with the highest no.of text messages sent.
System.out.println("Days with the highest no. of text messages:");
for (int index = 0; index < noOfTextMessages.length; index++) {
    if (highestNoOfTextMessages == noOfTextMessages[index]) {
        System.out.println(daysOfTheWeek[index]);
    }
}

// Problem (4) Find the total number of messages sent during the week.
int totalNoOfTextMessages = 0;
for (int index = 0; index < noOfTextMessages.length; index++) {
    totalNoOfTextMessages += noOfTextMessages[index];
}
```



```

        System.out.println("Total no. of text messages: " +
                           totalNoOfTextMessages);
    }
}

```

Program output:

```

Enter the lower bound for the no. of text messages: 13
No. of days with more than 13 text messages: 4
Lowest no. of text messages: 10
Highest no. of text messages: 20
Days with the highest no. of text messages:
Monday
Saturday
Total no. of text messages: 104

```

Iterating over an array of objects

Program 6.3 illustrates iteration over an array of `String` objects. The program prints all the track titles that contain the word "Java". In each iteration of the loop (1) the current track name is given by the reference `cdTrackNames[trackNumber]`, where `trackNumber` is the current index in the array `cdTrackNames`. For each track name, we call the method `indexOf()` from the `String` class with the literal "Java" as argument. If the return value is not -1, this means that the word "Java" occurs in the current track name and the track name can be printed.

PROGRAM 6.3 Iterating over an array of strings

```

// Array iteration
public class ArrayIteration2 {
    public static void main(String[] args) {
        String[] cdTrackNames = {
            "The Ballad of the Loop Variable",
            "Do the Java Jive",
            "My Loop will go on",
            null
        };
        cdTrackNames[3] = cdTrackNames[1] + " at Yuletide";

        // Print all track names with the word "Java" in them.           // (1)
        for (int trackNumber = 0;
              trackNumber < cdTrackNames.length;
              trackNumber++) {
            if(cdTrackNames[trackNumber].indexOf("Java") != -1) {
                System.out.println(cdTrackNames[trackNumber]);
            }
        }
    }
}

```



Program output:

```
Do the Java Jive  
Do the Java Jive at Yuletide
```

BEST PRACTICES

Iterating over an array using the following code ensures that *all* elements of the simple array are accessed, and avoids the `IndexOutOfBoundsException`:

```
for (int i = 0; i < arrayName.length; i++) { /* ... arrayName[i] ... */ }
```

6.5 Multidimensional arrays

Arrays we have seen so far are called *simple* or *one-dimensional arrays*, and only *one index* is required to identify elements in the array. We used a simple array to record the number of text messages sent from a mobile phone for each day of the week. What if we need to keep the same statistics for several mobile phones? We could use many simple arrays, one for each mobile phone, and have different array names to distinguish between them. This solution would not be viable if we needed to maintain statistics for many mobile phones. Using many different array names for the simple arrays would soon become unwieldy. We would like a way to maintain several simple arrays as values in a data structure. *Multidimensional arrays* provide the answer.

In this section we look closely at how we can declare, create, initialize and use multidimensional arrays.

Creation and initialization of multidimensional arrays

6

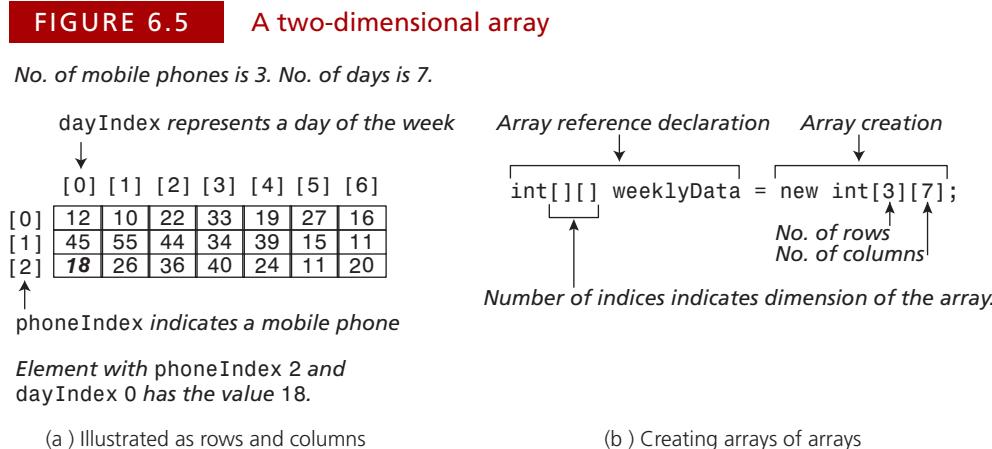


We want to maintain the weekly data about text messages sent from several mobile phones. Figure 6.5a shows the data for three mobile phones for a whole week. The telephones are identified by a phone index, and the day of the week by a day index. The phone with an index value of 2 and the day with an index value of 0 has the value 18. These two indices are necessary to access the number of text messages sent by a particular phone on a particular day. The tabular form in Figure 6.5a can be implemented by a two-dimensional array, requiring two indices to identify an element.

Multidimensional arrays can be implemented in Java by creating arrays of arrays. The declaration statement in Figure 6.5b declares an array reference (`weeklyData`) and creates a two-dimensional array that has three rows and seven columns.

The number of indices indicates the *dimension* of the array. The indices are interpreted from left to right, where the first index indicates the row and the second indicates the column. The number of indices on both sides of the assignment operator in the declaration statement must be the same. After the execution of the declaration statement, all the

twenty-one elements of the two-dimensional array have the default value for the `int` type (0).



We want to store the data from Figure 6.5a in the two-dimensional array `weeklyData`. After its creation as shown in Figure 6.5b, we can assign values to the elements individually:

```
// Initialization of 1st mobile phone
weeklyData[0][0] = 12; weeklyData[0][1] = 10; weeklyData[0][2] = 22;
weeklyData[0][3] = 33; weeklyData[0][4] = 19; weeklyData[0][5] = 27;
weeklyData[0][6] = 16;
// Initialization of 2nd mobile phone
weeklyData[1][0] = 45; weeklyData[1][1] = 55; weeklyData[1][2] = 44;
weeklyData[1][3] = 34; weeklyData[1][4] = 39; weeklyData[1][5] = 15;
weeklyData[1][6] = 11;
// Initialization of 3rd mobile phone
weeklyData[2][0] = 18; weeklyData[2][1] = 26; weeklyData[2][2] = 36;
weeklyData[2][3] = 40; weeklyData[2][4] = 24; weeklyData[2][5] = 11;
weeklyData[2][6] = 20;
```

It is important to note that the expressions `weeklyData[2][0]` and `weeklyData[0][2]` indicate different elements in the array. The expression `weeklyData[2][0]` indicates the third mobile phone on Monday, while the expression `weeklyData[0][2]` indicates the first mobile phone on Wednesday. It is important to understand what the indices represent, and be consistent in their usage.

Generalising what we have said about simple arrays, we can also declare, create and initialize the two-dimensional array `weeklyData` with the following declaration statement:

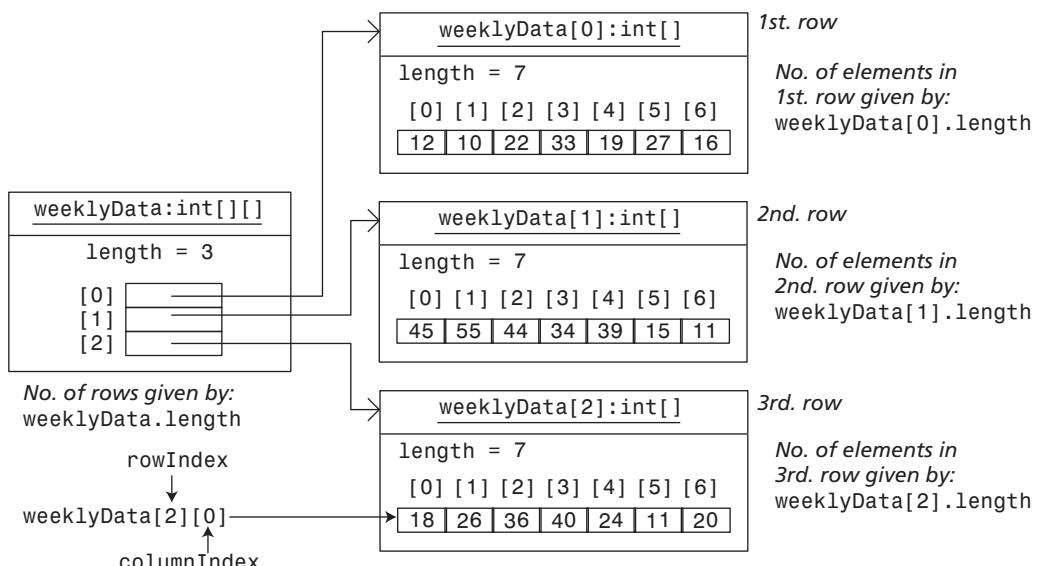
```
int[][] weeklyData = { // Declaration, creation and initialization.
    {12, 10, 22, 33, 19, 27, 16}, // 1st mobile phone
    {45, 55, 44, 34, 39, 15, 11}, // 2nd mobile phone
    {18, 26, 36, 40, 24, 11, 20} // 3rd mobile phone
};
```



In the declaration statement above, we see that the array reference declaration is the same as in Figure 6.5b. We have used the block notation to initialize the two-dimensional array. The outer block encloses a list of inner blocks, in which each inner block specifies the weekly data for one mobile phone, and the number of values in each block corresponds to the number of days in a week. The initialization makes it clear that each element in the outer block is a simple array.

Conceptually we can think of a two-dimensional array as the tabular form shown in Figure 6.5a. However, the actual data structure created is shown in Figure 6.6. The reference `weeklyData` actually refers to a simple array of length 3, in which each element corresponds to a row in the tabular form shown in Figure 6.5a. Each element is thus in turn a simple array that has a length of 7, in which each element holds an `int` value indicating the number of text messages sent. All multidimensional arrays are constructed from several simple arrays, as shown in Figure 6.6. Not surprisingly, such data structures are called *arrays of arrays*.

FIGURE 6.6 Array of arrays: explicit structure



Array of arrays, `weeklyData`, has type `int[][]`, i.e. an array of arrays of `int`.
Each row, `weeklyData[rowIndex]`, has type `int[]`, i.e. an array of `int`.

Each element in each row, `weeklyData[rowIndex][columnIndex]`, has type `int`.

The expression `weeklyData[2][0]` indicates a particular element in the structure shown in Figure 6.6, in which the first index represents a particular mobile phone and the second index represents a particular weekday. The expression `weeklyData[2]` therefore refers to a simple array that stores the data for the third mobile phone. The expression `weeklyData[2].length` indicates the number of elements in this simple array, namely 7.

The declaration:

```
int[][] weeklyData; // Declaration only
```

declares that the reference `weeklyData` can refer to any two-dimensional array of `int` values (i.e. has the type `int[][]`). Each row, `weeklyData[rowIndex]`, is a simple array of `int` values (i.e. has the type `int[]`), and each element, `weeklyData[rowIndex][columnIndex]`, is an integer (i.e. has the type `int`).

Program 6.4 uses the two-dimensional array `weeklyData` to accomplish some typical tasks that involve iterating over a multi-dimensional array.

Printing a two-dimensional array in tabular form

Problem (1) in Program 6.4 prints the array data in tabular form, using two nested `for(;;)` loops. The outer loop iterates over the phone indices, and the inner loop iterates over the simple array for each mobile phone that is identified by the phone index, printing the weekly data for each mobile phone in turn. The values are aligned in columns using the format specification "`%4d`".

Iterating over a specific row in a two-dimensional array

Problem (2) in Program 6.4 calculates the total number of text messages sent from the mobile phone indicated by index 1. This result is computed by iterating over the row `weeklyData[1]`, and adding each value given by the expression `weeklyData[1][dayIndex]`, $0 \leq dayIndex < weeklyData[1].length$.

Iterating over a specific column in a two-dimensional array

Problem (3) in Program 6.4 calculates the total number of text messages sent from all mobile phones on Wednesday (day index 2). This result is computed by adding the values in the column indicated by the day index 2. Each value in this column is given by the expression `weeklyData[phoneIndex][2]`, $0 \leq phoneIndex < weeklyData.length$.

Iterating over all the columns in a two-dimensional array

Problem (4) in Program 6.4 finds days on which the total number of text messages sent from all mobile phones is greater than 100. We have to add the values for all the telephones for each day (analogous to Problem (3)), and see if the sum is greater than 100.

We have used two nested `for(;;)` loops for this. The inner loop adds the values in each column (i.e. adds the values for all telephones for each day), and if the count is greater than 100, we print the day index. The outer `for(;;)` loop makes sure that all the columns (i.e. all the days of the week) are handled, using the expression `weeklyData[0].length` for the number of days in the week for all telephones. As all the rows have the same length (i.e. the same number of days in a week), we could use the length of any row in the loop condition of the outer loop.



PROGRAM 6.4 Multidimensional arrays

```
// Multi-dimensional Array Iteration using for(;;) loop
public class MultidimesionalArrayIteration {

    public static void main(String[] args) {

        int[][] weeklyData = { // Declaration, creation and initialisation.
            {12, 10, 22, 33, 19, 27, 16}, // 1st mobile phone
            {45, 55, 44, 34, 39, 15, 11}, // 2nd mobile phone
            {18, 26, 36, 40, 24, 11, 20} // 3rd mobile phone
        };

        // Problem (1) Print the data in tabular form
        for (int phoneIndex = 0;
             phoneIndex < weeklyData.length;
             phoneIndex++) {
            System.out.print("Phone index " + phoneIndex + ": ");
            for (int dayIndex = 0;
                 dayIndex < weeklyData[phoneIndex].length;
                 dayIndex++) {
                System.out.printf("%4d", weeklyData[phoneIndex][dayIndex]);
            }
            System.out.println();
        }

        // Problem (2) Find the total number of text messages sent from
        // the mobile phone indicated by index 1
        int sumWeek = 0;
        for (int dayIndex = 0; dayIndex < weeklyData[1].length; dayIndex++) {
            sumWeek += weeklyData[1][dayIndex];
        }
        System.out.println(
            "Total no. of text messages sent from the mobile phone given" +
            " by index 1: " + sumWeek);

        // Problem (3) Find the total number of text messages sent from all
        // mobile phones on Wednesday (day index 2)
        int sumMessages = 0;
        for (int phoneIndex = 0;
             phoneIndex < weeklyData.length;
             phoneIndex++) {
            sumMessages += weeklyData[phoneIndex][2];
        }
        System.out.println(
            "Total no. of text messages sent from all mobile phones on" +
            " Wednesday: " + sumMessages);

        // Problem (4) Find which days the total no. of text messages sent
```



```

// from all mobile phones is greater than 100.
for (int dayIndex = 0; dayIndex < weeklyData[0].length; dayIndex++) {
    int sumDays = 0;
    for (int phoneIndex = 0;
        phoneIndex < weeklyData.length;
        phoneIndex++) {
        sumDays += weeklyData[phoneIndex][dayIndex];
    }
    if (sumDays > 100) {
        System.out.println("The day with index " + dayIndex +
                           " has over 100 text messages registered.");
    }
}
}
}

```

Program output:

```

Phone index 0: 12 10 22 33 19 27 16
Phone index 1: 45 55 44 34 39 15 11
Phone index 2: 18 26 36 40 24 11 20
Total no. of text messages sent from the mobile phone given by index 1: 243
Total no. of text messages sent from all mobile phones on Wednesday: 102
The day with index 2 has over 100 text messages registered.
The day with index 3 has over 100 text messages registered.

```

Ragged arrays

Each row in a two-dimensional array is a simple array, but these inner simple arrays need not have the same length. The length of the rows can vary. Such arrays are called *ragged arrays*.

Suppose we want to create an overview of rainfall measurements at weather stations across a country. Each row of an array can represent a region and each column can represent the measurement of rainfall at a weather station in the region. It is not certain that each region has the same number of weather stations. The code below shows how an array of arrays can be constructed to model the rainfall statistics.

```

// Create a two-dimensional array with the required no. of rows for
// the regions with weather stations.
double[][] rainfallData = new double[3][]; // (1)
// (2) Create a simple array for each region with required no. of stations.
rainfallData[0] = new double[2]; // Two weather stations
rainfallData[1] = new double[1]; // One weather station
rainfallData[2] = new double[4]; // Four weather stations

```

Note that in (1) we specify the length of the first dimension from the left that represents the number of regions. Since each element in the two-dimensional array `rainfallData` is a simple array, each row can be created individually, as in (2). After execution of the code above, all the seven elements are initialized to the default value for the type `double`, 0.0. We can now assign other values to the elements.



As an alternative, we can also use the block notation in a declaration statement to create and initialize the elements of the two-dimensional array to specific values. This approach is illustrated by Program 6.5. In iterating over a ragged array, we make sure that the inner loop uses the correct length for each row. The loop at (2) explicitly refers to the row length by the expression `rainfallData[regionIndex].length`.

Note that we cannot change the number of rows in a multi-dimensional array, but we can replace the rows with other arrays. For example, if we want to replace the rainfall measurements for a region, we can create a new simple array and replace the old one with:

```
rainfallData[regionIndex] = new double[] {25.5, 12.75, 9.5};
```

PROGRAM 6.5 Ragged arrays

```
// Using ragged arrays
public class RaggedArrays {

    public static void main(String[] args) {

        // (1) Create and initialise the two-dimensional array
        // with rainfall data
        double[][] rainfallData = {
            {56.6, 30.2},           // Two weather stations
            {20.5},                 // One weather station
            {15.8, 7.0, 45.8, 0.6} // Four weather stations
        };

        // Print rainfall data.
        for (int regionIndex = 0;
             regionIndex < rainfallData.length;
             regionIndex++) {
            System.out.printf("Rainfall for region%2d: ", regionIndex);
            for (int stationIndex = 0;                                // (2)
                 stationIndex < rainfallData[regionIndex].length;
                 stationIndex++) {
                System.out.printf("%10.2f",
                                   rainfallData[regionIndex][stationIndex]);
            }
            System.out.println();
        }
    }
}
```

Program output:

```
Rainfall for region 0:      56.60      30.20
Rainfall for region 1:      20.50
Rainfall for region 2:      15.80      7.00      45.80      0.60
```



Arrays with more than two dimensions

We give an example of a three-dimensional array here, but arrays of any number of dimensions can be created in a similar fashion. However, arrays of dimension 4 and higher become more difficult to visualise.

Imagine we use containers to transport bicycles. These containers can be stacked in a rectangular grid in a ship's hold. The length and breadth of this grid measures 10 containers by 20 containers respectively. We can stack containers 15 high. The following three-dimensional array can be used to keep track of the number of bicycles in each container in the ship's hold:

```
int[][][] bicycles = int [10][20][15]; // [1][b][h]
```

The three indices identify a particular container stacked in the ship's hold. The reference `bicycles[1]` refers to a particular two-dimensional array (`int[][]`), and the reference `bicycles[1][b]` refers to a particular one-dimensional array (`int[]`). The value of `bicycles[1][b][h]` is the number of bicycles in a particular container. We could also construct this three-dimensional array piecemeal as we did with the construction of a ragged array. This approach is left to the reader as an exercise.

BEST PRACTICES

When iterating over a multi-dimensional array, use the length of *each* row explicitly, to avoid the `ArrayIndexOutOfBoundsException` in the case of a ragged array

E.g. `twoDimArray[i].length`.

6.6 More on iteration: enhanced for loop

We often want to iterate over a collection of elements, such as an array, modifying the elements. Earlier we used a `for(;;)` loop for this purpose, for example to read the values in the array `noOfTextMessages` for Problem (1) in Program 6.2:

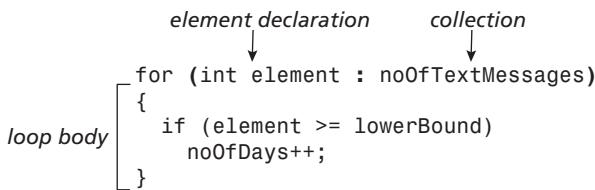
```
// Problem (1) Find how many days have a number of text messages  
// equal to or greater than a specified lower bound.  
...  
for (int index = 0; index < noOfTextMessages.length; index++) {  
    if (noOfTextMessages[index] >= lowerBound)  
        noOfDays++;  
}
```

6



The *enhanced for* loop is tailored to successively reading all the values in a *collection*. In each iteration of this loop the current element can be accessed. Figure 6.7 shows the `for(;;)` loop above rewritten using the enhanced *for* loop statement.

FIGURE 6.7 Enhanced `for` loop



In this book we will use the notation `for(:)` for the enhanced `for` loop, in contrast to the notation `for(;;)` for the counter-controlled `for` loop. The body of the `for(:)` loop is executed for *each* value in the collection (referenced by the variable in the element declaration). In Figure 6.7, the collection is an array.

The type of the element variable is the element type of the collection. In Figure 6.7 the array type is `int[]` and the element variable is declared as having the type `int`, which is the element type. The `for(:)` loop iterates over the specified collection, and for each iteration of the loop, the element variable is assigned a new value from the collection. The element variable is declared in the header of the `for(:)` loop, and can only be used in the loop body. In other words, the element variable is a local variable in the `for(:)` loop. However, changing its value in the loop body does *not* change any value in the collection. With the `for(:)` loop we also avoid out-of-bounds errors.

Here are cases where the `for(;;)` loop is preferable to the `for(:)` loop:

- Requiring the index to access particular element(s) or change element value(s).
- Iteration over more than one collection simultaneously.
- Iteration needs to be in increments other than one.
- The direction of the iteration is in reverse order.

Iterating over multidimensional arrays with the `for(:)` loop

Enhanced `for` loops can also be nested, for example to iterate over a multidimensional array. Problem (1) from Program 6.4 uses two nested `for(;;)` loops:

```
// Problem (1) Print the data in table form
for (int phoneIndex = 0;
     phoneIndex < weeklyData.length;
     phoneIndex++) {
    System.out.print("Phone index " + phoneIndex + ": ");
    for (int dayIndex = 0;
         dayIndex < weeklyData[phoneIndex].length;
         dayIndex++) {
        System.out.printf("%4d", weeklyData[phoneIndex][dayIndex]);
    }
    System.out.println();
}
```



We can simplify this code by using two nested `for(:)` loops:

```
// Problem (1) Print the data in table form
int rowIndex = 0;
for (int[] mobilephoneData : weeklyData) {
    System.out.print("Phone index " + (rowIndex++) + ": ");
    for (int noOfTextMessagesPerDay : mobilephoneData) {
        System.out.printf("%4d", noOfTextMessagesPerDay);
    }
    System.out.println();
}
```

This iterates over a two-dimensional array (`int[][]`) referenced by the array variable `weeklyData`. Each element in this two-dimensional array has the type `int[]`, i.e. it is a simple array of `int`. This type is specified for the element variable `mobilephoneData` in the outer `for(:)` loop. The inner `for(:)` loop iterates over a simple array of `int` values (`int[]`). In the inner `for(:)` loop, the element variable `noOfTextMessagesPerDay` indicates an `int` value in a simple array given by the reference `mobilephoneData` that is declared in the outer `for(:)` loop.

To keep track of which mobile phone we are dealing with, we use an extra `int` variable `rowIndex`, which is incremented explicitly as the outer `for(:)` loop iterates through the mobile phones.

BEST PRACTICES

Use the `for(:)` loop to iterate over an array if the element index is *not* required in the loop body, otherwise use the `for(;;)` loop.

6.7 More miscellaneous operations on arrays

6



Copying arrays

We have already noted that assigning one array reference to another array reference does not copy the elements from one array to the other. We must do the copying *element-wise*, i.e. we must copy the value of the element at a given index to the corresponding element in the other array at the same index. Problems (1) and (2) in Program 6.6 illustrate copying arrays.

Copying arrays of primitive values

Problem (1) in Program 6.6 illustrates copying primitive values from one array to another. The two arrays have the same length. We use a `for(;;)` loop to copy the `int` value in each element from one array to the corresponding element in the other array. In each iteration of the loop, the loop variable `i` indicates the corresponding elements in the two arrays. After the completion of the loop, corresponding elements in the two arrays have the same `int` value.

Copying arrays of objects

Problem (2) in Program 6.6 illustrates copying one array of objects to another. The operation is analogous to Problem (1). The elements of an array of objects are references, and hold reference values that identify objects. In each iteration of the loop, the reference value in an element of one array is assigned to the corresponding element in the other array. After the completion of the loop, corresponding elements in the two arrays are *aliases* to the same object. In other words, the two arrays share their objects. This kind of copying is called *reference value copying*.

Comparing arrays

If we want to find out whether two arrays are equal, i.e. have the same values in the corresponding elements, we cannot do that by comparing the array references using the == operator. As we know, comparison with the == operator only determines whether the two references are aliases. We have to do the comparison *element-wise*. Problem (3) and (4) in Program 6.6 illustrate comparison of arrays.

Comparing arrays of primitive values

Problem (3) in Program 6.6 uses a `for(;;)` loop to compare the values in the corresponding elements of the two int arrays. A guard variable (`equalValues`) terminates the loop as soon as the values in any two corresponding elements are not equal. Program output shows that the int arrays `intValuesI` and `intValuesII` are equal, which is not surprising, because we copied int values from the array `intValuesI` to the array `intValuesII` in Problem (1).

Comparing arrays of objects

Problem (4) in Program 6.6 illustrates comparison of arrays of objects. Problem (2) demonstrated reference value copying for arrays of objects. To compare such arrays, we compare the elements for *reference value* equality (or inequality) using the == (or !=) operator. This problem is essentially the same as Problem (3). Program output correctly shows that the String arrays `refValuesII` and `refValuesIII` are not equal, as the corresponding elements are not aliases.

6

PROGRAM 6.6 Copying and comparing arrays

```
// Misc. Array Operations
public class MiscArrayOperations {
    public static void main(String[] args) {
        // Problem (1) Copying an array of primitive values
        int[] intValuesI = {1, 3, 1949};           // Copy from this array
        int[] intValuesII = new int[intValuesI.length];// to this array.
        for (int i = 0; i < intValuesI.length; i++) {
            intValuesII[i] = intValuesI[i];
        }

        // Problem (2) Copying an array of objects (Reference value copying)
        String[] refValuesI = {"1.", "March", "1949"}; // Copy from this array
        String[] refValuesII = new String[refValuesI.length];// to this array.
        for (int i = 0; i < intValuesI.length; i++) {
```

```

    refValuesII[i] = refValuesI[i];
}

// Problem (3) Comparing arrays of primitive values
boolean equalValues = true;
for (int i = 0; equalValues && i < intValuesI.length; i++) {
    if (intValuesI[i] != intValuesII[i]) {
        equalValues = false;
    }
}
String notStr = "not ";
if (equalValues) {
    notStr = "";
}
System.out.println("Arrays intValuesI and intValuesII are " + notStr +
    "equal");

// Problem (4) Comparing arrays of objects for reference value equality
String[] refValuesIII = {"1949", "March", "1."};
boolean equalRefValues = true;
for (int i = 0; equalRefValues && i < refValuesIII.length; i++) {
    if (refValuesIII[i] != refValuesII[i]) {
        equalRefValues = false;
    }
}
notStr = "not ";
if (equalRefValues) {
    notStr = "";
}
System.out.println("Arrays refValuesIII and refValuesII are " +
    notStr + "equal");
}
}

```

Program output:

```

Arrays intValuesI and intValuesII are equal
Arrays refValuesIII and refValuesII are not equal

```

6



Working with partially-filled arrays

Creating an array requires information about the length of the array. However, it is not always known in advance how many items will be stored in the array. For example, when reading data from the terminal window or a file, we may have no clue beforehand how many items will be read. One solution is to overestimate the length of the array. However, if the array is not filled completely, i.e. it is partially-filled, we have to keep track of how many items are actually stored in the array. Processing of data in the array must ensure that only elements with valid data are processed. The field `length` in the array cannot be used for this purpose. One solution is to associate an integer variable with the array. This

variable acts as a counter and keeps track of how many valid items are in the array at any given time.

Program 6.7 illustrates how to process data in a partially-filled array. The program reads a sentence entered by the user. Each word in the sentence is stored as a `String` object in an array of strings. A word in our context is any sequence of characters delimited by whitespace. We stipulate that the maximum number of words in a sentence is fifty.

We use an index counter (`wordIndex`) to keep track of where the last word read from the terminal window was stored in the array. The loop condition at (1) ensures that the end-of-line string ("EOL") has not been read and there is still room in the array.

The loop at (2) prints the words in the sentence, but does so in reverse. The index `i` has the initial value `wordCount-1` and is decremented after each iteration until its value is less than 0. The associated counter `wordCount` helps to keep track of how many elements have valid data in the array.

PROGRAM 6.7 Working with partially-filled arrays

```
import java.util.Scanner;

public class PartiallyFilledArrays {
    public static void main(String[] args) {

        // Setup to read from the terminal window
        Scanner keyboard = new Scanner(System.in);

        // Create the array to hold maximum 50 words
        String[] sentence = new String[50];

        System.out.print("Enter a sentence (terminate with \"EOL\"): ");

        int wordIndex = -1;
        String word = keyboard.next(); // Read the first word.
        while (!word.equals("EOL") && wordIndex < sentence.length) { // (1)
            wordIndex++; // Index is incremented before storing
            sentence[wordIndex] = word;
            word = keyboard.next(); // Read the next word.
        }
        int wordCount = wordIndex + 1;
        System.out.println("No. of words: " + wordCount);

        // Print the words in reverse.
        for (int i = wordCount - 1; i >= 0; i--) { // (2)
            System.out.printf("%s ", sentence[i]);
        }
        System.out.println();
    }
}
```



Program output:

```
Enter a sentence (terminate with "EOL"): Don't worry, be happy. EOL
No. of words: 4
happy. be worry, Don't
```

6.8 Pseudo-random number generator

In computer games and simulations we often need to generate a sequence of random numbers. For a dice game it is necessary to simulate rolling the dice, i.e. generating a dice roll value between one and six. Ideally, the probability of a given value is the same (one sixth) for each value on a die. The numbers generated this way are called *random numbers*. To generate random numbers with the help of a program is not possible, as a program can only compute values, not pick them randomly. To guarantee equal probability for all cases is very difficult, so we have to settle for *pseudo-random numbers*, i.e. a sequence of numbers that closely approximates a sequence of random numbers. A lot of research has gone into finding mathematical formulae that compute good approximations to random numbers. We will look at what Java provides in this area.

The Random class

The `java.util.Random` class implements pseudo-random generators for many primitive data types (`boolean`, `byte`, `int`, `long`, `float`, `double`), but we will concentrate on a pseudo-random generator for `int` values.

First we need to create an object of the class `Random`:

```
Random generator = new Random();
```

We can then call the method `nextInt()` repeatedly on this object:

```
int number = generator.nextInt();
```

Each call to the method will return a random integer in the interval $[-2^{31}, 2^{31}-1]$, which is the range of the data type `int`.

6



Determining the range

We often are interested in generating random numbers in a particular range. For example, the following code will return a random number in the interval $[0, 10]$:

```
number = generator.nextInt(11); // Random integer in [0, 10]
```

If the parameter value is n , the integer returned is in the interval $[0, n-1]$. By supplying a new value to the `nextInt()` method, we can change the upper bound of the original interval.

If we want to shift the interval, we can add (or subtract) an *offset* from the value returned by the `nextInt()` method. In the code below, values generated in the original interval $[0,$

10] will now lie in the interval [2, 12], i.e. the offset 2 maps the interval [0, 10] onto the interval [2, 12]:

```
number = 2 + generator.nextInt(11); // Random integer in [2, 12]
```

If we want the values in the interval to have a *distance* greater than 1, we can multiply the value generated by a distance value:

```
number = 2 + 3*generator.nextInt(5); // Random integer in {2, 5, 8, 11, 14}
```

With a distance value of 3, the expression `3*generator.nextInt(5)` always returns a value in the set {0, 3, 6, 9, 12}, while an offset of 2 ensures that the variable `number` is assigned a value from the set {2, 5, 8, 11, 14}.

Simulating a dice roll

Program 6.8 illustrates how we can roll two dice:

```
int result = (1 + generator.nextInt(6)) + // 1st. die  
             (1 + generator.nextInt(6)); // 2nd. die
```

The program rolls the two dice a certain number of times, as read from the keyboard. It counts the number of times each value between 2 and 12 occurs as the sum of a two-dice roll. The frequencies are stored in an array, and are printed at the end. Note that we ignore the first two elements of the array `frequency`, as the values 0 and 1 cannot occur as a roll value for a roll of two dice.

PROGRAM 6.8 Frequency of dice roll values using two dice

```
import java.util.Random;  
import java.util.Scanner;  
// Simulates dice roll using a pseudo-random number generator, and  
// computes the frequency of two dice values [2,12] and their probability  
public class TwoDiceRollFrequency {  
    public static void main(String[] args) {  
  
        // Array for counting the frequency of two-dice values.  
        int[] frequency = new int[13]; // Ignore frequency[0] and frequency[1].  
        Scanner keyboard = new Scanner(System.in);  
        System.out.print("Enter the number of times to roll the two dice: ");  
        int noOfThrows = keyboard.nextInt();  
  
        Random generator = new Random(); // Pseudo-random number generator  
  
        for (int i = 1; i <= noOfThrows; i++) {  
            // Roll two dice.  
            int result = (1 + generator.nextInt(6)) + // 1st. die  
                         (1 + generator.nextInt(6)); // 2nd. die  
            // Increment the frequency of the value for two dice.  
            frequency[result]++;  
        }  
    }  
}
```



```

        for (int i = 2; i < frequency.length; i++) {
            System.out.printf("Sum of two dice values: %2d, " +
                "no. of times: %4d, probability: %.2f%n",
                i, frequency[i], ((double)frequency[i]/noOfThrows));
        }
    }
}

```

Program output:

```

Enter the number of times to roll the two dice: 4000
Sum of two dice values: 2, no. of times: 88, probability: 0.02
Sum of two dice values: 3, no. of times: 224, probability: 0.06
Sum of two dice values: 4, no. of times: 327, probability: 0.08
Sum of two dice values: 5, no. of times: 432, probability: 0.11
Sum of two dice values: 6, no. of times: 598, probability: 0.15
Sum of two dice values: 7, no. of times: 663, probability: 0.17
Sum of two dice values: 8, no. of times: 573, probability: 0.14
Sum of two dice values: 9, no. of times: 460, probability: 0.12
Sum of two dice values: 10, no. of times: 298, probability: 0.07
Sum of two dice values: 11, no. of times: 220, probability: 0.06
Sum of two dice values: 12, no. of times: 117, probability: 0.03

```

Generating the same sequence of pseudo-random numbers

The way in which we have used the pseudo-random number generator up to now cannot guarantee the same sequence of pseudo-random numbers each time the program is run. This is because the pseudo-random number generator is based on the time of the system clock. This is obviously different each time the program is run. If we want to generate the same sequence of pseudo-random numbers each time the program is run, we can specify a *seed* in the call to the `Random` constructor:

```
Random persistentGenerator = new Random(31);
```

In the declaration above, the seed is the prime number thirty-one. The seed is usually a prime number (i.e. a number that is only divisible by itself or one), as such numbers are highly suitable for implementing good pseudo-random number generators.



6.9 Review questions

1. Given that the reference `row` refers to a simple array, how would we find the number of elements in the array?

2. Name the three different ways in which the bracket notation `[]` is used.

3. Which lines of code create a simple array of integers referenced by the array called `age`?

- a `int[] age;`
- b `int[] age = int[100];`
- c `int age = new int[10];`
- d `int[] age = new int(10);`
- e `int[] age = new int[];`
- f `int[] age = new int[10];`

4. Which of the following statements about arrays are true?

- a All arrays have a method with the name `length()`.
- b All arrays have a public field with the name `length`.
- c Given that the reference `arrayRef` refers to an array, the first element in the array is accessed by `arrayRef[1]`.
- d When an array is created with `new type[n]`, all elements of the array are initialized with the default value of `type`.

5. Which code will create and initialize an array for CDs?

- a `CD[] cdCollection = { new CD(), new CD(), new CD() };`
- b `CD[] cdCollection;`
`cdCollection = { new CD(), new CD(), new CD() };`
- c `CD[] cdCollection = new CD[3];`
`cdCollection = { new CD(), new CD(), new CD() };`
- d `CD[] cdCollection = new CD[3];`
`cdCollection[0] = new CD();`
`cdCollection[1] = new CD();`
`cdCollection[2] = new CD();`
- e `CD[] cdCollection = new CD[10];`
`cdCollection = new CD[] { new CD(), new CD(), new CD() };`
- f `CD[] cdCollection = new CD[] { new CD(), new CD() };`

6. Which of the following statements about arrays are true?

- a Index out of bounds is checked at compile time.



- b** Index out of bounds is checked at runtime.
- c** The index value can be any expression that evaluates to an integer value.
- d** Index value must satisfy the relation $0 \leq \text{index-value} < \text{array-length}$.
- 7.** Given the following declaration, what is the type of `twoDimArrayName`, `twoDimArray-Name[1]` and `twoDimArrayName[1][2]`?
- ```
String[][] twoDimArrayName = String[5][4];
```
- 8.** Which code will create and initialize an *array of arrays* with CDs, where each row is a shelf of CDs?
- a** `CD[][] cdShelves = {  
 { new CD() },  
 { new CD(), new CD() },  
 { new CD(), new CD(), new CD() }  
};`
- b** `CD[][] cdShelves;  
cdShelves = {  
 { new CD() },  
 { new CD(), new CD() },  
 { new CD(), new CD(), new CD() }  
};`
- c** `CD[][] cdShelves = new CD[3][2];  
cdShelves = {  
 { new CD(), new CD() },  
 { new CD(), new CD() },  
 { new CD(), new CD() }  
};`
- d** `CD[][] cdShelves = new CD[3][2];  
cdShelves[0][0] = new CD(); cdShelves[0][1] = new CD();  
cdShelves[1][0] = new CD(); cdShelves[1][1] = new CD();  
cdShelves[2][0] = new CD(); cdShelves[2][1] = new CD();`
- e** `CD[][] cdShelves = new CD[][] {  
 { new CD() },  
 { new CD(), new CD() },  
 { new CD(), new CD(), new CD() }  
};`
- f** `CD[][] cdShelves = new CD[][] {  
 null, null, null  
};  
cdShelves[0] = new CD[] { new CD() };  
cdShelves[1] = new CD[] { new CD(), new CD() };  
cdShelves[2] = new CD[] { };`
- g** `CD[][] cdShelves = {  
 { new CD() },  
 { new CD(); new CD(); new CD() },  
 { new CD(); new CD(); new CD() }  
};`



```
{ new CD(); new CD() }
};
```

9. Which statements are true for a `for(:)` loop? Assume that the loop body does not execute statements that terminate the loop.
- a The element variable cannot be declared outside the loop.
  - b The collection can be empty.
  - c The loop body is executed at least once.
  - d Assignment to the element variable will change the values in the collection.

10. Which `for(:)` loops are valid? Assume that we have the following declaration:

```
String[] strArray = new String[5];
```

- a `String str;`  
`for (str : strArray) {`  
 `System.out.println(str);`  
`}`
- b `for (String str : strArray) {`  
 `System.out.println(str);`  
`}`
- c `for (String str : strArray) {`  
 `str = "^_^";`  
 `System.out.println(str);`  
`}`
- d `str i = 0;`  
`for (String str : strArray) {`  
 `System.out.println(i++ + ":" + str);`  
`}`

11. What is printed by the following program:

```
class ArrayPrint_PE1 {
 public static void main(String[] args) {
 int[] elements = new int[4];
 for (int i = 0; i < elements.length; i++)
 elements[i] = i + 1;
 for (int value : elements)
 System.out.printf("%2d", value);
 }
}
```

- a 1 2 4 6
- b 0 1 2 3
- c 1 3 6 10
- d 1 2 3 4



- 12.** What is printed by the following program:

```
class ArrayPrint_PE2 {
 public static void main(String[] args) {
 int[] elements = new int[4];
 int i = 0;
 for (int value : elements) {
 value = i + 1;
 i++;
 }
 for (int value : elements)
 System.out.printf("%2d", value);
 }
}
```

- a** 1 2 4 6
- b** 0 1 2 3
- c** 0 0 0 0
- d** 1 2 3 4

- 13.** In a street there are two hotels with three floors each. Each floor in a hotel has four rooms. Which declaration creates a multidimensional array exactly large enough to store information about the number of guests in each room on each floor in each hotel?

- a** int[][][] noOfGuests = new int[2][3][4];
- b** int[][][] noOfGuests = new int[4][3][2];
- c** int[][][] noOfGuests = new int[2][][];  
noOfGuests[0] = new int[3][4]; noOfGuests[1] = new int[3][4];
- d** int[][][] noOfGuests = new int[2][][];  
noOfGuests[0] = noOfGuests[1] = new int[3][4];

- 14.** What is printed by the following program:

```
class Length_PE3 {
 public static void main(String[] args) {
 String[] slogan = {"Java", "Jive"};
 int noOfChar = 0;
 for(int i = 0; i < slogan.length(); i++)
 noOfChar += slogan[i].length;
 System.out.println(noOfChar);
 }
}
```

- a** 8
- b** 0
- c** The program will not compile.



- 15.** Which statements are true about pseudo-random numbers? Assume the following declarations:

```
Random generator = new Random();
int number;
```

- a The variable `number` contains a value from the interval [0, 5], inclusive, after the assignment:

```
number = generator.nextInt(5);
```

- b The variable `number` contains a value from the interval [-5, 0], inclusive, after the assignment:

```
number = -5 + generator.nextInt(6);
```

- c The variable `number` contains a value from the interval [-5, 5], inclusive, after the assignment:

```
number = -5 + generator.nextInt(10);
```

- d The variable `number` contains a value from the set {0, 2, 4, 6, 8} after the assignment:

```
number = 2 * generator.nextInt(5);
```

- e The variable `number` contains a value from the set {3, 5, 7, 9, 11} after the assignment:

```
number = 3 + 2 * generator.nextInt(5);
```

## 6.10 Programming exercises

- 1.** Write a program that reads a sequence of integers from the terminal window and prints a frequency report showing how many times the integers 0, 1, ..., 9 occur in the sequence. Assume that the sequence is terminated by a negative integer. For example, if the user enters the following sequence:

```
8
0
2
8
8
9
5
9
-3
```

6



The program prints the following frequency report:

```
0 occurs 1 time
2 occurs 1 time
5 occurs 1 time
8 occurs 3 times
9 occurs 2 times
```



Tip: create an array of ten integers in which each element is a counter for its index value.

- 2.** Extend the program from Exercise 6.1 so that a histogram is printed in which the columns are horizontal. A column corresponds to the number of occurrences of each integer.

Example:

```
0: *
1: ***
2: *
3:
4: **
5: *
6: ****
7:
8: ***
9: **
```

- 3.** Modify the program from Exercise 6.2 so that the histogram is printed with vertical columns.

Example:

```
*
*
* *
* * *
* * * *
* * * * *
==0==1==2==3==4==5==6==7==8==9==
```

- 4.** Write a program that reads a list of ten integers from the terminal window and stores them in an array. The program should print the position of the smallest number in the array.

If the position of the smallest number is other than 0, the program should swap the number in position 0 with the smallest value. After swapping, the smallest number is now in position 0 and the number that was previously in position 0 is now where the smallest number was. Finally the program should print the values in the array.

- 5.** Modify Program 6.3 so that it uses the `for(:)` loop.
- 6.** Modify Program 6.4 so that the values for the two-dimensional array `weeklyData` are read from the terminal window.
- 7.** Modify Program 6.4 so that it calculates the total number of text messages sent each day, and the total number of text messages sent from each mobile phone:

| Day no.        | 0  | 1  | 2   | 3   | 4  | 5  | 6  | Sum(phone): |
|----------------|----|----|-----|-----|----|----|----|-------------|
| Phone index 0: | 12 | 10 | 22  | 33  | 19 | 27 | 16 | 139         |
| Phone index 1: | 45 | 55 | 44  | 34  | 39 | 15 | 11 | 243         |
| Phone index 2: | 18 | 26 | 36  | 40  | 24 | 11 | 20 | 175         |
| Sum(day):      | 75 | 91 | 102 | 107 | 82 | 53 | 47 |             |

- 6
- 
8. Write a program to play the game of *Craps*. The game is played by rolling two dice, where the sum of the rolled values of the two dice determines the course of the game. The program simulates the game by enforcing the following rules:
    - If the sum of the roll values is 7 or 11 at the first roll, the player has won. The sum of the roll values is called a *natural*.
    - If the sum of the roll values is 2 (*snakes eyes*), 3 (*cross eyes*) or 12 (*box cars*) at the first roll, the player has lost.
    - If the sum of the roll values is 4, 5, 6, 8, 9 or 10 at the first roll, the sum of the roll values establishes a *point* and the player can roll the dice again. The player continues to roll the dice until the sum of the roll values is either 7 or equal to the point. A sum of the roll values equal to 7 means the player has lost, while a sum of the roll values equal to the point means the player has won.
  9. Write a program to play the game of *Mastermind*. The aim of the game is to guess a secret code which is a finite sequence of digits, usually four. The program generates the code and gives feedback to the player after each guess based on the following rules:
    - Number of *bulls* in the guess, i.e. number of digits that are correctly identified in the code and are in the correct position.
    - Number of *cows* in the guess, i.e. number of digits that are correctly identified in the code, but are not in the correct position.

Example (the code is 9464):

```
Guess no. 1: 4444
No. of bulls: 2
No. of cows: 0
Guess no. 2: 4675
No. of bulls: 0
No. of cows: 2
Guess no. 3: 3494
No. of bulls: 2
No. of cows: 1
...
```

10. Write a program to grade a multiple-choice quiz. The correct answers for the quiz are:
  1. A      6. C
  2. D      7. B
  3. B      8. A
  4. C      9. D
  5. D      10. A

Pass marks are seven out of ten.

The program stores the correct answers in an array. The candidate's answers are read from the terminal window and are stored in another array. Let the character 'X' represent a question not answered by the candidate.

The program calculates and prints the following results:

- Whether the candidate passed the quiz or not.
- The total number of questions answered correctly by the candidate.
- The total number of questions answered incorrectly by the candidate.
- The total number of questions not answered by the candidate.
- A list of all the questions, showing what the candidate answered and what the correct answer was.

Example:

```
Enter in the answers: A D C C D C X B D A
The candidate PASSED
No. of questions answered correctly: 7
No. of questions answered incorrectly: 2
No. of questions not answered: 1
Question Candidate Ans. Correct Ans.
1. A A
2. D D
3. C B
...
...
```





## Defining classes

### LEARNING OBJECTIVES

By the end of this chapter you will understand the following:

- How to define your own classes that implement abstractions.
- How to declare, initialize and use field variables.
- How to declare and call methods.
- How to pass information to methods and how methods return values.
- How to use the `this` reference to refer to the current object.
- How static members of a class are declared and used.
- How information is passed to the program via program arguments.
- How to initialize the state of newly-created objects using constructors.
- How enumerated types can be used to define finite sets of symbolic constants.

### INTRODUCTION

Chapter 4 introduced the object model that is the foundation for object-oriented programming, in which programs are composed of objects that collaborate to provide the required functionality. An object belongs to a class that defines the common properties and behaviour of a particular type of objects. When writing programs, defining and understanding problem-specific classes is essential. In this chapter we focus on how to define and use our own classes, called user-defined classes, as opposed to the predefined classes found in the Java standard library. We also introduce a special kind of class that can be used to define finite sets of symbolic constants, called enumerated types.

## 7.1 Class members

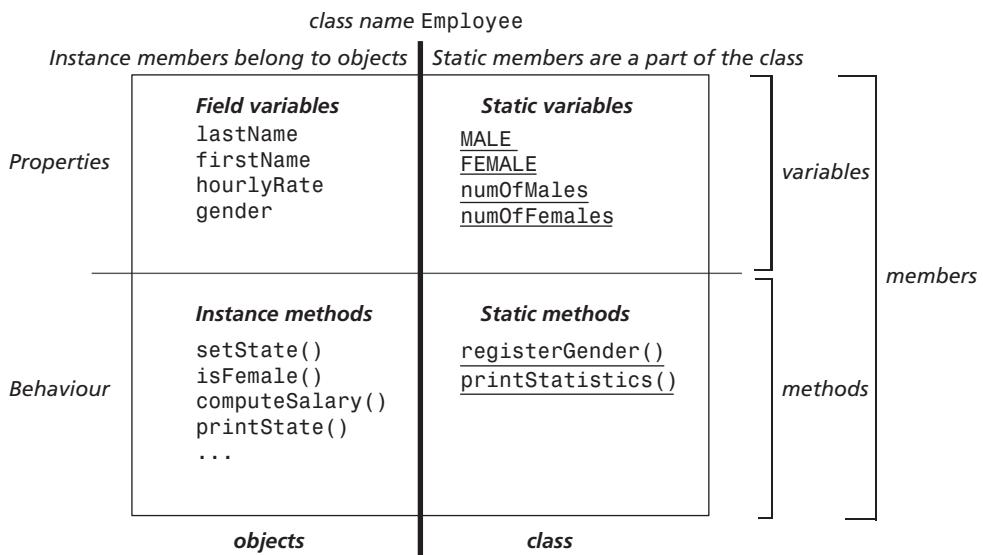
A *class declaration* defines the properties and the behaviour of the objects of the class. We will discuss the purpose of the different declarations that can be specified in a class declaration. We will also look at how these declarations are used inside the class and by other classes. These declarations are called *member declarations*.

Figure 7.1 gives an overview of the different *members* that can be declared in a class. *Field variables* represent properties and *instance methods* define the behaviour of the *objects* of the class. The term *instance* means an object of a class. Field variables and instance methods are collectively called *instance members*, and belong to the objects of the class.

In Java, a class can also define the properties and behaviour that belong to the class (see Figure 7.1). *Static variables* represent properties, and *static methods* define the behaviour of the class. Static variables and static methods are collectively called *static members*, and belong to the *class*, *not* to the objects of the class. Static members are discussed in Section 7.4.

Member declarations in a class can be declared in any order. It's common practice to group instance and static members separately, and further organise them according to fields and methods, as shown in Figure 7.1.

FIGURE 7.1 Overview of members in a class declaration



In addition to the members mentioned above, a class can also declare *constructors*. Constructors resemble methods, but the primary use of a constructor is to set the state of an object when the object is created. Section 7.5 discusses constructors.

We will use a class for employees in a company as a running example, and fill in the details of the class declaration in subsequent sections. You may find it useful to refer to



Figure 7.1 as we discuss the various members, to help identify to which group a member belongs.

## 7.2 Defining object properties using field variables

### Field declarations

As we have noted, field variables in a class declaration define the properties of objects that can be created from the class. Each object gets its own copy of the field variables. A *field declaration* specifies both the *field type* and the *field name*, just like in the declaration of local variables. The code below defines four field variables for the class `EmployeeV1`:

```
class EmployeeV1 { // Assume that no constructors are declared.
 // Field variables
 String firstName;
 String lastName;
 double hourlyRate;
 boolean gender; // false means male, true means female
 // ...
}
```

The field declarations above show that a field can be either a variable, such as the field name `hourlyRate`, that can store a value of a primitive type, for example the primitive type `double`, or it can be a reference variable, such as the field name `firstName`, that can store a reference value of an object, for example that of the `String` class. Objects that are created from the class will have a field for each of the field declarations in the class declaration.

### Initializing fields

We create objects from a class using the `new` operator, which requires a constructor call:

```
EmployeeV1 anEmployee = new EmployeeV1();
```

The `new` operator creates a new object of the `EmployeeV1` class in memory. This object will have room for all the fields declared in the class `EmployeeV1`. In the current version of the class, all objects created this way will have their fields initialized to *default values*, as shown in Figure 7.2a.

The *state* of an object comprises all values in the field variables of the object at any given time. The state can change over time, as the field values change. In the case of the `EmployeeV1` class, the *initial state* of an object is thus the default values of the field variables (Figure 7.2a).



**FIGURE 7.2** Field initialization (no constructors defined)

| anEmployee:EmployeeV1                                                     | employeeA:EmployeeV2                                                            |
|---------------------------------------------------------------------------|---------------------------------------------------------------------------------|
| lastName = null<br>firstName = null<br>hourlyRate = 0.0<br>gender = false | lastName = "Joe"<br>firstName = "Jones"<br>hourlyRate = 15.50<br>gender = false |

(a) Field initialization with default values

(b) Field initialization with initial values

It is possible to assign an initial value to the variable in a variable declaration. The same can be done for field variables. The class EmployeeV2 specifies an initial value for all its field variables:

```
class EmployeeV2 { // Assume that no constructors are declared.
 // Field variables with initial values
 String firstName = "Joe";
 String lastName = "Jones";
 double hourlyRate = 15.50;
 boolean gender = false; // false means male, true means female
 // ...
}
```

We can create an object of the class EmployeeV2 using the new operator:

```
EmployeeV2 employeeA = new EmployeeV2();
```

Execution of the expression new EmployeeV2() will always return an object with the initial state shown in Figure 7.2b, assuming that the class does not define any constructors that initialize the fields. Section 7.5 explains how the initial state can be customised by using constructors.

#### BEST PRACTICES

Always initialize the fields of a newly-created object in a constructor, so that the object gets a valid initial state.



## 7.3 Defining behaviour using instance methods

### Method declaration and formal parameters

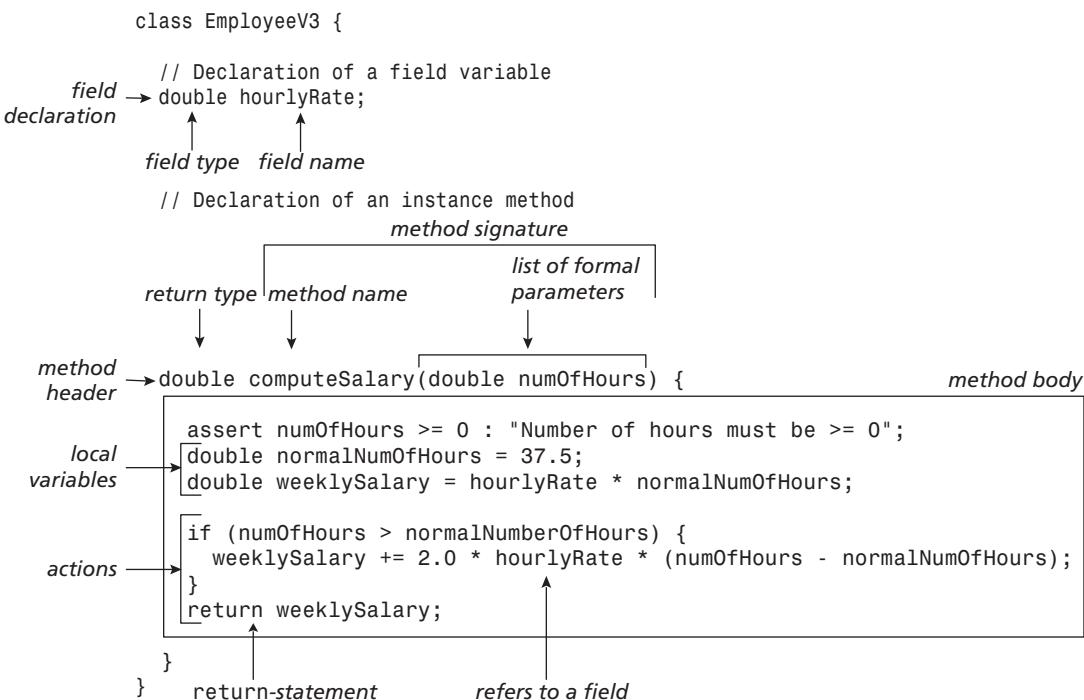
In Chapter 3 we saw that a method declaration comprises a method header and a method body. Figure 7.3 shows the declaration of the method computeSalary() in the class EmployeeV3. The method header declares the *return type*, the *method name* and the *parameter list*.

The return type of the method `computeSalary()` is the primitive type `double`, which means that the method returns a floating-point number when the method is called. If a method is not supposed to return a value, the keyword `void` should be specified instead of the return type. For example, see the method `setState()` in class `EmployeeV3` shown in Program 7.1. A non-void method must specify a return type.

The name of the method reflects what it does. The parameter list of the method indicates what information the method needs to do its job. The parameter list declares *formal parameters* for the method, in which each parameter is specified as a variable declaration with the parameter name and the parameter type. The method `computeSalary()` has a formal parameter `numOfHours` with the primitive type `double`. The type of a formal parameter can also be a reference type, for example a class or an array. The parameter list is always enclosed in parentheses, (), even if the method has no parameters. The method and the parameter list constitute the *signature* of the method. The signature determines which method declaration is chosen for execution by a method call. The method `computeSalary()` has the following signature:

```
computeSalary(double)
```

**FIGURE 7.3** Method declaration



The method body comprises variable declarations and actions. Variable declarations in the method body define the local variables needed to hold values during the execution of the method. Such variables are accessible only in the method body, and are not accessible outside the method in which they are declared. Formal parameters are also local variables. Several methods can have the same names for their local variables, but these are only

accessible within the method in which they are declared. The method `computeSalary()` has the following local variables: `normalNumOfHours`, `numOfHours`, `weeklySalary`.

The method body implements the actions. For example, the `if` statement in the body of the method `computeSalary()` is used to determine whether the employee that the object represents should receive any compensation for overtime. There are several statements, such as assignments, control flow statements and method calls, that can be used when implementing the behaviour of objects.

Local variables and statements can be defined in any order, but the rule is that a local variable must be declared before it can be used in the method body. It is a good idea to declare a local variable at the same time as when first assigning a value to it. This aids in making the purpose of the program clear.

### PROGRAM 7.1 Declaration of instance methods

```
class EmployeeV3 { // Assume that no constructors are declared.
 // Field variables
 String firstName;
 String lastName;
 double hourlyRate;
 boolean gender; // false means male, true means female

 // Instance methods

 // Assign values to the field variables of an employee.
 void setState(String fName, String lName,
 double hRate, boolean genderValue) {
 firstName = fName;
 lastName = lName;
 hourlyRate = hRate;
 gender = genderValue;
 }

 // Determines whether an employee is female.
 boolean isFemale() { return gender; }

 // Computes the salary of an employee, based on the number of hours
 // worked during the week.
 double computeSalary(double numOfHours) {
 assert numOfHours >= 0 : "Number of hours must be >= 0";
 double normalNumOfHours = 37.5;
 double weeklySalary = hourlyRate * normalNumOfHours;
 if (numOfHours > normalNumOfHours) {
 weeklySalary += 2.0 * hourlyRate * (numOfHours - normalNumOfHours);
 }
 return weeklySalary;
 }

 // Prints the values in the field variables of an employee.
```



```

void printState() {
 System.out.print("First name: " + firstName);
 System.out.print("\tLast name: " + lastName);
 System.out.printf("\tHourly rate: %.2f", hourlyRate);
 if (isFemale()) {
 System.out.println("\tGender: Female");
 } else {
 System.out.println("\tGender: Male");
 }
}
}

```

---

## Method calls and actual parameter expressions

A method call is used to execute a method body. Figure 7.4 shows an example. A call specifies the object whose method is called (given by the reference `manager` in this case), the name of the method and any information the method needs to execute its actions. These are referred to as *actual parameters*, or *arguments*. An actual parameter is an *expression*, in contrast to a formal parameter specified in the method declaration, which is always a *variable*. The *signature of a method call* consists of the method name and the type of the actual parameter expressions. In Figure 7.4, for example, the method call has the following signature:

```
setState(String, String, double, boolean) // (1) Call signature
```

The compiler checks that a method exists that corresponds to the method call. It requires the signature of the method call to be compatible with the signature of the method declaration. This compatibility implies that the value of the first actual parameter can be assigned to the first formal parameter, the value of the second actual parameter can be assigned to the second formal parameter and so on. In Figure 7.4 the declaration of the method `setState()` has the following signature:

```
setState(String, String, double, boolean) // (2) Method signature
```

We can therefore see that the signature of the call to the method `setState()` at (1) corresponds to the signature of the method declaration at (2). For all formal parameters, the value of the actual parameter type is actually assigned to the corresponding formal parameter variable, as shown in Figure 7.4 on page 168.

Finally, here are some examples of method calls that result in compile time errors. The following method calls to the method `setState()`:

```
manager.setState(name, hourlyRate*2.0, "Jones", false); // (3) Method call
manager.setState(name, "Jones", hourlyRate*2.0); // (4) Method call
```

have the following signatures respectively:

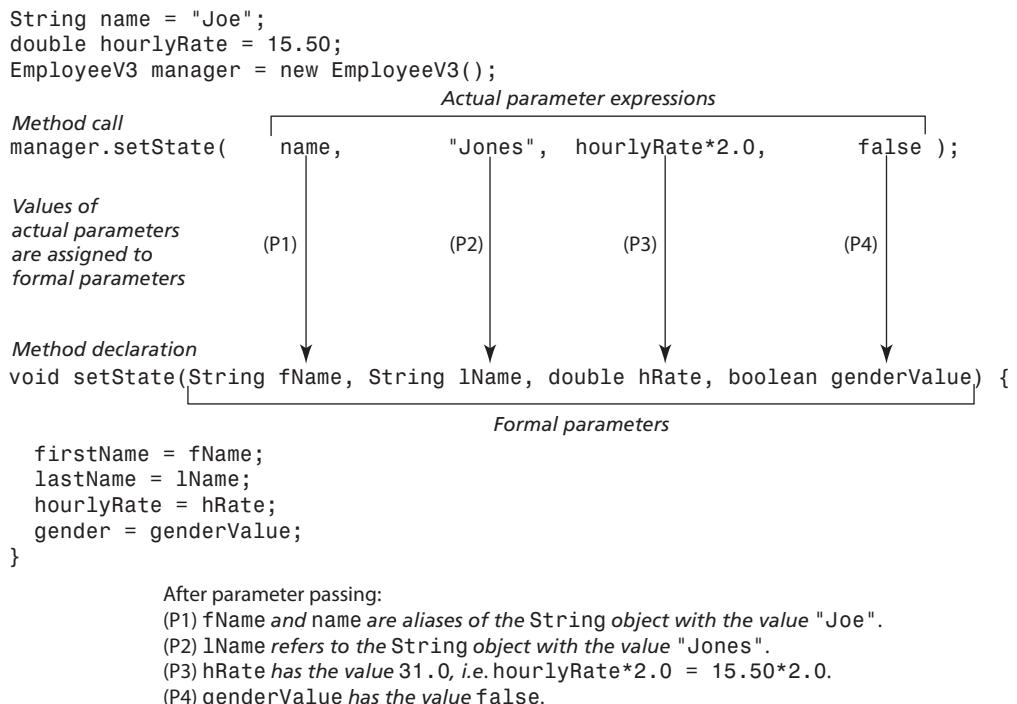
```
setState(String, double, String, boolean) // (5) Call signature
setState(String, String, double) // (6) Call signature
```

We can see that the call signatures at (5) and (6) do not correspond to the method signature in (2) above. The call signature at (5) results in a compile time error, because of the



second parameter in the call, as a value of type `double` cannot be assigned to a `String` reference in the method signature. In the call signature at (6), the number of actual parameters is not correct – it should be four.

**FIGURE 7.4 Parameter passing**



## Parameter passing: call-by-value

An actual parameter expression is evaluated and its value is assigned to the corresponding formal parameter variable. All actual expressions are evaluated before the call to the method is executed. In the call to the `setState()` method in Figure 7.4, all actual parameters are simple expressions that evaluate to the following values: a reference value of a `String` object with the value "Joe", a reference value of a `String` object with the value "Jones", the floating-point value 31.0 (`hourlyRate*2.0 = 15.5*2.0`), and the Boolean value `false`.

Before the method is executed, values of the actual parameter expressions are assigned to the corresponding formal parameter variables. Parameter passing in Figure 7.4 is equivalent to the following assignments:

```
String fName = name; // (P1) reference value of "Joe"
String lName = "Jones"; // (P2) reference value of "Jones"
double hRate = 31.0; // (P3) primitive value 31.0
boolean genderValue = false; // (P4) primitive value false
```

At (P1) and (P2) in Figure 7.4, the actual parameter values are *reference values*, so these reference values are passed before execution of the method `setState()` starts. The assign-



ment at (P1) implies that references `fname` and `name` are aliases to the same `String` object with the value "Joe" at the start of execution of the method `setState()`. Analogously, the reference `lName` at (P2) refers to the `String` object with the value "Jones" at the start of the execution in the method `setState()`. Note that no objects are copied, only reference values. (P3) and (P4) show the passing of primitive values, `double` and `boolean` respectively. This way of passing parameters in which only values are passed is called *call-by-value*.

Program 7.2 illustrates the execution of the method call in Figure 7.4. The program prints the state of the `EmployeeV3` object referred to by the `manager` reference before and after the call to the `setState()` method. The program output confirms that the state of the `EmployeeV3` object created in (1) was updated with the values of the actual parameter expressions in the method call at (2).

The remainder of Program 7.2 is explained in *Consequences of call-by-value* on page 170.

## PROGRAM 7.2 Parameter passing: call-by-value

```
// Illustrating parameter passing
public class Client3A {
 public static void main(String[] args) {

 String name = "Joe";
 double hourlyRate = 15.50;
 EmployeeV3 manager = new EmployeeV3(); // (1)
 System.out.println("Manager state before call to setState() method");
 manager.printState();
 manager.setState(name, "Jones", hourlyRate*2.0, false); // (2)
 System.out.println("Manager state after call to setState() method");
 manager.printState();
 System.out.println();

 System.out.printf("Manager hourly rate before adjusting: %.2f%n",
 manager.hourlyRate); // (3)
 adjustHourlyRate(manager.hourlyRate); // (4) LOGICAL ERROR!
 System.out.printf("Manager hourly rate after adjusting: %.2f%n",
 manager.hourlyRate);
 System.out.println();

 EmployeeV3 director = new EmployeeV3(); // (5)
 System.out.println("Director state before call to copyState() method");
 director.printState();
 copyState(manager, director); // (6)
 assert (manager.lastName.equals(director.lastName) &&
 manager.firstName.equals(director.firstName) &&
 manager.hourlyRate == director.hourlyRate &&
 manager.gender == director.gender) :
 "Manager and director have different states after copying.";
 System.out.println("Director state after call to copyState() method");
 director.printState();
 }
}
```



```

 System.out.println("Manager state after call to copyState() method:");
 manager.printState();
 System.out.println();
 }

 // Method that tries to adjust the hourly rate.
 static void adjustHourlyRate(double hourlyRate) { // (7)
 hourlyRate = 1.5 * hourlyRate; // (8)
 System.out.printf("Adjusted hourlyRate: %.2f%n", hourlyRate);
 }

 // Method that copies the state of one employee over to another employee.
 static void copyState(EmployeeV3 fromEmployee,
 EmployeeV3 toEmployee) { // (9)

 toEmployee.setState(fromEmployee.firstName, // (10)
 fromEmployee.lastName,
 fromEmployee.hourlyRate,
 fromEmployee.gender);

 toEmployee = fromEmployee = null; // (11)
 }
}

```

Program output:

```

Manager state before call to setState() method
First name: null Last name: null Hourly rate: 0.00 Gender: Male
Manager state after call to setState() method
First name: Joe Last name: Jones Hourly rate: 31.00 Gender: Male

Manager hourly rate before adjusting: 31.00
Adjusted hourlyRate: 46.50
Manager hourly rate after adjusting: 31.00

Director state before call to copyState() method
First name: null Last name: null Hourly rate: 0.00 Gender: Male
Director state after call to copyState() method:
First name: Joe Last name: Jones Hourly rate: 31.00 Gender: Male
Manager state after call to copyState() method:
First name: Joe Last name: Jones Hourly rate: 31.00 Gender: Male

```



### ***Consequences of call-by-value***

Inside the method body a formal parameter is used like any other local variable in the method, so that changing its value in the method has no effect on the value of the corresponding actual parameter in the method call.

The class `Client3A` defines a static method `adjustHourlyRate()` at (7) in Program 7.2. This method changes the value of the formal parameter `hourlyRate`:

```
hourlyRate = 1.5 * hourlyRate; // (8)
```

The method `adjustHourlyRate()` is called at (4) in the hope of adjusting the hourly rate of an employee. Output from the program shows that changing the value of the formal parameter `hourlyRate` in the method has no effect on the value of the variable `manager.hourlyRate`, which is the corresponding actual parameter. A simple solution for getting the adjusted value from the method `adjustHourlyRate()` is to modify the method so that it returns the adjusted value:

```
static double adjustHourlyRate(double hourlyRate) { // (7)
 hourlyRate = 1.5 * hourlyRate; // (8)
 System.out.printf("Adjusted hourly rate: %.2f%n", hourlyRate);
 return hourlyRate;
}
```

The value returned by the method call can then be assigned explicitly:

```
manager.hourlyRate = adjustHourlyRate(manager.hourlyRate);
```

### **Reference values as actual parameter values**

The state of an object whose reference value is passed to a formal parameter variable can be changed in the method, and the changes will be apparent after return from the method call. Such a state change is illustrated by the method `copyState()` in Program 7.2.

The class `Client3A` defines the static method `copyState()` at (9), which copies the state of one employee to another employee. The method is called at (6):

```
copyState(manager, director); // (6)
```

After the call we see from the output that the state of the `Employee3V` object referred to by the reference `director` has the same state as the `Employee3V` object referred to by the reference `manager`. Only the reference values of the two objects are passed in the method call. The state of the employee object referred to by the reference `director` is changed via the formal parameter reference `toEmployee`. These changes are apparent after return from the method call, as confirmed by the program output. At (11) the values of both the formal variables are set to `null`, but this has no effect on the variables that make up the actual parameter expressions. These variables, `manager` and `director`, still refer to their respective objects after return from the method.

### **Arrays as actual parameter values**

Passing arrays as parameter values does not differ from what we have seen earlier for other objects that occur in actual parameter expressions. If the actual parameter evaluates to a reference value of an array, for example, then this reference value is passed. Program 7.3 illustrates the use of arrays as actual parameters.

Four arrays are created with information about three employees at (1) in Program 7.3. The same index in all the four arrays gives information about the same employee. This information is used to create an array of three employees in (2), (3) and (4). In addition,



an array contains the number of hours each employee has worked in a week, at (5). Finally, the salaries of all the employees in the array `employeeArray` are computed by a call to the static method `computeSalaries()`.

Because we use the `[]` notation to declare an array, we use the same notation to declare an array reference as a formal parameter. An example of such a declaration is shown at (7) in Program 7.3:

```
static void computeSalaries(EmployeeV3[] employeeArray,
 double[] hoursArray) {...}
```

The references `employeeArray` and `hoursArray` refer to arrays of type `EmployeeV3[]` and `double[]`, respectively. Without the `[]` notation, these references would not be array references. (6) shows a call to this method:

```
computeSalaries(employeeArray, hoursArray);
```

Again parameter passing is equivalent to the following assignments in which the reference values of the arrays are passed:

```
EmployeeV3[] employees = employeeArray; // Reference value of array
double[] hours = hoursArray; // Reference value of array
```

We don't use the `[]` notation for array variables when these are passed as actual parameters in a method call. Instead, the reference value of the array is passed just like the reference value of any other object, as shown in the assignments above. Note that the signature of the method is:

```
computeSalaries(EmployeeV3[], double[]) // Method signature
```

and the signature of the method call corresponds to the method signature. Note also that there is no requirement on the length of the array in the method call, which is implicitly given by the `length` field of the array. However, the call must have the right *array type*.

If an *array element* occurs in an actual parameter expression, the value of the element is used in the evaluation of the expression and the expression value is passed, as one would expect. If the actual parameter is an array element of a primitive type, for example `hourlyRateArray[2]`, whose type is `double`, the primitive value is passed. If the actual parameter is an array element of a reference type, for example `lastNameArray[1]`, whose type is `String`, the reference value of the element object is passed. (4) in Program 7.3 shows a call to the method `setState()`, in which the actual parameter expressions are array elements. Note that the types of the actual parameters at (4) are in agreement with the type list (`String, String, double, boolean`) in the signature of the `setState()` method.

7



### PROGRAM 7.3 Arrays as actual parameters

```
// Passing arrays
public class Client3B {

 public static void main(String[] args) {
 // (1) Associated arrays with information about employees:
 String[] firstNameArray = { "Tom", "Dick", "Linda" };
```

```

String[] lastNameArray = { "Tanner", "Dickens", "Larsen" };
double[] hourlyRateArray = { 30.00, 25.50, 15.00 };
boolean[] genderArray = { false, false, true };

// (2) Array with employees:
EmployeeV3[] employeeArray = new EmployeeV3[3];

// (3) Create all employees:
for (int i = 0; i < employeeArray.length; i++) {
 employeeArray[i] = new EmployeeV3();
 employeeArray[i].setState(firstNameArray[i], lastNameArray[i],
 hourlyRateArray[i], genderArray[i]); // (4)
}

// (5) Array with hours worked by each employee:
double[] hoursArray = { 50.5, 32.8, 66.0 };

// (6) Compute the salary for all employees:
computeSalaries(employeeArray, hoursArray);
}

// (7) Compute the salary for all employees:
static void computeSalaries(EmployeeV3[] employees,
 double[] hours) {
 for (int i = 0; i < employees.length; i++) {
 System.out.printf("Salary for %s %s: %.2f\n",
 employees[i].firstName,
 employees[i].lastName,
 employees[i].computeSalary(hours[i]));
 }
}
}

```

Program output:

```

Salary for Tom Tanner: 1905.00
Salary for Dick Dickens: 956.25
Salary for Linda Larsen: 1417.50

```

## The current object: **this**

7



When we call an instance method of an object, we say that the method is *invoked* on the object. The object whose method is invoked becomes the *current object*. Inside the method, the current object can be referred to by the keyword `this`, i.e. the keyword `this` is a reference. We can think of the `this` reference as an implicit actual parameter that is passed to an instance method each time it is invoked. For example, in the method call:

```
manager.setState(name, "Jones", hourlyRate*2.0, false); // (2)
```

the reference value held in the reference `manager` will be available in the `this` reference inside the method `setState()` when the method is executed. So the method `setState()` in Program 7.1 could be declared as follows using the `this` reference:

```
void setState(String fName, String lName,
 double hRate, boolean genderValue) {
 this.firstName = fName;
 this.lastName = lName;
 this.hourlyRate = hRate;
 this.gender = genderValue;
}
```

The reference `this` can be used exactly like any other reference in the method, except that its value cannot be changed, i.e. it is a `final` reference. For example, we can refer to the field `firstName` in the current object by using the expression `this.firstName`, as shown above in the declaration of the method.

If a local variable has the same name as a field variable, the local variable will *shadow* the field variable, as shown in this declaration of the same method:

```
void setState(String firstName, String lastName,
 double hRate, boolean genderValue) {
 firstName = firstName; // (1)
 lastName = lastName; // (2)
 hourlyRate = hRate;
 gender = genderValue;
}
```

The first two formal parameters have the same names as the field variables `firstName` and `lastName`. Formal parameters are local variables, and inside the method these two names refer to the formal parameters. At (1) and (2) above, values of the formal parameters are assigned back to the formal parameters. We can use the `this` reference to distinguish the field variable from the local variable when they have the same name:

```
void setState(String firstName, String lastName,
 double hRate, boolean genderValue) {
 this.firstName = firstName; // (1')
 this.lastName = lastName; // (2')
 this.hourlyRate = hRate;
 this.gender = genderValue;
}
```

7

## Method execution and the `return` statement

When we invoke a method on an object, the method can in turn invoke other methods, either on the current object or on other objects. Figure 7.5 illustrates invocation of the method `setState()` in which we have introduced two local variables (`i` and `s`) and two calls to the method `printState()`:

```
void setState(String firstName, String lastName,
 double hRate, boolean genderValue) {
```

```

// Some superfluous local variables
int i;
String s;

this.printState(); // Print state before
this.firstName = firstName;
this.lastName = lastName;
this.hourlyRate = hRate;
this.gender = genderValue;
this.printState(); // Print state after
}

```

The call to the method `setState()` is not complete when the executions of the two calls to the method `printState()` has completed. The current object is an object of class `EmployeeV3`, and its state will be printed twice. Note that execution continues in the method `setState()` after return from the calls to the method `printState()`. The execution of the calls to the method `printState()` is embedded in the call to the method `setState()`.

A local variable is not initialized to a default value during method execution. If we try to use a local variable before it has been assigned a value, the compiler will report an error.

### BEST PRACTICES

Always initialize local variables when they are declared.

We can see from Figure 7.5 on page 176 that control returns from the method `setState()` only after the last statement in the method has been executed. This need not always be the case: the `return` statement can be used to end execution of a method wherever appropriate.

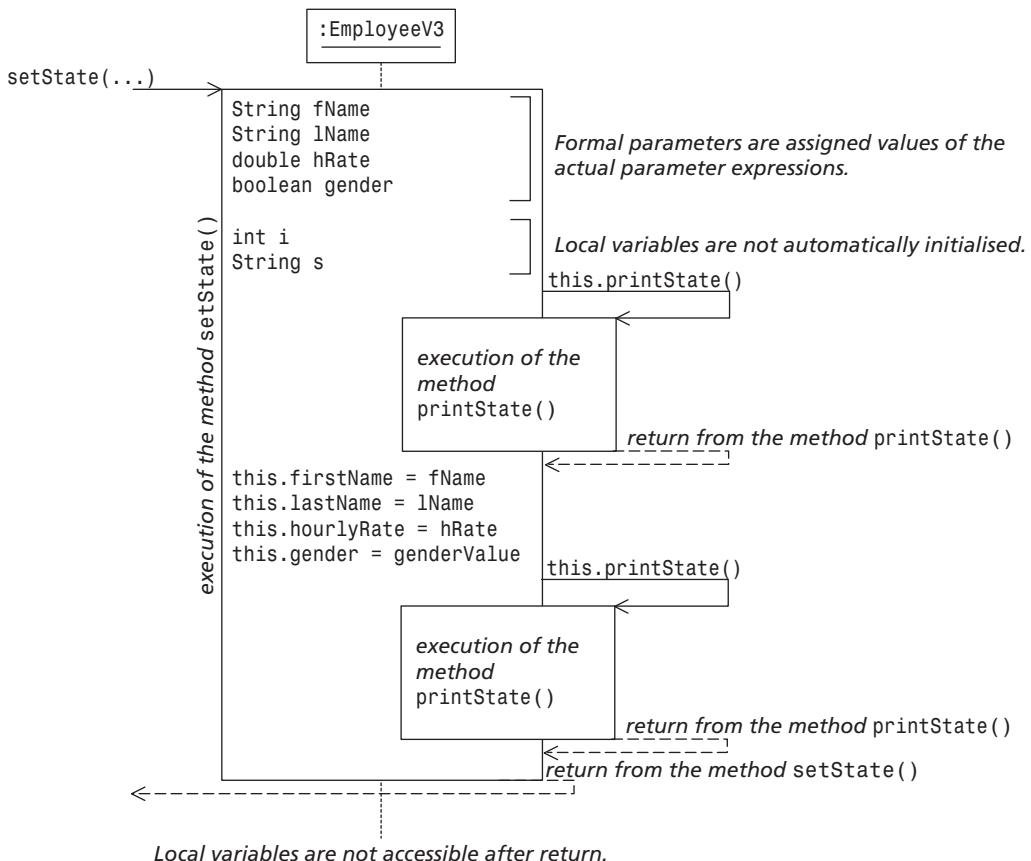
The `return` statement comes in two forms. The first form of the statement consists of the keyword `return` only:

```
return; // Method execution stops, and control returns.
```

This form can be used when the method does not return a value, for example in the method `setState()`. We could have included a `return` statement as the last statement, but that would be redundant, as the execution of the method will end anyway when there are no more statements left to execute in the method body.



**FIGURE 7.5** Method execution



The second form of the `return` statement requires an expression in addition to the keyword `return`. The value of this expression is returned after the execution of the method stops at the `return` statement. The method must explicitly specify the *return type* in the method header, and the *return value* must be of this return type. The `return` statement is required for methods that return a value, and the compiler will insist upon it. The method `computeSalary()` below illustrates the use of the `return` statement:

```

double computeSalary(double numOfHours) {
 assert numOfHours >= 0 : "Number of hours must be >= 0";
 double normalNumberOfHours = 37.5;
 double weeklySalary = hourlyRate * normalNumberOfHours;
 if (numOfHours <= normalNumberOfHours) {
 return weeklySalary; // (1)
 }
 return weeklySalary +
 2.0 * hourlyRate * (numOfHours - normalNumberOfHours); // (2)
}

```

The execution of the method stops when either (1) or (2) is executed. The method will return the value of the expression in the `return` statement that was executed. In both (1)



and (2) the expression evaluates to a value of type `double`, which is the return type specified in the method header. A method can return only one value, either a value of a primitive type or a reference value of an object.

The method `computeSalary()` above has two exits, corresponding to the two return statements. Execution can be difficult to understand if a method has too many exits. We can rewrite the method `computeSalary()` so that it has only one exit and in which the variable `weeklySalary` always holds the correct salary, allowing for overtime:

```
double computeSalary(double num0fHours) {
 assert num0fHours >= 0 : "Number of hours must be >= 0";
 double normalNumberOfHours = 37.5;
 double weeklySalary = hourlyRate * normalNumberOfHours;
 if (num0fHours > normalNumberOfHours) {
 weeklySalary += 2.0 * hourlyRate * (num0fHours - normalNumberOfHours);
 }
 return weeklySalary;
}
```

### BEST PRACTICES

Keeping the number of exits from a method to a minimum aids in understanding the program logic.

## Passing information using arrays

Program 7.4 shows two ways of passing information between methods using arrays.

At (1a) the method `main()` calls the method `fillStringArray()` declared at (3a), passing an empty array of strings. The method `fillStringArray()` reads as many strings as the length of the array, and fills the array. After control returns from the method `fillStringArray()`, the actual parameter `firstNameArray` still refers to the same array whose reference value was passed, but which is now filled with string values. The method call and the method declaration are as follows:

```
fillStringArray(firstNameArray); // (1a) Call
...
static void fillStringArray(String[] strArray) { ... } // (3a) Declaration
```

At (2a) the method `main()` calls the method `createStringArray()` declared at (4a). The method `createStringArray()` first asks the user for the number of values to read. It then creates an array of strings of this size, and subsequently fills the array. The method `createStringArray()` returns the reference value of the array, which on return is assigned to the local reference variable `lastNameArray`. The array object created in the method `createStringArray()` continues to exist even after the method has finished executing, because the reference value of the array was stored in a local reference variable on return. The method call and the method declaration in this case are as follows:

```
String[] lastNameArray = createStringArray(); // (2a) variable type String[]
```

```
...
 static String[] createStringArray() { ... } // (4a) return type String[]
```

Which approach is best depends on how much work the calling method wants the called method to do, i.e. more than just read the values and fill the array.

### PROGRAM 7.4 Handling arrays

```
import java.util.Scanner;
public class ArrayMaker {

 public static void main(String[] args) {
 // (1) Read first names:
 String[] firstNameArray = new String[3];
 System.out.println("Read first names");
 fillStringArray(firstNameArray); // (1a)
 printStrArray(firstNameArray);

 // (2) Read last names:
 System.out.println("Read last names");
 String[] lastNameArray = createStringArray(); // (2a)
 printStrArray(lastNameArray);
 }

 // (3) Reference value of the array to be filled is passed
 // as formal parameter:
 static void fillStringArray(String[] strArray) { // (3a)
 Scanner keyboard = new Scanner(System.in);
 for (int i = 0; i < strArray.length; i++) {
 System.out.print("Next: ");
 strArray[i] = keyboard.nextLine();
 }
 }

 // (4) The method creates and fills an array of strings.
 // The reference value of this array is returned by the method:
 static String[] createStringArray() { // (4a)
 Scanner keyboard = new Scanner(System.in);
 System.out.print("Enter the number of items to read: ");
 int size = keyboard.nextInt();
 String[] strArray = new String[size];
 keyboard.nextLine(); // Clear any input first
 for (int i = 0; i < strArray.length; i++) {
 System.out.print("Next: ");
 strArray[i] = keyboard.nextLine();
 }
 return strArray;
 }

 // (5) Prints the strings in an array to the terminal window:
```



```
static void printStrArray(String[] strArray) {
 for (String str : strArray) {
 System.out.println(str);
 }
}
```

Program output:

```
Read first names
Next: Tom
Next: Dick
Next: Linda
Tom
Dick
Linda
Read last names
Enter the number of items to read: 3
Next: Tanner
Next: Dickens
Next: Larsen
Tanner
Dickens
Larsen
```

---

## Automatic garbage collection

Objects that are no longer in use are taken care of by the JVM without the program having to do anything special. These objects are deleted from memory, and the memory freed can be used for new objects. It is always the JVM that decides *when* such a clean up should take place, for example when it starts to run out of free memory. This clean-up process is called *automatic garbage collection*.

Note that if a method creates an object and return its reference value, the reference value of the object can be used after the method returns. Such an object would therefore not be a candidate for garbage collection. If the reference value of an object is only stored in a local variable, the object will not be accessible after return from the method. Such an object can be garbage collected by the JVM.

## 7.4 Static members of a class

7



Static members specify properties and behaviour of the *class*, and are therefore not a part of any object created from the class.

The following simple example illustrates the use of static members. How can we keep track of the number of male and female employee objects that have been created? We can define two counters that are incremented, depending on whether a male or a female employee is created. If these counters are declared as instance variables in the class decla-

ration, they will exist in every employee object we create from the class. Each object will have two counters, so which counters should we use to do the book keeping? One solution is to maintain the counters as static variables for the class only, and update the appropriate counter each time an employee object is created.

We use the keyword `static` in the declaration of static members to distinguish them from instance members. Program 7.5 shows declarations of static variables and methods at (1) and (2) respectively.

The class `EmployeeV4` in Program 7.5 declares the following two static variables (see under (1)):

```
static int numOfFemales;
static int numOfMales;
```

Static variables are *global variables* in the sense that there is only one occurrence of such variables and that is in the class, in contrast to instance variables that exist in each object created from the class.

Static variables are also useful for defining *global constants*. Once a value is assigned to such a variable its value cannot be changed. Such constants can be used inside the class and by other classes. If we decide that the Boolean value `true` indicates a female employee, then the Boolean value `false` indicates a male employee. Instead of using the Boolean values directly for this purpose, we can define and use constants with meaningful names (see (1) in Program 7.5):

```
static final boolean MALE = false;
static final boolean FEMALE = true;
```

The keyword `static` states that they can be considered as global variables, and the keyword `final` prevents the values assigned to these variables from being changed, i.e. they are global constants. Section 7.6 on page 192 illustrates a more elegant solution for defining a fixed number of such constants.

The class `EmployeeV4` also declares two static methods: `registerGender()` and `printStatistics()` at (2). The method `registerGender()` increments the relevant counter depending on the value of its parameter, and the method `printStatistics()` prints the number of male and female employees registered at any given time. We will put these methods to use in the next subsection.

Figure 7.6 shows the UML diagram for the class `EmployeeV4` with all its members. Static members are underlined in the diagram to distinguish them from instance members.

7



## PROGRAM 7.5    Static members

```
class EmployeeV4 { // Assume that no constructors are declared.

 // (1) Static variables:
 static final boolean MALE = false;
 static final boolean FEMALE = true;
 static final double NORMAL_WORKWEEK = 37.5;
 static int numOfFemales;
```

```

static int numOfMales;

// (2) Static methods:
// Register an employee's gender by updating the relevant counter.
static void registerGender (boolean gender) {
 if (gender == FEMALE) {
 ++numOfFemales;
 } else {
 ++numOfMales;
 }
}
// Print statistics about the number of males and females registered.
static void printStatistics() {
 System.out.println("Number of females registered: " +
 EmployeeV4.numOfFemales); // (3)
 System.out.println("Number of males registered: " +
 EmployeeV4.numOfMales); // (4)
}

// Rest of the specification is the same as in class EmployeeV3
// ...

}

// Accessing static members
public class Client4A {

 public static void main(String[] args) {
 // (5) Print information in class EmployeeV4 before any objects
 // are created:
 System.out.println("Print information in class EmployeeV4:");
 System.out.println("Females registered: " +
 EmployeeV4.numOfFemales); // (6) class name
 System.out.println("Males registered: " +
 EmployeeV4.numOfMales); // (7) class name

 // (8) Create a male employee.
 EmployeeV4 coffeeboy = new EmployeeV4();
 coffeeboy.setState("Tim", "Turner", 30.00, EmployeeV4.MALE);
 coffeeboy.registerGender(EmployeeV4.MALE); // (9) referanse
 System.out.println("Print information in class EmployeeV4:");
 coffeeboy.printStatistics(); // (10) referanse

 // (11) Create a female employee.
 EmployeeV4 receptionist = new EmployeeV4();
 receptionist.setState("Amy", "Archer", 20.50, EmployeeV4.FEMALE);
 EmployeeV4.registerGender(EmployeeV4.FEMALE);
 System.out.println("Print information in class EmployeeV4:");
 EmployeeV4.printStatistics();
 }
}

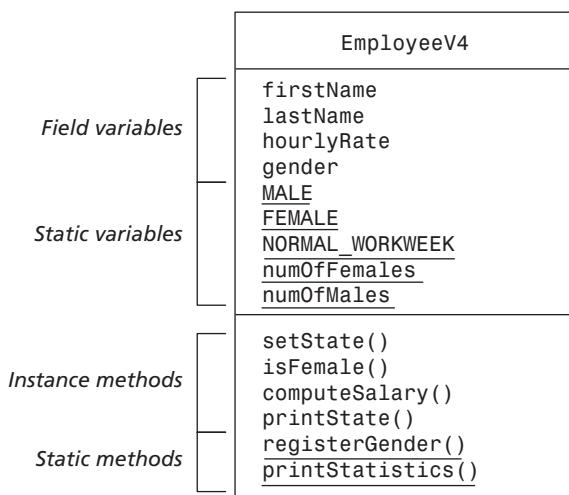
```

Program output:

```
Print information in class EmployeeV4:
Females registered: 0
Males registered: 0
Print information in class EmployeeV4:
Number of females registered: 0
Number of males registered: 1
Print information in class EmployeeV4:
Number of females registered: 1
Number of males registered: 1
```

---

FIGURE 7.6 UML diagram showing all members in a class



## Accessing static members

A static member can be accessed using the notation `className.memberName`. At (3) and (4) in Program 7.5 the method `printStatistics()` refers to the static variables `numOfFemales` and `numOfMales` using this notation: `EmployeeV4.numOfFemales` and `EmployeeV4.numOfMales`.

We can use the member name to refer to static members in the same class inside any method, provided that the name is not shadowed by a local variable. The method `registerGender()` in Program 7.5 refers to the static variables `numOfFemales` and `numOfMales` in class `EmployeeV4`.

Inside an instance method, we can also use the `this` reference to refer to static members in the same class. The `this` reference refers to the current object, and an object has a reference type (for example, its class), thus the notation `this.staticMember` uniquely identifies the member in the class.



In Program 7.1 on page 166 the method `computeSalary()` in the class `EmployeeV3` declares a local variable `normalNumOfHours` that represents the normal number of working hours in a week (37.5). If other methods need this information, this value must be duplicated locally where it is needed. If the normal number of hours changes in the future, the value must be changed in all places in the code where it is used. This value is an obvious candidate for a global constant:

```
static final double NORMAL_WORKWEEK = 37.5;
```

Now it is only necessary to change the value in one place and recompile the code. Its name makes it obvious what the value represents. The method `computeSalary()` below uses the value of the global constant `NORMAL_WORKWEEK`. This new declaration of the method shows the three ways in which we can refer to a static member inside an instance method in the same class:

```
double computeSalary(double numOfHours) {
 assert numOfHours >= 0 : "Number of hours must be >= 0";
 double weeklySalary = hourlyRate * this.NORMAL_WORKWEEK; // (1)
 if (numOfHours > EmployeeV4.NORMAL_WORKWEEK) { // (2)
 weeklySalary += 2.0 * hourlyRate * (numOfHours - NORMAL_WORKWEEK); // (3)
 }
 return weeklySalary;
}
```

Program 7.5 also shows how static members of a class can be accessed from other classes using the class name, as shown at (6) and (7). They can also be accessed using a reference of the class. The method `main()` in class `Client4A` calls the static methods `registerGender()` and `printStatistics()` in class `EmployeeV4`, using the reference `coffeeboy` at (9) and (10). We don't need to create an object of the class to refer to the static members of the class.

### No this reference for static members

Static methods cannot refer to instance members by name, not even by using the `this` reference – static methods have no `this` reference. They are a part of the class, and not a part of the current object.

## Initializing static variables

Static members of a class are initialized automatically before the class is used. If a static variable specifies an initial value, the static variable will be initialized with that value, otherwise it will be initialized with the default value of its data type specified in the declaration. Output from (6) and (7) in Program 7.5 correctly shows that the static variables `EmployeeV4.numOfFemales` and `EmployeeV4.numOfMales` are initialized to the default value 0.



## BEST PRACTICES

To aid in understanding the program, it is a good idea to initialize static variables in their declaration, and to use the class name when accessing static members.

### The `main()` method and program arguments

A Java application must have a primary class that defines a method `main` and which has the method header:

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

Execution always starts in the `main()` method of the class. In the examples we have seen so far, the program terminates when all the actions in the `main()` method have been executed.

The signature of the `main()` method above shows that it has an array of strings (`String[]`) as a formal parameter. *Program arguments* specified on the command line are stored in an array of strings. The reference value of this array is assigned to the formal parameter `args` before the execution of the `main()` method starts. In the program, the `main()` method can obtain these program arguments from the array using the formal parameter variable `args`. The strings in the array can be indexed the usual way, with the first string as `args[0]` and the last string as `args[args.length-1]`. If there are no program arguments, the parameter variable `args` will refer to an array of strings with zero elements, i.e. `args.length` is zero. Any attempt to index a value in this case will result in an `ArrayIndexOutOfBoundsException`, since the array has no elements.

Program 7.6 shows an example in which the program receives information about an employee from the command line in the form of program arguments. The command line specifies the following information:

```
> java Client4B Mona Lisa 20.5 true
```

The program arguments are specified after the class name. Program 7.6 checks at (1) that the array `args` has exactly four program arguments. If this is the case, the program arguments are printed at (2). The values in the `args` array are assigned to local variables at (3). At (4) and (5) the string values are converted to values of the correct type. An `Employee` object is created and assigned these values at (6). Note that the values from the command line are passed as strings, and must be explicitly converted to other values if it is necessary in the program.

#### PROGRAM 7.6    Reading program arguments

```
// Using program arguments
public class Client4B {

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



```

// (1) Check that all information about an employee is given
// on the command line:
if (args.length != 4) {
 return;
}

// (2) Print the array args:
System.out.println("Program arguments:");
for (String arg : args) {
 System.out.println(arg);
}

// (3) Assign information from the array args to local variables:
String firstName = args[0];
String lastName = args[1];
double hourlyRate = Double.parseDouble(args[2]); // (4) Floating-point
boolean gender;
if (args[3].equals("true")) { // (5) Boolean value
 gender = EmployeeV4.FEMALE;
} else {
 gender = EmployeeV4.MALE;
}

// (6) Create an employee, and print its state:
EmployeeV4 decorator = new EmployeeV4();
decorator.setState(firstName, lastName, hourlyRate, gender);
System.out.println("Information about an employee:");
decorator.printState();
System.out.printf("Salary: %.2f%n", decorator.computeSalary(40.0));
}
}

```

Program output:

```

> javac EmployeeV4.java Client4B.java
> java Client4B Mona Lisa 20.5 true
Program arguments:
Mona
Lisa
20.5
true
Information about an employee:
First name: Mona Last name: Lisa Hourly rate: 20.50 Gender: Female
Salary: 871.25

```



## 7.5 Initializing object state

### Default constructors: implicit or explicit

Constructors have a special role in a class declaration. The use of the `new` operator, together with a constructor call, results in the execution of the constructor that corresponds to the constructor call. The main purpose of a constructor is to set the initial state of the current object, i.e. the object that has been created by the `new` operator.

If the class `EmployeeV5` does not declare a constructor, the compiler will generate a constructor for the class equivalent to the following declaration:

```
EmployeeV5() { ... } // (1)
```

Note the use of the class name and the absence of parameters in the constructor header. The actual contents of the constructor body are not important for this discussion, except to note that the constructor body has *no* actions to initialize the state of the current object.

The constructor declaration resembles a method declaration, but it is not a method. It is called the *default constructor* for the class `EmployeeV5`. Because it was generated by the compiler, it is called the *implicit default constructor*. If we create an object of this class:

```
EmployeeV5 cook = new EmployeeV5(); // (2)
```

the constructor call `EmployeeV5()` will result in the constructor at (1) being executed. This constructor has no effect on the state of the current object.

As we can see, the implicit default constructor is not always adequate, as it does *not* effect the state of the current object. The field variables will always be initialized to their default values (or to any initial value specified in the field declarations). Therefore a class will usually provide explicit constructors to set the initial state of an object. This ensures that the object is initialized properly before it is used.

A class can choose to declare an *explicit default constructor*. For example, the class `EmployeeV5` declares such a constructor at (3):

```
class EmployeeV5 {
 // Explicit default constructor
 EmployeeV5() { // (3) No parameters
 firstName = "Joe";
 lastName = "Jones";
 hourlyRate = 15.50;
 gender = MALE;
 }
 // Rest of the specification is the same as in class EmployeeV4
}
```



Now the constructor call at (2) above will result in the explicit default constructor at (3) being executed. Each object created in this way will always have the same state, as shown in Figure 7.2b on page 164. A standard constructor is not adequate, however, for situations that require a more customised initial object state.

A constructor always has the same name as the class, so that the signature comprises the class name and type of the formal parameters. It is called in conjunction with the `new` operator to return the reference value of the object that has been created. A constructor cannot return a value. Apart from that, the constructor body can contain declarations and actions, similar to an instance method body. A constructor can use the `this` reference to refer to the current object, and all members in the class can be accessed in the constructor body.

## Constructors with parameters

A class can declare constructors with formal parameters, to create objects with appropriate initial states. Constructors with parameters are called *non-default constructors*.

Program 7.7 shows the use of non-default constructors to initialize fields in objects created with the `new` operator. Declaration (2) and (3) create two objects of the class `EmployeeV6`. Parameter passing takes place in the same way as for method calls. The initial state of the objects referred to by the references `operator1` and `operator2` are shown in Figure 7.4a and Figure 7.4b respectively. In addition to initializing the state of the object with the values of the actual parameter expressions in the constructor call, the constructor in the class `EmployeeV6` also calls the static method `registerGender()` to update the count of male and female employees.

If a class declares *any* constructor, the *implicit* default constructor cannot be applied. In Program 7.7, the declaration at (1) results in a compiler time error:

```
EmployeeV6 clerk = new EmployeeV6(); // (1) Compile time error!
```

The signature of the constructor call at (1) is not compatible with the signature of the non-default constructor:

```
EmployeeV6(String, String, double) // Non-default constructor signature
```

The class `EmployeeV6` must declare the default constructor *explicitly* to allow this constructor to be called.

### BEST PRACTICES

Always declare an explicit default constructor.



**FIGURE 7.7** Initializing of object state with non-default constructors

```
EmployeeV6 operator1 = new EmployeeV6("Tim", "Tanner", 20.60, EmployeeV6.MALE); // (2)
EmployeeV6 operator2 = new EmployeeV6("Amy", "Archer", 18.50, EmployeeV6.FEMALE); // (3)
```

| operator1:EmployeeV6 |
|----------------------|
| lastName = "Tim"     |
| firstName = "Tanner" |
| hourlyRate = 20.60   |
| gender = false       |

(a)

| operator2:EmployeeV6 |
|----------------------|
| lastName = "Amy"     |
| firstName = "Archer" |
| hourlyRate = 18.50   |
| gender = true        |

(b)

**PROGRAM 7.7** Non-default constructors

```
class EmployeeV6 {
 // Only non-default constructor
 EmployeeV6(String fName, String lName, double hRate, boolean gender) {
 this.firstName = fName; // field access via this reference
 this.lastName = lName;
 this.hourlyrate = hRate;
 this.gender = gender;
 this.registerGender(gender); // call to static method
 }
 // Rest of the specification is the same as in class EmployeeV4
 // ...
}
// Using constructors
public class Client6 {

 public static void main(String[] args) {

 // EmployeeV6 clerk = new EmployeeV6(); // (1) Compile time error!
 // No default constructor.

 // Print information in class EmployeeV6
 System.out.println("Print information in class EmployeeV6:");
 EmployeeV6.printStatistics();
 System.out.println();

 // Create an employee, and print its information
 EmployeeV6 operator1 = new EmployeeV6("Tim", "Turner", 30.00,
 EmployeeV6.MALE); // (2)
 printEmployeeInfo(operator1, 40.0);
 System.out.println();

 // Create a new employee, and print its information
 EmployeeV6 operator2 = new EmployeeV6("Amy", "Archer", 20.50,
 EmployeeV6.FEMALE); // (3)
 printEmployeeInfo(operator2, 50.0);
 }
}
```



```

static void printEmployeeInfo(EmployeeV6 employee,
 double numOfHours) {
 System.out.println("Printing information about an employee:");
 employee.printState();
 System.out.printf("Salary: %.2f%n",
 employee.computeSalary(numOfHours));
 System.out.println("Print information in class EmployeeV6:");
 EmployeeV6.printStatistics();
}
}

```

Program output:

Print information in class EmployeeV6:

Number of females registered: 0

Number of males registered: 0

Printing information about an employee:

First name: Tim Last name: Turner Hourly rate: 30.00 Gender: Male

Salary: 1275.00

Print information in class EmployeeV6:

Number of females registered: 0

Number of males registered: 1

Printing information about an employee:

First name: Amy Last name: Archer Hourly rate: 20.50 Gender: Female

Salary: 1281.25

Print information in class EmployeeV6:

Number of females registered: 1

Number of males registered: 1

## Overloading constructors

If necessary a class can declare several constructors. Each constructor call will result in the execution of the constructor that has a signature compatible with the constructor call, analogous to method calls – see the subsection *Method calls and actual parameter expressions* on page 167.

Program 7.8 shows a class EmployeeV7 that declares three constructors. We say that the constructors are *overloaded*, i.e. they have the same class name, but their signatures differ because they have different formal parameter lists.

The class Client7 creates three objects of the EmployeeV7 class and prints pertinent information about them. The initial state of each object is dependent on which constructor was executed when the object was created. Creating the object at (4) results in the constructor at (1) being executed. Creating the object at (5) results in the constructor at (2) being executed. Finally, creating the object at (6) results in the constructor at (3) being executed.



## BEST PRACTICES

Any initialization code in the constructor overrides initial values specified in the field declarations. Therefore it is a good idea always to initialize fields in one place: in the constructor.

### PROGRAM 7.8 Constructor overloading

```
class EmployeeV7 {
 static final double STANDARD_HOURLY_RATE = 15.50;

 // Constructors
 EmployeeV7() {
 firstName = "Joe";
 lastName = "Jones";
 hourlyRate = STANDARD_HOURLY_RATE;
 gender = MALE;
 registerGender(MALE);
 }

 EmployeeV7(String fName, String lName, boolean gender) { // (1)
 this.firstName = fName;
 this.lastName = lName;
 this.hourlyRate = STANDARD_HOURLY_RATE;
 this.gender = gender;
 this.registerGender(gender);
 }

 EmployeeV7(String fName, String lName,
 double hRate, boolean gender) { // (2)
 this.firstName = fName;
 this.lastName = lName;
 this.hourlyRate = hRate;
 this.gender = gender;
 this.registerGender(gender);
 }

 // The rest of the specification is the same as in class EmployeeV6
}
// Using overloaded constructors
public class Client7 {

 public static void main(String[] args) {

 // Print information in class EmployeeV7
 System.out.println("Printing information in class EmployeeV7:");
 EmployeeV7.printStatistics();
 }
}
```



```

// Create an employee, and print its information
EmployeeV7 guard1 = new EmployeeV7(); // (4)
printEmployeeInfo(guard1, 50);

// Create an employee, and print its information
EmployeeV7 guard2 = new EmployeeV7("Tim", "Turner",
 EmployeeV7.MALE); // (5)
printEmployeeInfo(guard2, 40);

// Create a new employee, and print its information
EmployeeV7 guard3 = new EmployeeV7("Amy", "Archer", 20.50,
 EmployeeV7.FEMALE); // (6)
printEmployeeInfo(guard3, 35);

// Print information in class EmployeeV7
System.out.println("\nPrinting information in class EmployeeV7:");
EmployeeV7.printStatistics();
}

static void printEmployeeInfo(EmployeeV7 employee, double numOfHours) {
 System.out.println();
 System.out.println("Printing information about an employee:");
 employee.printState();
 System.out.printf("Salary: %.2f%n",
 employee.computeSalary(numOfHours));
}
}

```

Program output:

Printing information in class EmployeeV7:

Number of females registered: 0

Number of males registered: 0

Printing information about an employee:

First name: Joe Last name: Jones Hourly rate: 15.50 Gender: Male

Salary: 968.75

Printing information about an employee:

First name: Tim Last name: Turner Hourly rate: 15.50 Gender: Male

Salary: 658.75

Printing information about an employee:

First name: Amy Last name: Archer Hourly rate: 20.50 Gender: Female

Salary: 768.75

Printing information in class EmployeeV7:

Number of females registered: 1

## 7.6 Enumerated types

### Simple form of enumerated types

An *enumerated type* (also called *enum* for short) defines a fixed number of enum constants. An enumerated constant is a unique name that refers to a particular object. Figure 7.8 shows the simple form of the enum type. The keyword `enum` indicates that the declaration is an enumerated type. The enum constants are specified in a list in the block that comprises the body of the enumerated type. Thus the enum type `Weekday` defines seven enum constants, one for each day of the week.

The set of objects for an enum type is confined to the objects listed by the declaration of the enum type. These objects are created automatically only once during program execution. It is not possible to create more objects of the enum type with the `new` operator. We can consider the enum constants as global constants that are declared `static` and `final`.

The enum type `Weekday` defines a reference type. Just as with other reference types, we can declare variables of the enum type, but these variables can only refer to objects that are designated by the enum constants:

```
Weekday lateOpeningDay = Weekday.THURSDAY;
```

**FIGURE 7.8** Simple form of enumerated types



### Selected methods for enumerated types

Table 7.1 shows two methods that can be applied to all enumerated types. The method `toString()` returns the name of the enum constant. This method is applied implicitly to the parameters of the `println()` method in the following code examples:

```
System.out.println(lateOpeningDay); // Prints THURSDAY.
System.out.println(Weekday.SUNDAY); // Prints SUNDAY.
```

We can also compare constants in an enumerated type:

```
assert(lateOpeningDay != Weekday.SUNDAY); // true
assert(lateOpeningDay == Weekday.THURSDAY); // true
```

Enum constants can also be used as case labels in a `switch` statement. Given that the variable `day` has the type `Weekday`, the `switch` statement below will determine whether it is a



working day or falls on a weekend. Note that enum constants in case labels are not referred to by the name of the enum type, because it is implied by the type of the switch statement, which in this case is the type of the variable day, i.e. Weekday:

```
switch(day) { // (1)
 case SATURDAY: case SUNDAY:
 System.out.println("The day is " + day + ", it must be weekend.");
 break;
 default:
 System.out.println(day + " is a working day.");
}
```

If we need to iterate over all the constants of an enumerated type, we can first create an array with all the constants by calling the static method values(), and then use a for(:) loop to iterate over the array:

```
Weekday[] daysArray = Weekday.values();
for (Weekday day : daysArray) {
 // switch statement given at (1) above
}
```

Program 7.9 shows the execution of the code examples presented in this subsection.

TABLE 7.1 Selected method for all enumerated types

| Methods for enumerated types                 |                                                                                                                                                                                                           |
|----------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>String toString()</code>               | Returns the string representation of the current enum constant, i.e. the name of the current constant.                                                                                                    |
| <code>static enumTypeNames[] values()</code> | Returns an array with the enum constants that are declared in the enum type that has the <i>enumType-Name</i> . The order of the constants in the array is the same as the order in the enum declaration. |

PROGRAM 7.9 Use of enumerated types

```
// An enum client
public class Weekdays {
 public static void main(String[] args) {

 Weekday lateOpeningDay = Weekday.THURSDAY; // Reference of enum type

 // Method toString() applied implicitly
 System.out.println(lateOpeningDay); // Prints THURSDAY.
 System.out.println(Weekday.SUNDAY); // Prints SUNDAY.

 // Testing for equality
 assert(lateOpeningDay != Weekday.SUNDAY); // true
 }
}
```



```

assert(lateOpeningDay == Weekday.THURSDAY); // true

// Iterate over days of the week:
System.out.println("Days of the week:");
Weekday[] daysArray = Weekday.values();
for (Weekday day : daysArray) {
 switch(day) { // (1)
 case SATURDAY: case SUNDAY:
 System.out.println("The day is " + day +
 ", it must be weekend.");
 break;
 default:
 System.out.println(day + " is a working day.");
 }
}
}
}
}

```

Program output:

```

THURSDAY
SUNDAY
Days of the week:
MONDAY is a working day.
TUESDAY is a working day.
WEDNESDAY is a working day.
THURSDAY is a working day.
FRIDAY is a working day.
The day is SATURDAY, it must be weekend.
The day is SUNDAY, it must be weekend.

```

---

## General form of enumerated types

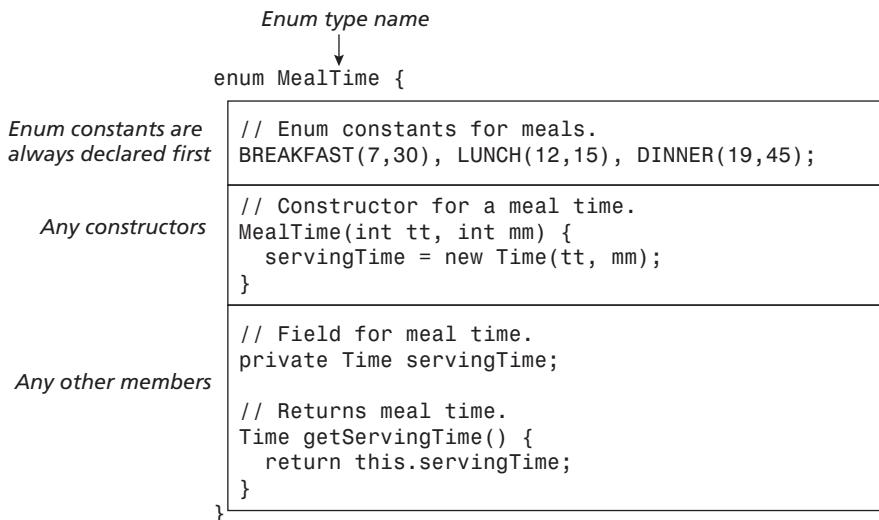
The general form of the enumerated type declaration is shown in Figure 7.9. An enumerated type can declare constructors and other members, analogous to those in a class declaration. In this way, it is possible to define properties and behaviour of enum constants in an enumerated type.

Constructors cannot be called directly. A constructor is called *implicitly* when an object representing an enum constant is created automatically from the enum declaration during execution. Each constant name in the enum type `MealTime` is declared with an *actual* parameter list. The constructor which has the corresponding formal parameter list is executed. Specifying `BREAKFAST(7, 30)` in the enum constant list results in the constructor being called with 7 and 30 as values of the formal parameters `tt` and `mm`, representing hours and minutes, respectively.



**FIGURE 7.9**

The general form of enumerated types



The enumerated type `MealTime` in Figure 7.9 declares a field, `servingTime`, of reference type `Time` (see the listing in Program 7.9). Each object of the enumerated type `MealTime` will have this field, which will be initialized by the call to the constructor. The three objects, corresponding to each of the enum constants, will have states corresponding to the serving times for the different meals. The enumerated type `MealTime` also declares an instance method, `getServingTime()`, which returns the time for serving a meal. The enumerated type `MealTime` has only three objects that are referred to by the enum constants in the declaration. We can invoke the method `getServingTime()` on these objects in the usual way:

```
System.out.println(MealTime.BREAKFAST.getServingTime()); // Prints 07:30
```

The class `MealService` in Program 7.9 prints the times when the different meals are served.

**PROGRAM 7.10**

Serving meals

```

// Time is given as hours (0-23) and minutes (0-59).
class Time {

 // Fields for the time.
 int hours;
 int minutes;

 // Constructor
 Time(int hours, int minutes) {
 assert (0 <= hours && hours <= 23 &&
 0 <= minutes && minutes <= 59) :
 "Invalid hours and/or minutes";
 this.hours = hours;
 }
}

```



```

 this.minutes = minutes;
 }

 // String representation of the time, TT:MM
 public String toString() {
 return String.format("%02d:%02d", hours, minutes);
 }
}

// Using enums
public class MealService {
 public static void main(String[] args) {

 // (1) Create an array of meals:
 MealTime[] meals = MealTime.values();

 // (2) Print meal times:
 for (MealTime meal : meals) {
 System.out.println(meal + " is served at " + meal.getServingTime());
 }
 }
}

```

Program output.

```

> javac MealTime.java Time.java MealService.java
> java -ea MealService
BREAKFAST is served at 07:30
LUNCH is served at 12:15
DINNER is served at 19:45

```

---

## Declaring enumerated types inside a class

So far we have declared an enumerated type as a top-level declaration in its own separate source file. Other clients can access enum constants by using the class name and the constant name. An enumerated type can also be declared as a member in a class declaration. It makes sense to do this if the use of the enum constants is localised to a single class. The code below shows the declaration of the enumerated type `Weekday` as a member of the class `Weekdays`. Access to the enum constants is the same as it was before (see also Program 7.9):

```

// Enum type as member in a class
public class Weekdays {
 // Enum type as member in a class.
 enum Weekday {
 MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
 }
 public static void main(String[] args) {
 // Same as before.
 }
}

```



## 7.7 Review questions

1. Identify all the members and constructors in the declaration of the class Counter. Group them as shown in Figure 7.1.

```
class Counter {
 final static int MAX_VALUE = 100;
 static String description = "This class creates counters.";
 int value;

 Counter() { value = 1; }
 Counter(int initialValue) { value = initialValue; }

 int getCounter() { return value; }
 void setCounter(int newValue) { value = newValue; }
 void incrementCounter() { ++value; }
 void decrementCounter() { --value; }
 void resetCounter() { value = 0; }

 static String getDescription() { return description; }
}
```

2. \_\_\_\_\_ belong to objects, while \_\_\_\_\_ belong to the class.

3. Values of all the field variables in an object comprise its \_\_\_\_\_.

4. What do we mean by the initial state of an object?

5. What is the default value for any reference variable?

6. In the class RectangleV2, which field variables are initialized with default values and which are initialized with an initial value when an object of the class is created?

```
class RectangleV2 {
 double length;
 double breadth;
 double area = length * breadth;
}
```

7. Which of the following statements are true?

- a Members in a class can be declared in any order.
- b A class must always provide an explicit default constructor.
- c The implicit default constructor is always executed when an object of the class is created using the new operator.
- d The explicit default constructor can have formal parameters.

8. In the class ClientA, identify the class names, local variables, method calls, formal parameters and actual parameters. The class Counter is defined in Exercise 7.1.



```
// Counting cars
public class ClientA {
 public static void main(String[] args) {
 int numOfCars = 10;
 Counter carCounter = new Counter(numOfCars);
 carCounter.incrementCounter();
 System.out.println(carCounter.getCounter());
 }
}
```

Given the class `Variables` and a reference `obj` that refers to an object of this class, what will be printed by the method call `obj.doIt(2)`.

```
class Variables {
 static String s = " is equal to ";

 double d = 20;

 int k = 10;

 void doIt(int d) {
 for (int k = 0; k < 2; k++) {
 System.out.println("k + d" + s + (k + d)); // (1)
 }
 System.out.println("k + d" + s + (k + d)); // (2)
 String s = "=";
 System.out.println("k + d" + s + (k + d)); // (3)
 }
}
```

**9.** Which statements are true about methods?

- a** The method signature comprises the return type, method name and type of the formal parameters.
- b** Formal and actual parameters must be compatible with regard to type, number and order.
- c** Parameter passing in Java entails that the values of the actual parameter expressions are assigned to the corresponding formal parameter variables.
- d** A method always returns a value.
- e** The return value must always be assigned to a variable.

7



**10.** Answer “yes” or “no” to the following questions:

- a** Is it possible to declare several methods with the same name, but different parameter lists, in the same class?
- b** If the actual parameter in a method call is a variable name, can the corresponding formal parameter variable have the same name?
- c** Can a `void` method contain a `return` statement?

- 11.** Determine whether you would use a void or a non-void method for the following tasks. If implementing a non-void method, which return type would you choose?
- a** Determine whether a person has reached retirement age.
  - b** Give an explanation to the user of what the program does.
  - c** Find the largest of two numbers.
  - d** Read an int value entered by the user.
  - e** Assign an int value to a field variable.
  - f** Create an array of Counter objects for a client (see Question 7.1).
  - g** Create a two-dimensional array of strings.
  - h** Create statistics for the number of newspapers sold per day for a four-week period, given that the weekdays are numbered from 0 upwards (Monday has weekday number 0).

- 12.** What will the following program print? The class Counter is declared in Question 7.1.

```
// Parameter passing
public class ParameterClient {
 public static void main(String[] args) {
 Counter counter = new Counter();
 int numoftimes = 5;

 System.out.println("Before method call: " +
 "counter value is equal to " + counter.getCounter() +
 " and number of times is " + numoftimes);
 updateCounter(counter, numoftimes);
 System.out.println("After method call: " +
 "counter value is equal to " + counter.getCounter() +
 " and number of times is " + numoftimes);
 }

 static void updateCounter(Counter counter, int numoftimes) {
 for (; numoftimes > 0; --numoftimes) {
 counter.incrementCounter();
 }
 }
}
```

- 13.** What will the following program print? The class Counter is declared in Question 7.1.

```
// More parameter passing
public class SwapClient {
 public static void main(String[] args) {
 int startvalue = 10;
 Counter counter1 = new Counter(startvalue);
 Counter counter2 = new Counter(startvalue * 2);
 System.out.println("Before swapping: " +
 "counter1 is " + counter1.getCounter() +
 " and counter2" + " is " + counter2.getCounter());
```



```

 swap(counter1, counter2);
 System.out.println("After swapping: " +
 "counter1 is " + counter1.getCounter() +
 " and counter2" + " is " + counter2.getCounter());
 }

 static void swap(Counter counter1, Counter counter2) {
 Counter t3 = counter1;
 counter1 = counter2;
 counter2 = t3;
 }
}

```

- 14.** Given the following declarations, where the class `Counter` is declared as in Question 7.1:

```

Counter counterA = new Counter();
Counter[] counterArrayA = new Counter[5];
Counter[][] counterArrayB = new Counter[2][3];

```

and the following method declaration:

```

static void playingWithParameters(Counter counter, Counter[] array)
{ /* ... */ }

```

Which of these method calls are valid?

- a `playingWithParameters(counterA, counterArrayA[]);`
- b `playingWithParameters(counterA, counterArrayA);`
- c `playingWithParameters(counterArrayA[2], counterArrayA);`
- d `playingWithParameters(counterArrayA[3], counterArrayB[0]);`
- e `playingWithParameters(counterArrayB[1][3], counterArrayA);`
- f `playingWithParameters(counterArrayB[0][2], counterArrayB[1]);`
- g `playingWithParameters(counterA, counterArrayB);`

- 15.** Rewrite the class `Counter` declared in Question 7.1 using the `this` reference explicitly.

- 16.** Which of these statements are valid, given the declaration of the class `Counter` from Question 7.1, and that `ref` refers to an object of this class?

- a `System.out.println(ref.value);`
- b `System.out.println(Counter.value);`
- c `System.out.println(ref.MAX_VALUE);`
- d `System.out.println(Counter.MAX_VALUE);`



- e** System.out.println(ref.getCounter());
- f** System.out.println(Counter.getCounter());
- g** System.out.println(ref.getDescription());
- h** System.out.println(Counter.getDescription());

**17.** Which of the following statements are true about constructors?

- a** A constructor is a method.
- b** A constructor must specify a return type.
- c** If a class declares more than one constructor, then these are overloaded.
- d** Actions in a constructor can refer to all members of the class.
- e** The this reference cannot be used in a constructor.

**18.** Given the following class:

```
// Being four-sided
public class FourSided {
 public static void main(String[] args) {
 Rectangle shape = new Rectangle();
 System.out.println(shape.length * shape.breadth);
 }
}
```

What will the program above print if we use the following declarations for the class Rectangle?

- a** class Rectangle {  
 double length;  
 double breadth;  
}
- b** class Rectangle {  
 double length = 20;  
 double breadth = 30;  
}
- c** class Rectangle {  
 double length = 20;  
 double breadth = 30;  
  
 Rectangle() {  
 length = 10;  
 breadth = 5;  
 }
}

**19.** Given the following variable declarations (1) and (2):

```
Counter counter1 = new Counter(); // (1)
Counter counter2 = new Counter(2); // (2)
```



Determine whether the variable declarations (1) and (2) are valid if we use the following declarations of the class Counter:

a class Counter {  
 int value;  
}

b class Counter {  
 int value;  
 Counter() { value = 1; }  
}

c class Counter {  
 int value;  
 Counter(int startValue) { value = startValue; }  
}

d class Counter {  
 int value;  
 Counter() { value = 1; }  
 Counter(int startValue) { value = startValue; }  
}

20. Which lines of code in the declaration of the NewCounter class will result in a compile time error?

```
class NewCounter {
 static String description = "A new class that creates counters.";

 int value;

 NewCounter(int initialValue) { // A constructor
 System.out.println(value); // (1)
 System.out.println(this.value); // (2)
 System.out.println(NewCounter.value); // (3)
 System.out.println(description); // (4)
 System.out.println(this.description); // (5)
 System.out.println(NewCounter.description); // (6)
 value = initialValue;
 }

 void resetCounter() { // An instance method
 NewCounter t1 = new NewCounter(10);
 System.out.println(value); // (7)
 System.out.println(this.value); // (8)
 System.out.println(t1.value); // (9)
 System.out.println(NewCounter.value); // (10)
 System.out.println(description); // (11)
 System.out.println(this.description); // (12)
 System.out.println(t1.description); // (13)
 System.out.println(NewCounter.description); // (14)
 value = 0;
 }
}
```



```

 }

 static String getDescription() { // A static method
 NewCounter t1 = new NewCounter(10);
 System.out.println(value); // (15)
 System.out.println(this.value); // (16)
 System.out.println(t1.value); // (17)
 System.out.println(NewCounter.value); // (18)
 System.out.println(description); // (19)
 System.out.println(this.description); // (20)
 System.out.println(t1.description); // (21)
 System.out.println(NewCounter.description); // (22)
 return description;
 }
}

```

- 21.** Which of the following statements are not true of enumerated types?
- a** An enumerated type is a reference type defined with the keyword `enum`.
  - b** An enumerated type can be declared in its own separate source code file.
  - c** An enumerated type can declare constructors.
  - d** An enumerated type can declare members, as in a class declaration.
  - e** Enum constants must always be declared first in an enumerated type declaration.
  - f** We can use the `new` operator to create objects of an enumerated type.
- 22.** Given the following declaration of an enumerated type:

```
enum LightColour { RED, YELLOW, GREEN }
```

and the following variable declaration:

```
LightColour colour;
```

Which of these statements will not compile?

- a** `colour = GREEN;`
- b** `for (LightColour colour : LightColour) {  
 System.out.println(colour);  
}`
- c** `switch(colour)  
 case LightColour.RED: System.out.println("STOP!"); break;  
 case LightColour.GREEN: System.out.println("GO!"); break;  
 case LightColour.YELLOW: System.out.println("CAREFUL!"); break;  
 default: assert false: "UNKNOWN COLOUR!";  
}`
- d** `Boolean b = colour.equals(GREEN);`
- e** `LightColour.GREEN = colour;`



## 7.8 Programming exercises

1. Modify the class `Counter` from Question 7.1 on page 197 so that it meets the following criteria:

- It is possible to create a counter that counts within an interval. The interval is given by a lower and an upper limit. It should be possible to specify the initial value of the counter, which must be within the interval.
- If no initial value is specified together with the interval, counting starts from the lower limit of the interval.
- If no interval is specified, but the initial value *is* specified, counting starts with the initial value and continues upwards.
- If neither the interval nor the initial value are specified, counting starts at 0 and continues upwards.
- It should be possible to increment and decrement a counter.
- It should be possible to get the current value of a counter, and to reset a counter to an initial value.
- Where necessary, check that the counter has a valid value.

Write a separate class to test the class `Counter`.

2. We will write a class called `Forex` that can be used for converting between two currencies. An object of the class stores the exchange rate between two currencies (for example, NOK 11.54 = GBP 1). The class offers two methods for converting between the two currencies. Choose suitable names for all members.

Write a client that creates several `Forex` objects and tests conversion between different currencies.

3. Write a class `Reverse` that reads program arguments and prints them in reverse. In the printout the arguments should be separated by comma (,). Write a separate method that takes care of the printout.

Example of program execution:

```
> java Reverse To be or not to be
be, to, not, or, be, To
```

4. Given the following skeleton of a class:

```
class Rectangle {
 double length;
 double breadth;
 // ...
}
```

Modify the class `Rectangle` so that:

- It is possible to create an object of the class `Rectangle` without specifying the dimensions, and the dimensions in this case should be initialized to the value 1.0.
- It is possible to create a rectangle whose dimensions can be specified.



- The dimensions of a rectangle are never less than 0.0.

The class should offer methods for getting and setting the dimensions of a rectangle. The class should also offer methods to compute the area and the perimeter of a rectangle.

Write a client to test the class `Rectangle`.

5. In a program we will write a class to represent water bottles. The water bottles have the following properties:

- All bottles have a maximum capacity, measured in litres.
- All bottles contain a quantity of water at any given time, also measured in litres.

We can perform the following operations on the bottles:

- Fill a bottle completely from the tap.
- Empty a bottle.
- Pour water from one bottle to another bottle.

Write a class called `Bottle` that implements the properties described above. Let the constructor of the class allow the maximum capacity of a bottle to be specified. To begin with, let all new bottles be empty. Which instance variables should be declared in the class `Bottle`?

Implement the following methods:

- a `quantity()`, which returns the amount of water in the current bottle at the moment.
- b `remaining()`, which returns the amount of water that can be filled in the current bottle before it is full.
- c `fillFully()`, which fills the current bottle completely.
- d `empty()`, which empties the current bottle.
- e `pour(Bottle b)`, which pours water from the bottle `b` to the current bottle. The amount of water poured into the current bottle is limited either by the capacity of the current bottle or by the amount of water in bottle `b`.

Implement auxiliary methods where necessary.

6. Write a client that uses the `Bottle` class from Exercise 7.5. The client should be able to do the following:

- a Create two `Bottle` objects: one two-litre bottle and one seven-litre bottle.
- b Fill the seven-litre bottle completely.
- c Pour water from the seven-litre bottle to the two-litre bottle.
- d Empty the two-litre bottle.
- e Print the amount of water in each bottle.

How many litres of water are left in the seven-litre bottle in the end?



- 7.** Based on Exercise 7.5, write a client that creates two `Bottle` objects: one three-litre bottle and one five-litre bottle.  
Find a way of filling the five-litre bottle with only four litres of water, using only combinations of methods `fillFully()`, `empty()` and `pour()` on the two bottles.
- 8.** In Program 7.5 on page 180 the class `EmployeeV4` defines the constants `EmployeeV4.MALE` and `EmployeeV4.FEMALE`. Write a separate source file with the declaration of an enumerated type called `Gender` that defines the enum constants `MALE` and `FEMALE`. Rewrite the classes `EmployeeV4` and `Client4A` to use this enumerated type.



## Object communication

### LEARNING OBJECTIVES

By the end of this chapter, you will understand the following:

- Eliminating source code duplication.
- Assigning suitable responsibilities and roles to objects.
- Making objects cooperate to perform tasks.
- Linking objects together so that they can find each other.
- How the compiler selects a method when there are several methods with the same name.
- Documenting source code to aid maintenance and further development.

### INTRODUCTION

This chapter shows how to create programs in which objects cooperate to solve problems. By writing classes that have well-defined responsibilities, we can build large but easy to understand programs.

#### 8.1 Responsibilities and roles

All code in a Java program must be defined within classes. In a well-designed program, it is not arbitrary which class contains which piece of code. Each object in a program is given a specific role. The properties and behaviour that are defined in a class determine the responsibilities the class has and the roles the objects of this class will play during program execution.

Being able to identify the role that each object has during execution makes it easier to understand the program as a whole. Classes that combine related properties and behaviour are a lot easier to understand than classes in which the responsibilities have been spread in an arbitrary manner.

## A naive solution to a given problem

We want to create a program that reads information about two employees from the terminal window, and prints a report stating which of them has the higher hourly rate. Using local variables, as shown in Program 8.1, provides a simple but naive solution. The program code fulfils the requirements of the stated problem, but, as we shall see, the code has some deficiencies that should be corrected.

**PROGRAM 8.1 Naive employee information processing**

```
import java.util.Scanner;
public class UglyComparison {
 public static void main(String[] args) {
 Scanner input = new Scanner(System.in);
 String firstNameA, firstNameB, lastNameA, lastNameB;
 double hourlyRateA, hourlyRateB;

 System.out.printf("Input information about employee A:\n");
 System.out.printf("- First name: "); // (1)
 firstNameA = input.nextLine();
 System.out.printf("- Last name: ");
 lastNameA = input.nextLine();
 System.out.printf("- Hourly rate: ");
 hourlyRateA = input.nextDouble();
 input.nextLine(); // skip the rest of the line

 System.out.printf("Input information about employee B:\n");
 System.out.printf("- First name: "); // (2)
 firstNameB = input.nextLine();
 System.out.printf("- Last name: ");
 lastNameB = input.nextLine();
 System.out.printf("- Hourly rate: ");
 hourlyRateB = input.nextDouble();
 input.nextLine(); // skip the rest of the line

 if (hourlyRateA == hourlyRateB) {
 System.out.printf("Both earn $%.2f per hour.\n", hourlyRateA);
 } else if (hourlyRateA > hourlyRateB) {
 System.out.printf("%s %s has the higher hourly rate: $%.2f "
 + "per hour.\n", firstNameA, lastNameA, hourlyRateA); // (3)
 } else {
 System.out.printf("%s %s has the higher hourly rate: $%.2f "
 + "per hour.\n", firstNameB, lastNameB, hourlyRateB); // (4)
 }
 }
}
```



```
 }
}
```

Program execution example:

Input information about employee A:

- First name: **Gina**
- Last name: **Fillion**
- Hourly rate: **31.65**

Input information about employee B:

- First name: **Edward**
- Last name: **Wolfram**
- Hourly rate: **29.75**

Gina Fillion has the higher hourly rate: \$31.65 per hour.

---

It is apparent that Program 8.1 contains a lot of duplicated code. Code duplication is an indication that the source code should be restructured. In this example the code at (1) and (2) reads information about each of the employees in exactly the same manner, apart from storing the result in different variables. Imagine the code duplication if the program had to read information about a hundred employees from the terminal window. This type of code duplication can be difficult to maintain, since modifying the behaviour requires changing the code in several places.

Because the information is spread across various local variables, there is no simple way to keep track of which variables belong together. The programmer must keep track of the fact that `hourlyRateA` is associated with `firstNameA` and not `firstNameB`. Both lines (3) and (4) perform the same operation, except that (3) uses `firstNameA`, `lastNameA` and `hourlyRateA`, while (4) uses `firstNameB`, `lastNameB` and `hourlyRateB`. Only the programmer knows the connection between the first name, the last name and the hourly rate of an employee, and must therefore state these connections manually.

## Combining properties and behaviour in classes

The next revision of the program uses a class that represents employees. The class `EmployeeV7` in Program 7.8 on page 190 defines related properties and behaviour about an employee. This class can be used to simplify the code from Program 8.1, as shown in Program 8.2.

### PROGRAM 8.2 Combining related properties and behaviour

```
import java.util.Scanner;

public class BetterComparison {
 public static void main(String[] args) {

 // Uses the class EmployeeV7 from Chapter 7.
 EmployeeV7 employeeA = new EmployeeV7();
 EmployeeV7 employeeB = new EmployeeV7();
```



```

System.out.printf("Input information about employee A:\n"); // (1)
inputEmployeeInfo(employeeA);

System.out.printf("Input information about employee B:\n"); // (2)
inputEmployeeInfo(employeeB);

if (employeeA.hourlyRate == employeeB.hourlyRate) {
 System.out.printf("Both earn $%.2f per hour.\n",
 employeeA.hourlyRate);
} else {
 EmployeeV7 higherPaid;
 if (employeeA.hourlyRate > employeeB.hourlyRate) { // (3)
 higherPaid = employeeA;
 } else {
 higherPaid = employeeB;
 }

 System.out.printf("%s %s has the higher hourly rate: $%.2f "
 + "per hour.\n",
 higherPaid.firstName, // (4)
 higherPaid.lastName,
 higherPaid.hourlyRate);
}
}

static void inputEmployeeInfo(EmployeeV7 anEmployee) {
 Scanner input = new Scanner(System.in);
 System.out.print("- First name: ");
 anEmployee.firstName = input.nextLine();
 System.out.print("- Surname: ");
 anEmployee.lastName = input.nextLine();
 System.out.print("- Hourly rate: ");
 anEmployee.hourlyRate = input.nextDouble();
 input.nextLine(); // skip rest of the line
}
}

```

---

Most of the duplicated code in Program 8.1 is eliminated in Program 8.2. Code for reading information about an employee has been moved to the helper method `inputEmployeeInfo()`. This helper method accepts a reference to an employee. The `main()` method calls the helper method twice instead of repeating the code for reading information about each employee.

Users of the new version of the program will not see any change in its behaviour compared to Program 8.1. Programmers, on the other hand, will now have less difficulty in understanding the source code. Since Program 8.2 now uses the `EmployeeV7` class, it is easy to



see that:

- The program reads information about exactly two employees.
- The same information is needed for both employees.
- The information read for both employees is the first name, the last name and the hourly rate.
- If one employee earns more than the other, the phrasing of the report that is printed is identical, regardless of which employee earns the most.

## Using references as parameters

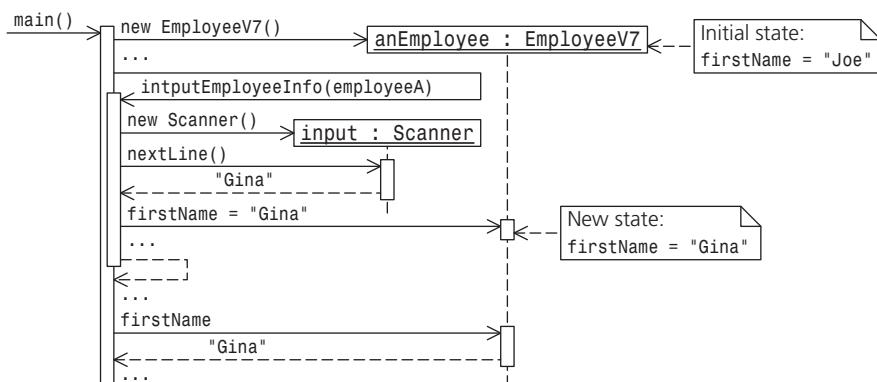
The `main()` method in Program 8.1 constructs two objects of class `EmployeeV7` that are referred to by the reference variables `employeeA` and `employeeB`. Information is read into each of these objects by calling the `inputEmployeeInfo()` method at (1) and (2).

The `inputEmployeeInfo()` method accepts a reference value of an employee object. The method can use this reference value to change the state of the object by reading the first name, the last name and the hourly rate into the field variables of the object.

The first time that the `inputEmployeeInfo()` method is called, the parameter `anEmployee` refers to the same object as `employeeA`. The second time it is called, the parameter refers to the same object as `employeeB`. In other words: the same code in the `inputEmployeeInfo()` method can be used to change the state of any employee object simply by passing a reference to the object.

The `inputEmployeeInfo()` method does not return a value, but instead stores the information it reads from the terminal window in the employee object. Since the `main()` method has references to the employee objects, it can later retrieve this information from the objects. Figure 8.1 shows how the `inputEmployeeInfo()` method stores a first name in an `EmployeeV7` object, and how the `main()` method later retrieves the same first name from the `EmployeeV7` object.

**FIGURE 8.1** Modifying the state of an object given as a parameter



Passing reference values of objects as parameters is quite common in programming. Each time `System.out.printf()` is called, the first argument given to the method is a reference



value to a `String` object. Unlike the `inputEmployeeInfo()` method, the `printf()` method does not modify the state of the object whose reference value is passed as an argument.

With the help of the `inputEmployeeInfo()` method we have now eliminated the duplicated code for reading employee information. The duplicated code for printing the report can also be eliminated by splitting the task into two parts:

- a Select the object that should be processed: find the object that represents the employee with the higher hourly rate.
- b Perform operations on the selected object: retrieve and report information from the employee object.

In the `if` statement at (3) in Program 8.2, the reference variable `higherPaid` is assigned the reference value of the `EmployeeV7` object with the higher hourly rate. The code at (4) can then use this reference value to perform operations on the object. The printing done at (4) is executed in exactly the same manner, regardless of which object was selected at (3).

## Advantages of good abstractions

The `EmployeeV7` class is responsible for keeping track of the first name, the last name, the hourly rate and the gender of an employee. The comparison program only uses some of the functionality provided by the `EmployeeV7` class. A simple extension to the program would for example be to print the weekly salary, assuming a 37.5 hour work week. To implement this, we need only insert this line after (4):

```
System.out.printf("The weekly salary for a 37.5 hour standard "
+ "work week is %$.2f", higherPaid.computeSalary(37.5));
```

Another advantage of placing the responsibility for specific operations on the actual objects is that different parts of a program can then take advantage of the operations.

Part of the challenge of creating a good program structure is deciding which responsibilities should be given to which class. Often there is no single obvious correct answer to how to distribute responsibilities. However, a rule of thumb is to try to minimize the amount of code and data duplication in the program. The design decisions that are made should be written down in the source code documentation. This will aid programmers who did not participate in the development process.

## Rethinking responsibility

Having found an acceptable solution for the problem posed at the beginning of the chapter, let us re-examine the responsibilities of the `EmployeeV7` class from Chapter 7. Each object of this class represents an employee and has fields to store pertinent information about an employee.

The `EmployeeV7` class is also responsible for keeping track of the number of male and female employees. This particular functionality was used in Chapter 7 to illustrate *static members*. However, it is time to reconsider whether this is a suitable responsibility for an employee class. This type of gender statistics typically is stored in the employee register of a company. We should consider removing this functionality from the employee class.



Program 8.2 defines the method `inputEmployeeInfo()`, which obtains information about an employee from the user. It is worth considering whether this method should be moved out of the `BetterComparison` class. We would like to collect all operations that interact with the terminal window into a single class, which would make it easier to replace the terminal window with a different data input strategy. The employee class is no longer responsible for interacting with the terminal window, and should therefore no longer provide a `printState()` method.

## 8.2 Communication and cooperation

A program typically uses many objects to accomplish a task. Each object is given a specific role, and the objects cooperate to perform the whole task.

In Java an object interacts with another object, either by calling a method, or by manipulating a field variable of the other object. In Program 8.2 the methods of the `BetterComparison` class interact with the `Scanner` object by calling methods, and interact with the `EmployeeV7` objects by storing and retrieving values in their field variables. Since no `BetterComparison` object is created and the methods of the `BetterComparison` class are static, this interaction cannot be characterised as *cooperation between objects*. However, it illustrates that this cooperation is possible through method calls and access to field variables.

### Dividing and assigning responsibility

The `BetterComparison` class in Program 8.2 contains both program logic (in the `main()` method) and code for user interaction (in the `inputEmployeeInfo()` method). The latter is likely to change over time, for example if we decide to develop a graphical user interface (GUI). We therefore want to give the responsibility of all user interaction in the program to a new object, thereby collecting all user interface code in one location in the program. The class declaration in Program 8.3 defines the behaviour of such an object. All code dealing with the terminal window is now collected in the class `TextUserInterface`. This class uses a new class `Employee` and the enumerated type `Gender`: these are defined in Program 8.4.

The new definition of the `Employee` class has more tightly focused responsibilities than earlier revisions of the class: it is responsible for storing information and calculating the weekly salary. Everything related to gender statistics has been removed, since we decided that this lies outside the responsibilities of an employee.

### PROGRAM 8.3 Class responsible for communicating with the user

```
import java.util.Scanner;

class TextUserInterface {
 Scanner input = new Scanner(System.in);

 String requestString(String wantedInfo) {
 System.out.printf("- %s: ", wantedInfo);
```



```

 return input.nextLine();
 }

 double requestDouble(String wantedInfo) {
 System.out.printf("- %s: ", wantedInfo);
 double value = input.nextDouble();
 input.nextLine(); // skip the rest of the line
 return value;
 }

 boolean requestYesNoAnswer(String wantedInfo) {
 String answer = requestString(wantedInfo + " (y/n)");
 return answer.equals("y");
 }

 Employee inputEmployee(String label, boolean inputGender) {
 System.out.printf("Input information about %s:\n", label);
 String firstName = requestString("First name");
 String lastName = requestString("Last name");
 double hourlyRate = requestDouble("Hourly rate");
 Gender gender = Gender.UNKNOWN;

 if (inputGender) {
 gender = requestGender();
 }

 return new Employee(firstName, lastName, hourlyRate, gender);
 }

 Gender requestGender() {
 if (requestYesNoAnswer("Is this a woman?")) {
 return Gender.FEMALE;
 }

 return Gender.MALE;
 }

 void reportSameHourlyRate(double hourlyRate) {
 System.out.printf("Both earn $%.2f per hour.\n", hourlyRate);
 }

 void reportHigherHourlyRate(Employee employee) {
 System.out.printf("%s %s has the higher hourly rate: $%.2f "
 + "per hour.\n",
 employee.firstName,
 employee.lastName,
 employee.hourlyRate);
 }

 void reportIncomeDifference(Employee employee, Employee manager,

```



```

 double income_difference)
{
 System.out.printf("%s %s earns $%.2f less than the manager %s %s "
 + "during a normal work week\n",
 employee.firstName, employee.lastName,
 income_difference,
 manager.firstName, manager.lastName);
}

void reportPercentageWomen(int employeeCount, double percentageWomen) {
 System.out.printf("The company has %d employees, %.1f% of which are "
 + "women.\n", employeeCount, percentageWomen);
}
}

```

---

#### PROGRAM 8.4 Employee class with clear responsibilities

```

// File: Employee.java
class Employee {
 static final double STANDARD_HOURLY_RATE = 15.50;
 static final double NORMAL_WORKWEEK = 37.5;
 static final double OVERTIME_COMPENSATION_PERCENTAGE = 200.0;
 static final double OVERTIME_COMPENSATION_FACTOR =
 OVERTIME_COMPENSATION_PERCENTAGE / 100.0;

 String firstName;
 String lastName;
 double hourlyRate;
 Gender gender;
 Employee boss; // (1)

 Employee(String fName, String lName, double hRate, Gender gender) {
 this.firstName = fName;
 this.lastName = lName;
 this.hourlyRate = hRate;
 this.gender = gender;
 }

 double overtimeHourlyRate() { // (2)
 return hourlyRate * OVERTIME_COMPENSATION_FACTOR;
 }

 double overtimeHours(double hoursWorkedDuringAWeek) { // (3)
 assert hoursWorkedDuringAWeek >= 0 : "Number of hours must be >= 0";
 double overtime = hoursWorkedDuringAWeek - NORMAL_WORKWEEK;

 if (overtime < 0) {
 return 0;
 }
 }
}

```

```

 }

 return overtime;
}

double computeSalary(double hoursWorkedDuringAWeek) { // (4)
 return NORMAL_WORKWEEK * hourlyRate
 + overtimeHours(hoursWorkedDuringAWeek) * overtimeHourlyRate();
}
}

// File: Gender.java
enum Gender { UNKNOWN, MALE, FEMALE }

```

---

Program 8.2 can now be rewritten to use the following three objects: one `TextUserInterface` object that communicates with the user through the terminal window, supplying information to two `Employee` objects. This rewrite results in the `GoodComparison` class shown in Program 8.5. Note that some of the methods in the `TextUserInterface` class are not used in Program 8.5. These methods will be used later in Program 8.6 and Program 8.7.

### PROGRAM 8.5 Responsibilities and object communication

```

public class GoodComparison {
 public static void main(String[] args) {
 TextUserInterface ui = new TextUserInterface(); // (1)
 Employee employeeA = ui.readEmployee("employee A", false); // (2)
 Employee employeeB = ui.readEmployee("employee B", false); // (3)

 if (employeeA.hourlyRate > employeeB.hourlyRate) {
 ui.reportHighestHourlyRate(employeeA);
 } else if (employeeB.hourlyRate > employeeA.hourlyRate) {
 ui.reportHighestHourlyRate(employeeB);
 } else {
 ui.reportSameHourlyRate(employeeA.hourlyRate);
 }
 }
}

```

---

### Clarifying the responsibilities of classes in the program

The `TextUserInterface` object created at (1) in Program 8.5 will be used for all communication with the user. Each of the two calls to the `readEmployeeInfo()` method at (2) and (3) asks the `TextUserInterface` object to create an `Employee` object and set its state with the information that the user enters. The string argument passed for each method call is used as a prompt to indicate to the user what input is expected. The reference values that are returned from the method calls are stored in the variables `employeeA` and `employeeB`.



The method `reportHigherHourlyRate()` in the `TextUserInterface` object is called to report when one employee earns more than the other employee. Here there is a clear separation between the responsibility of the `main()` method of the `GoodComparison` class, the `TextUserInterface` class, and the `Employee` class:

- The `main()` method in the `GoodComparison` class knows how to determine which employee is paid more, but knows nothing about obtaining information about employees or how to report the difference in salary to the user.
- The `TextUserInterface` class knows how to obtain information about an employee from the user, and how to report the difference in salary to the user, but knows nothing about determining which employee is paid more.
- The responsibility of the `Employee` class is to store information about employees and compute salaries, but not to determine which employee earns more, nor read from or print information to the terminal window.

With the help of the user interface abstraction, the `main()` method has now been freed from communicating with the user directly: instead it simply delegates this work to an `TextUserInterface` object. Changing the way the `TextUserInterface` class communicates with the user would affect neither the `GoodComparison` class nor the `Employee` class as long as the `TextUserInterface` object still fulfils its role in the program.

## Communication between objects at runtime

All communication between objects in the program is done through method calls, and information flows from one object to another through parameters, return values and field access. Methods and fields are accessed using object references. The big question now becomes how to obtain reference values that point to the object with which we want to work. Up to now all the reference values we have needed have been available in local variables that are available only within the current method call. Section 8.3 shows how to keep reference values available even after the method call has ended.

Running the programs `UglyComparison`, `BetterComparison` and `GoodComparison` will give exactly the same results. One could ask what has actually been achieved by restructuring the program, given that its behaviour has remained unchanged. The answer is that the last revision of the program is easier to understand, maintain, and develop further. The benefits may not seem like much for small programs, but this type of restructuring is essential to prevent large programs from becoming complex and unwieldy. The process of restructuring source code without changing its behaviour is called *refactoring*.

One of the benefits of refactoring the `UglyComparison` program into the `GoodComparison` program is that we now also have the classes `TextUserInterface` and `Employee`, which can be used without modification in other programs. This is a result of giving the classes clear responsibilities.

## Assigning the responsibility for calculating salaries

The task of calculating the weekly salary has now been split across three methods, where each method has been given a clearly defined sub-task:



- The method `overtimeHourlyRate()` at (2) calculates the compensation for each hour exceeding the normal weekly working hours.
- The method `overtimeHours()` at (3) calculates the number of hours overtime an employee has worked during a week.
- The method `computeSalary()` at (4) still calculates the weekly salary, but has now been greatly simplified by delegating much of the work to the other two methods.

In addition to being used for calculating the weekly salary, the new methods can be useful for other tasks as well. This versatility indicates that the responsibility has been sensibly split across the methods. For example, to denote a rule that an employee can only work 200 hours of overtime during a year, one could write:

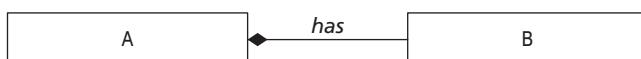
```
accumulatedOvertimeThisYear += overtimeHours(hoursThisWeek);
boolean overtimeLimitExceeded = accumulatedOvertimeThisYear > 200;
```

## 8.3 Relationships between objects

### Fields with reference values

If an object A has a field variable that stores the reference of another object B, then the code in object A can use object B through the field variable (see Figure 8.2). Object A can also pass the reference value to other objects that want to use object B.

**FIGURE 8.2 Field variable storing reference value**



Storing reference values can be illustrated by the following analogy: I have the phone number of a taxi service (object B) in my wallet (object A), so that I can call for a taxi at any time. I can also give the phone number to others, so that they can order a taxi as well. Here the phone number acts as a reference value to the taxi service object. Regardless of how many places the phone number is stored, there is still only one taxi service.

Each employee may have a manager. Suppose we want to store this relationship in the `Employee` object. The manager is also an employee, and can in turn have another manager. This relationship has been established by declaring the following field in the `Employee` class shown in Program 8.4:

```
Employee manager; // (1)
```

When we create an `Employee` object, the `manager` field at (1) is initialized to `null`. To establish a *manager-subordinate* relationship between two employees, the `manager` field of the subordinate object can be set to refer to the manager object. Managers are also employees, and are therefore also represented by objects of the `Employee` class.



## Objects that communicate

Program 8.6 shows the use of references between objects. A company object and a user interface object are created in the `main()` method of the `Managers` class. The company object is then told to report the income difference between some of the employees in the company. The company object is defined by the `SmallCompany` class, and has fields that refer to four different `Employee` objects. The constructor of `SmallCompany` creates *manager-subordinate* relationship between the employees by setting the `manager` reference in some of the employees. Figure 8.3 shows the objects that are created by the program and the links between them.

### PROGRAM 8.6 Use of the manager association

```
class SmallCompany {
 Employee president;
 Employee cfo; // Chief financial officer
 Employee cto; // Chief technology officer
 Employee engineer;

 TextUserInterface ui;

 SmallCompany(TextUserInterface ui) {
 this.ui = ui;

 president = new Employee("Norman", "Knuth", 37, Gender.MALE);
 cfo = new Employee("Steven", "Feynman", 30, Gender.MALE);
 cto = new Employee("Donald", "Borlaug", 32, Gender.MALE);
 engineer = new Employee("Richard", "Levy", 24, Gender.MALE);

 cfo.manager = president;
 cto.manager = president;
 engineer.manager = cto;
 }

 void reportIncomeDifferenceWithManager(Employee anEmployee) {
 Employee manager = anEmployee.manager;
 assert manager != null;
 double income_difference =
 manager.computeSalary(Employee.NORMAL_WORKWEEK)
 - anEmployee.computeSalary(Employee.NORMAL_WORKWEEK);
 ui.reportIncomeDifference(anEmployee, manager, income_difference);
 }
}

public class Managers {
 public static void main(String[] args) {
 TextUserInterface ui = new TextUserInterface(); // (1)
 SmallCompany company = new SmallCompany(ui); // (2)
 company.reportIncomeDifferenceWithManager(company.cfo); // (3)
 company.reportIncomeDifferenceWithManager(company.engineer); // (4)
```



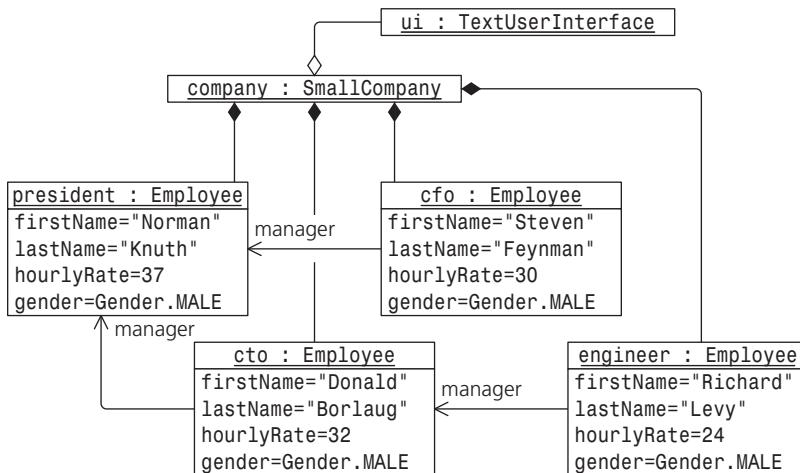
```
 }
}
```

Program output:

Steven Feynman earns \$262.50 less than the manager Norman Knuth during a normal work week

Richard Levy earns \$300.00 less than the manager Donald Borlaug during a normal work week

**FIGURE 8.3** References between objects



The method `reportIncomeDifferenceWithManager()` in the class `SmallCompany` finds the manager of an employee and reports the difference in income between them for a normal working week. The report is delivered to the user through a user interface that is linked to the `SmallCompany` object. The `main()` method in the `Managers` class is responsible for constructing the user interface object at (1), and passing the reference value of the user interface to the `company` object created at (2). The `SmallCompany` constructor stores the reference value of the user interface object in the `ui` field variable, thus establishing the relationship between the `company` object and the user interface object.

## Object ownership

To manipulate an object we need its reference value. If the desired reference value is not available locally, it must be obtained from another object or class. In Program 8.6 the `main()` method obtains the reference values of `Employee` objects from the `SmallCompany` object at (3) and (4). The `SmallCompany` object has the main responsibility of maintaining references to the `Employee` objects. This is a case in which the relationship denotes *ownership*: the `SmallCompany` object owns the `Employee` objects referenced by its field variables.

The *association* between an employee and their manager is *not* an ownership relationship. It is obvious that the subordinate `Employee` object does not own the `Employee` object representing the manager. If the manager had references to all their subordinates, then we



could perhaps call it ownership. But that would be strange, since if a manager is removed from an organisation, it does not automatically follow that all the subordinates are also removed.

Figure H.2 on page 772 shows the notation used for different kinds of association relationships. All the UML diagrams later in this chapter use this notation.

Generally, the owner of an object has the main responsibility of maintaining a reference to the object. A Java object will remain available as long as at least one reference to the object is maintained. If all references are lost, there is no way of getting hold of the object. In the following program code, the sole reference to an `Employee` object is removed:

```
Employee salesperson = new Employee("Maria", "Jensen", 25.0,
 Gender.FEMALE);
salesperson = null; // Removes the only reference to the Employee object
```

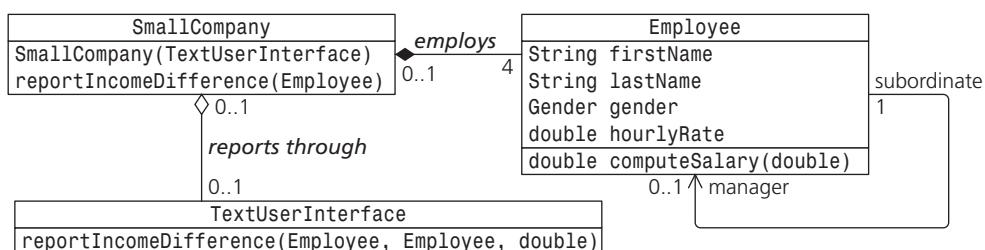
Objects that are no longer used may be garbage collected by the JVM.

## One-to-one and one-to-many associations

The `SmallCompany` class has four `Employee` reference fields, which means that for each `SmallCompany` object there will be four `Employee` objects. This relationship is called a *one-to-many association*.

Figure 8.4 shows the associations between the classes `SmallCompany`, `TextUserInterface` and `Employee`. This UML diagram shows the relative number of objects that take part at each end of an association. In the *employs* association between the classes `SmallCompany` and `Employee` there are four `Employee` objects for each `SmallCompany` object. Each `Employee` object may belong to a `SmallCompany`, but this is not mandatory (indicated by `0..1`). It is always possible to construct an `Employee` object without linking it to a `SmallCompany` object. Table H.1 on page 772 shows typical *multiplicity* values that can occur in associations between classes.

**FIGURE 8.4** Associations between classes



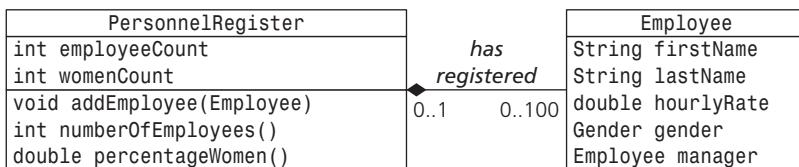
In Program 8.6 exactly four employees are referenced from one `SmallCompany` object. In the real world companies do not employ a fixed number of people. It is possible to make an association in which a variable number of employees can take part. Program 8.7 defines a `PersonnelRegister` class that maintains such an association. The responsibility of gathering gender statistics, which we decided did not belong to the `Employee` class, has also been given to the new `PersonnelRegister` class. Since the `PersonnelRegister` class, unlike the



`Employee` class, manages several `Employee` objects, it makes sense to place this responsibility here. Note that static fields are no longer required now that the gender statistics are stored in `PersonnelRegister` objects.

The class `PersonnelRegister` uses an array of employees to keep track of who is employed, (1). Figure 8.5 shows how the association between `PersonnelRegister` and `Employee` can be denoted in a UML diagram.

**FIGURE 8.5** Association between `PersonnelRegister` and `Employee` classes



**PROGRAM 8.7** One-to-many association in company with personnel register

```
class PersonnelRegister {
 Employee[] allEmployees;
 int employeeCount;
 int womenCount;

 PersonnelRegister() {
 allEmployees = new Employee[100];
 } // (1)

 void addEmployee(Employee newEmployee) {
 allEmployees[employeeCount] = newEmployee;
 employeeCount++;
 if (newEmployee.gender == Gender.FEMALE) {
 womenCount++;
 }
 }

 int numberofEmployees() {
 return employeeCount;
 }

 double percentageWomen() {
 return 100.0 * womenCount / employeeCount;
 }
}

public class NewCompany {
 public static void main(String[] args) {
 TextUserInterface ui = new TextUserInterface();
 PersonnelRegister register = new PersonnelRegister();
```



```

do {
 Employee newEmployee = ui.inputEmployee("employee", true);
 register.addEmployee(newEmployee);
} while (ui.requestYesNoAnswer("Are there any more employees?"));

ui.reportPercentageWomen(register.numberofEmployees(),
 register.percentageWomen());
}
}

```

Excerpt from running the program:

Input information about employee:

- First name: Amy
- Last name: Lightman
- Hourly rate: 29
- Is this a woman? (y/n): y
- Are there any more employees? (y/n): y

Input information about employee:

- First name: Dimitri

...

- Are there any more employees? (y/n): n

The company has 5 employees, 40.0% of which are women.

---

## Suggestions for further development

The structure of Program 8.7 and the classes it uses are now fairly well designed, but there is room for improvement:

- The program fails with an `InputMismatchException` if the hourly rate provided by the user is not a valid number. The program should use the `hasNextDouble()` method from the `Scanner` class to check for invalid numbers, and give the user another chance to enter a valid number.
- The answers given to the Yes/No questions are not examined properly. The program assumes the answer is No as long as the exact answer given was not "y". The program should verify the validity of the answer that was given.
- Even though the `PersonnelRegister` class can manage a variable number of employees, there is still an upper limit of one hundred employees. Since the class does not check whether this limit has been reached, the program will fail when the number of employees exceed one hundred.
- The `PersonnelRegister` class does not provide any way of removing employees.

## 8.4 Method overloading



Program 7.8 on page 190 defines the `EmployeeV7` class, which has three constructors with the same name but with different parameter lists. This means that the constructors have

different signatures, and that the constructor name “EmployeeV7” is *overloaded*. Which constructor is used to construct a particular object depends on the actual parameter list that is used with the `new` operator when creating the object.

Just as it is possible to overload constructors, it is also possible to overload method names. Within a class it is possible to define several methods with the same name, as long as they have different parameter lists. When the compiler encounters a method call using an overloaded method name in the source code, it determines to which method declaration the call actually refers. The compiler will choose the method declaration that matches the signature of the method call. This decision is made at compile time, and determines which method body will actually be executed at runtime.

We want to look up employees in the register and retrieve reference values of `Employee` objects. We want to be able to look up an employee, either by the index value, or by their name. Let’s define two new methods, both called “`getEmployee`” in the `PersonnelRegister` from Program 8.7. This is allowed, since the method declarations will have different parameter types.

```
Employee getEmployee(int employeeIdNumber) {
 return allEmployees[employeeIdNumber];
}

Employee getEmployee(String firstName, String lastName) {
 for (int i = 0; i < employeeCount; i++) {
 Employee anEmployee = allEmployees[i];
 if (firstName.equals(anEmployee.firstName) &&
 lastName.equals(anEmployee.lastName))
 return anEmployee;
 }
 return null;
}
```

The signature of the first method is `getEmployee(int)` and the signature of the second method is `getEmployee(String, String)`. When the compiler translates a method call, it looks at the method name and the types of the actual parameters to find the method that has the corresponding signature.

```
getEmployee(0); // Calls getEmployee(int)
getEmployee("Phil", "Green"); // Calls getEmployee(String, String)
```

## 8.5 Documenting source code

To help maintenance and further development, it is important that programs are easy to understand. Good documentation makes a program easier to understand.

An important form of documentation is comments written in the source code that explain how a program works. Other types of documentation include user manuals, UML diagrams and algorithm descriptions. This book focuses on documenting source code using comments and documenting algorithms using pseudocode.



## Multi-line comments

All the code examples so far have used single line comments starting with “//”. To write a comment spanning multiple lines, the “//” prefix can be used on multiple consecutive lines, like this:

```
// This text is an example of a comment in the Java programming
// language that spans multiple source code lines. Each line needs
// to be prefixed when using this syntax.
```

The Java language also provides an alternate syntax for writing multi-line comments:

```
/* This text is an example of a comment in the Java programming
language that spans multiple source code lines. Only the
start and end of the comment needs to be marked when using this
syntax. */
```

Everything from the character sequence “/\*” up to the character sequence “\*/” is ignored by the compiler.

## Documenting classes and members

The Java language provides a standardised way of documenting fields, methods and classes. To document one of these language constructs, a special type of comment is placed immediately preceding the construct. These special comments start with the character sequence “/\*\*” and end with the character sequence “\*/”. Notice the extra asterisk (\*) at the start of the comment compared to normal multi-line comments. Here is an example of a comment that describes a field variable:

```
/**
 * The price of the product, not including tax.
 */
double price;
```

These special comments are called *Javadoc comments*, named after the javadoc tool in the JDK. This tool can generate documentation that collects information written in these special comments, as well as extra information that the tool is able to extract by analysing the source code. The tool will automatically extract type descriptions of classes, fields, methods and method parameters. The javadoc tool recognizes specific markup tags within the Javadoc comments and uses these to present the information in a structured layout. Table 8.1 show the most common markup tags used for documenting methods.

TABLE 8.1 Javadoc markup tags for methods

| Markup tags                                     | Purpose                                   |
|-------------------------------------------------|-------------------------------------------|
| @param <i>parameter-name</i> <i>description</i> | Describes a formal parameter of a method. |
| @return <i>description</i>                      | Describes the return value from a method. |

## Hiding internal methods and fields

Some methods and fields are only meant for internal use in the class in which they are defined. The Java language provides a way to mark these methods and fields as unsuitable for use by other classes. By adding the modifier keyword `private` to the declaration of a class member, the compiler ensures that the member is not accessible, and will refuse to compile any code outside the class that tries to use the member. Code inside the class will still be able to use the member.

The modifier keyword `public` can be used in declarations of members to indicate that the member should not be hidden from external code. By default, the `javadoc` tool will only generate documentation for members that are declared `public`.

## A fully-documented Java class

Program 8.8 shows the fully-documented source code for the `PersonnelRegister` class. Figure 8.6 on page 229 shows the HyperText Markup Language (HTML) document generated by the `javadoc` tool from the source code for the `PersonnelRegister` class in Program 8.8.

The fields `allEmployees`, `employeeCount` and `womenCount`, and the method `addEmployeeToStatistics()` are declared `private` and are not included in the generated documentation. The methods `addEmployee()`, `getEmployee()`, `numberOfEmployees()` and `percentageWomen()` are declared `public` and therefore included in the generated documentation.

### PROGRAM 8.8 The complete and documented `PersonnelRegister` class

```
/**
 * A personnel register maintains a collection of employees. A company
 * can use a personnel register to keep track of who is employed in the
 * company.
 *
 * The personnel register collects gender statistics for all the
 * employees that are added to the register. These statistics reflect
 * the gender of each employee at the time the employee is added. The
 * statistics will not reflect any gender change done later.
 *
 * The personnel register has some major limitations: the register will
 * stop working if the number of employees exceeds 100, and there is no
 * way to remove employees from the register, once they are added.
 */
public class PersonnelRegister {
 static final int MAXIMUM_NUMBER_OF_EMPLOYEES = 100;
 private Employee[] allEmployees;
 private int employeeCount;
 private int womenCount;

 public PersonnelRegister() {
 allEmployees = new Employee[MAXIMUM_NUMBER_OF_EMPLOYEES];
 }
}
```



```

/**
 * Stores another employee in the register and updates the gender
 * statistics. Only an employee with known gender can be added
 *
 * @return The ID number assigned to the employee that was added.
 */
public int addEmployee(Employee newEmployee) {
 assert newEmployee.gender != Gender.UNKNOWN
 : "Gender must be known before registration.";
 assert employeeCount < allEmployees.length;

 int employeeIdNumber = employeeCount;
 allEmployees[employeeIdNumber] = newEmployee;
 addEmployeeToStatistics(newEmployee);
 return employeeIdNumber;
}

private void addEmployeeToStatistics(Employee newEmployee) {
 employeeCount++;
 if (newEmployee.gender == Gender.FEMALE) {
 womenCount++;
 }
}

/**
 * Retrieves a stored employee with a specific ID number.
 *
 * @param employeeIdNumber The ID number of the employ to retrieve. A
 * valid ID number must be non-negative and less than the number
 * returned by numberOfEmployees().
 */
public Employee getEmployee(int employeeIdNumber) {
 return allEmployees[employeeIdNumber];
}

/**
 * Retrieves a stored employee with a specific name.
 *
 * @param firstName The first name of the employee
 * @param lastName The last name of the employee
 * @return the employee, or null if no employee with that name was
 * found.
 */
public Employee getEmployee(String firstName, String lastName) {
 for (int i = 0; i < employeeCount; i++) {
 Employee anEmployee = allEmployees[i];
 if (firstName.equals(anEmployee.firstName) &&
 lastName.equals(anEmployee.lastName)) {
 return anEmployee;
 }
}

```



```

 }
 }
 return null;
}

/**
 * @return The total number of employees that have been added to
 * this register.
 */
public int numberOfEmployees() {
 return employeeCount;
}

/**
 * @return The percentage of women employees in the register. Since
 * the register only keeps track of men and women added to the
 * register, it is possible to calculate the percentage of men
 * using this formula: 100 - women_percentage.
 */
public double percentageWomen() {
 return 100.0 * womenCount / employeeCount;
}
}

```

---

The syntax for generating documentation using the Javadoc tool from the command line is:

```
> javadoc PersonnelRegister.java
```

This generates HTML files that contain the documentation for the program based on the comments in the source code file `PersonnelRegister.java`. Just like the `javac` compiler, the `javadoc` tool requires the exact name of the source code file. The documentation that is shipped with the JDK describes other Javadoc comment markup tags and command line options that the `javadoc` tool understands.

## How to document programs

Many code examples in this book are documented line by line, for the sole purpose of explaining the language constructs of Java. Under normal circumstances a programmer is already familiar with the language constructs, and the documentation should therefore focus on explaining aspects of the implementation.

### BEST PRACTICES

Give priority to the non-obvious aspects of a program when documenting it. Don't spend time documenting single lines of code, unless of course a single line of code is difficult to understand. Refactor the code rather than providing a lengthy explanation.



Outside the context of a book on programming, a comment such as the following is meaningless and should be avoided:

```
t = t*2 + a; // doubles t and adds a
```

The calculation described by this comment is self-explanatory. What is not explained, is the purpose of the calculation. Choosing meaningful variable names helps to make the purpose clear.

**FIGURE 8.6** Generated Javadoc documentation

The screenshot shows a Mozilla Firefox browser window with the title "PersonnelRegister - Mozilla Firefox". The address bar shows the URL "PersonnelRegister". The page content is the generated Javadoc documentation for the `PersonnelRegister` class. The class hierarchy is shown as `java.lang.Object` → `PersonnelRegister`. The class description states: "A personnel register maintains a collection of employees. A company can use a personnel register to keep track of who is employed in the company. The personnel register collects gender statistics for all the employees that are added to the register. These statistics reflect the gender of each employee at the time the employee is added. The statistics will not reflect any gender change done later. The personnel register has some major limitations: the register will stop working if the number of employees exceeds 100, and there is no way to remove employees from the register, once they are added." The "Constructor Summary" section contains the constructor `PersonnelRegister()`. The "Method Summary" section lists the following methods:

| Type     | Name                                                                            | Description                                                                |
|----------|---------------------------------------------------------------------------------|----------------------------------------------------------------------------|
| int      | <code>addEmployee(Employee newEmployee)</code>                                  | Stores another employee in the register and updates the gender statistics. |
| Employee | <code>getEmployee(int employeeIdNumber)</code>                                  | Retrieves a stored employee with a specific ID number.                     |
| Employee | <code>getEmployee(java.lang.String firstName, java.lang.String lastName)</code> | Retrieves a stored employee with a specific name.                          |
| int      | <code>numberOfEmployees()</code>                                                |                                                                            |
| double   | <code>percentageWomen()</code>                                                  |                                                                            |



## BEST PRACTICES

When documenting an entire program, focus on how the program is used, how the program is structured, which classes and objects cooperate to implement the functionality of the program.

When documenting classes, focus on the responsibilities classes have, what roles the objects play, and how they all cooperate with each other.

Each method should have a description of what it does, what the parameters represent and what result is returned, if it is not obvious from the declaration.

Several limitations and improvements for Program 8.7 were discussed at the end of Section 8.3. Pay attention to such limitations when documenting a program. If a program does *not* work under certain circumstances, this should be documented. The Javadoc comment at the beginning of Program 8.8 clearly states that the class only works as long as there are no more than a hundred employees. The documentation for a method should explain what assumptions the method makes about its parameter values.

Assertions can also be used to explain to the reader what assumptions the code makes:

```
assert newEmployee.gender != Gender.UNKNOWN
 : "Gender must be known before registration.";
assert employeeCount < allEmployees.length;
```

These assertions from the `addEmployee()` method explain that the gender of the employee that is being added needs to be known, and that it is not permitted to add any more employees if the register is already full. Another benefit of using assertions is that the assumptions they document can be checked automatically at runtime. Trying to run the following code, for example, will cause the first assertion above to fail:

```
PersonnelRegister register = new PersonnelRegister();
register.addEmployee(new Employee("Kris", "Epicene", Gender.UNKNOWN));
```



## 8.6 Review questions

1. The properties and behaviour defined for a class determine what \_\_\_\_\_ the class has, and what \_\_\_\_\_ objects of this class can fulfil.
2. Given a user-defined class Egg and a method that has the signature void boilEgg(Egg), which of these statements are true?
  - a The boilEgg() method receives a copy of the Egg object that is passed as parameter.
  - b The boilEgg() method receives an Egg object as a parameter.
  - c The boilEgg() method receives a reference value as parameter. It refers to an Egg object.
  - d The boilEgg() method may change the state of the Egg object, so that the Egg object may have a different state after the method call completes.
3. In Java an object can ask another object to do something by \_\_\_\_\_ a \_\_\_\_\_ of the other object.
4. An association between two classes can be established by one or both of the classes declaring a \_\_\_\_\_ in order to store a \_\_\_\_\_ of an object of the other class.
5. An association in which a Car object owns four Tire objects is called a \_\_\_\_\_-to-\_\_\_\_\_ association.
6. If a method name is overloaded, when will the decision on which method body to execute be made?
  - a When compiling the method call statement.
  - b When the program is started.
  - c When the method call is executed.

## 8.7 Programming exercises

1. In New York State there is a sales tax of 8 cents per dollar. The tax is not charged for certain products, such as fruits, vegetables, meat and dairy products. Write a Product class that represents products that can be sold. A product should have a name, a base price and should be marked whether or not it is exempt from sales tax. A product object should provide an easy way to determine how much a customer must pay to buy the product.
2. Write a class that represents the selection of products a grocery store provides. Name this class “GroceryStoreSelection”, and create a one-to-many association between this class and the Product class in Exercise 8.1. The GroceryStoreSelection class should provide methods to add products to the selection and retrieve products by their product name.

- 3.** Write a program that has a text-based menu that allows the user to register new products, and which finds the sale price of products that have already been registered. The program should use the classes from Exercise 8.1 and Exercise 8.2. To register a new product, the user should enter the name of the product, the base price of the product, and whether the product is tax exempt. To find the sale price of the product the user should only need to specify the name of the product.
- 4.** Write a class that represents individual playing cards from a standard deck of Anglo-American cards, and a class that represents a whole deck of fifty-two cards. Each card can have one of thirteen ranks (2–10, Jack, Queen, King and Ace) and one of four suites (Spades, Hearts, Diamonds and Clubs.) What factors should be considered when designing the classes? Consider how the classes will be used in different types of card games, e.g. poker and blackjack.
- 5.** We want to create a program that implements a simple card game for two players, with the following rules:
  - 1** A deck of 52 cards is shuffled.
  - 2** Each player is given a card.
  - 3** Each player gets the opportunity to replace their card with a new card from the deck, if they so choose.
  - 4** The player that has the card with the highest value wins.

In this game cards 2 through 10 are worth their face value, and the Jack, Queen, King and Ace are worth 11, 12, 13 and 14 respectively. The program should communicate with both players via the terminal window. What classes and methods are needed in the program?

Write a complete program that will play a single round of the game. The dialogue between the players and the program should look like this:

```
A simple card game:
Creating deck of cards.
Creating player 1.
Enter name of new player: Ron
Creating player 2.
Enter name of new player: Veronica
Shuffling the deck of cards.
Giving each player a card.
Allowing players to change their card.
Ron, you have the card 10 of Diamonds.
Do you want to change your card, Ron (y/n)? n
Veronica, you have the card 6 of Spades.
Do you want to change your card, Veronica (y/n)? y
Giving Veronica a new card.
Ron has the card 10 of Diamonds.
Veronica has the card Jack of Hearts.
Veronica won.
```





## P A R T    T H R E E

Some Useful Techniques for Building Programs



## Sorting and searching arrays

### LEARNING OBJECTIVES

By the end of this chapter you will understand the following:

- The meaning of natural order for values in a data type.
- The use of relational operators for comparing values of primitive data types.
- How to sort values using the selection sort and insertion sort algorithms.
- Why developing pseudocode prior to coding is useful.
- How to search for a value using linear search and binary search algorithms.
- How to write a method that defines the natural order of objects for your own class.

### INTRODUCTION

Sorting and searching are needed for many purposes, for example ranking the candidates after an election, or when looking up phone numbers in an electronic phone book. This chapter explains a few basic sorting and searching algorithms, and demonstrates their use when data is stored in arrays. The main steps of each algorithm are defined using a structured language called *pseudocode*, which is used to refine the problem at hand into successively smaller parts until each part is simple enough to be implemented in a programming language. Inserting the pseudocode as comments in the source code is used as a technique to help understand how the various steps are implemented, resulting in source code that is well documented.



## 9.1 Natural order

Values of numerical primitive data types have a *natural order*, allowing a *relational operator* to be used to determine the ranking of two values. This means that we are always able to compare two values, say  $a$  and  $b$ , to determine whether  $a$  is smaller than, equal to, or larger than  $b$ . This relationship holds for all numerical primitive data types, i.e. byte, short, int, long, float and double, as well as for characters (of type char). The only primitive data type that does not have a natural order is boolean. While it makes sense to compare two Boolean values for equality, it has no meaning to state that true is smaller than or larger than false.

Many types of objects also have a natural order. We have compared two strings several times in this book and chosen one of them based on their lexicographical order. The String class defines this natural order for strings. Later in this chapter we will look at how a class can define a natural order for its objects.

### Relational operators for primitive data types

Table 9.1 shows relational operators that are predefined for operands of numerical data types and characters. The relational operators always return a boolean value, indicating whether the test succeeded (true) or failed (false).

TABLE 9.1

Relational operators for primitive data types

| Operator | Tests whether                         |
|----------|---------------------------------------|
| $a == b$ | $a$ is equal to $b$ .                 |
| $a < b$  | $a$ is less than $b$ .                |
| $a > b$  | $a$ is greater than $b$ .             |
| $a <= b$ | $a$ is less than or equal to $b$ .    |
| $a >= b$ | $a$ is greater than or equal to $b$ . |

### Understanding relational operators

Program 9.1 shows a simple way to test your understanding of relational operators. It uses assert statements to formulate assumptions as Boolean expressions that should be true. If all the assertions are correct, as in Program 9.1, the program will run without any runtime error messages. If not, the program will terminate and the JVM will print an error message. Remember to turn on the assertions by specifying the “-ea” flag:

```
> java -ea TestRelationalOperators
```

Had the last assertion been incorrect in Program 9.1, for example:

```
assert 'x' < 'p';
```



the JVM would have printed an error message such as the following:

```
Exception in thread "main" java.lang.AssertionError
at TestRelationalOperators.main(TestRelationalOperators.java:17)
```

identifying the source code file (`TestRelationalOperators.java`) and line number (17) where the error occurred. This makes it easy to look up the statement in a text editor and make the necessary modifications to the code.

### PROGRAM 9.1 Using relational operators to compare primitive values

```
// Using assertions to understand relational operators.
public class TestRelationalOperators {
 public static void main(String[] args) {
 // Tests for integers.
 assert 1 == 1;
 assert 1 < 2;
 assert 1 <= 1;
 assert 3 > 2;
 assert 3 >= 3;
 // Tests for decimal numbers.
 assert 1.0 < 1.01;
 assert 1000.0 > 999.99;
 // Test for characters.
 assert 'x' == 'x';
 assert 'x' < 'y';
 assert 'x' <= 'z';
 assert 'x' > 'p';
 System.out.println("All assertions are true.");
 }
}
```

Program output:

All assertions are true.

### Comparing floating-point values

Note that the equality operator `==` should be used with care when comparing floating-point values for equality, as certain floating-point values cannot be represented exactly in computer memory. While for most practical purposes the values `0.333333333333` (with twelve decimals) and `1.0/3.0` can be considered equal, the assertion at (1) in the code below will not succeed:

```
final double ONE_THIRD = 0.333333333333;
double ratio = 1.0/3.0;
assert ratio == ONE_THIRD : // (1)
 "ERROR: " + ratio + " is different from " + ONE_THIRD;
```



Executing this code (with the “-ea” flag) will result in an error message similar to the following:

```
Exception in thread "main" java.lang.AssertionError:
ERROR: 0.3333333333333333 is different from 0.333333333333
at TestDecimalEquality.main(TestDecimalEquality.java:7)
```

Although we could circumvent the problem above by defining the constant `ONE_THIRD` to be the result of the expression `1.0/3.0`, the problem with using the `==` operator to compare floating-point values persists. A better solution is to test whether the first value is within a range of  $\pm \epsilon$  of the second value, where  $\epsilon$  is some small value:

```
...
double epsilon = 1.0E-12; // 0.000000000001
assert Math.abs(ratio - ONE_THIRD) <= epsilon
"ERROR: " + ratio + " is different from " + ONE_THIRD;
```

## 9.2 Selection sort

Sorting means ordering a set of primitive data values or objects in ascending or descending order based on the natural order defined by their data type. How can we sort the numbers drawn in this week’s national lottery, or all members in a local sports club? To answer this question, we’ll look first at sorting a set of integer values stored in an array.

The *selection sort* algorithm works by finding the smallest (or largest) value in the unsorted part of the array, and swapping this value with the value at the beginning of the unsorted part. In this way, the beginning of the array gradually becomes sorted and the remaining unsorted part contains one less element to sort for each pass over the array. This procedure is repeated until there is only one element left in the unsorted part. The values will be sorted in ascending order if the smallest value is picked in each pass, and in descending order if the largest value is picked in each pass.

### Sorting an array of integers

Consider the following array:

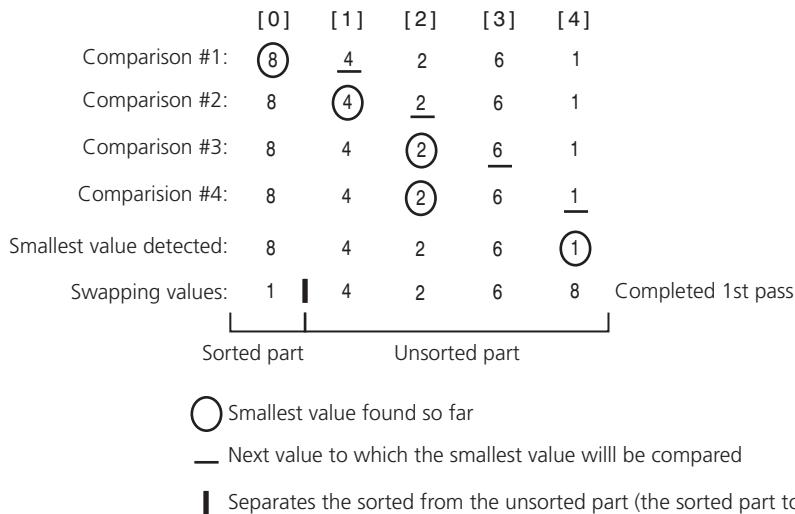
```
int[] numbers = { 8, 4, 2, 6, 1 };
```

We want to sort this array in ascending order using the selection sort algorithm. Figure 9.1 illustrates what happens during the first pass. The array is searched for the *smallest* number, which is then swapped with the first element in the array. After the first pass, we have a sorted part of the array consisting of *a single value* placed correctly in the first element of the array.

In each of the subsequent passes, the smallest of the remaining values will be swapped with the value in the first position of the unsorted part of the array. In this way the sorted part of the array expands by one element placed correctly per pass, yielding a sorted array after the final pass. Figure 9.2 shows the array after each pass. Note that in Figure 9.2 the smallest value and the value in the first position of the unsorted part are identical in passes three and four.



**FIGURE 9.1** First pass of an integer array during selection sort



**FIGURE 9.2** The main steps of selection sort in an array of integers

|          | [0] | [1] | [2] | [3] | [4]              |
|----------|-----|-----|-----|-----|------------------|
| Start:   | [ 8 | 4   | 2   | 6   | 1 ]              |
| Pass #1: | 1   | [ 4 | 2   | 6   | 8 ]              |
| Pass #2: | 1   | 2   | [ 4 | 6   | 8 ]              |
| Pass #3: | 1   | 2   | 4   | [ 6 | 8 ]              |
| Pass #4: | 1   | 2   | 4   | 6   | 8      Completed |

[ ... ] Unsorted part that is searched for the smallest value  
 — Identifies values that were swapped after each pass

## Pseudocode for the selection sort

We can now formalise the steps in the selection sort algorithm using pseudocode:

For each pass:

Find the smallest value in the unsorted part of the array

Swap the smallest value with the first element of the unsorted part

This algorithm will sort the array in *ascending* order. If we need to sort the array in *descending* order, the algorithm must find the *largest* value in each pass, and swap this value with the value in the first position of the unsorted part of the array.

Pseudocode emphasises readability and simplicity, but cannot be compiled into an executable program in the way that Java source code can be. You can think of pseudocode as a recipe; it describes what steps are needed to prepare, say, Coq au vin, but it does not actually cook it for you. Still, the recipe is important, because chances are you will not end up with the desired results without it. Similarly, pseudocode helps you think through the



problem at hand, and identifies the steps that are needed to solve it, before writing any code. The pseudocode can also be copied into the source code as comments, making it easier to relate the steps defined in the algorithm to the code that actually implements them.

Pseudocode is commonly used to develop and document algorithms, and is written in natural language. However, some statements begin with keywords that are used to indicate flow of control, such as “For each” to indicate the start of a repetition (i.e. loop), or “If” to indicate selection among alternative actions (i.e. starting a selection statement). In addition, indentation is used to show the grouping of operations, where statements at the same indentation level belong to the same block (i.e. a compound statement).

## Coding the selection sort

Program 9.2 shows the implementation of the selection sort algorithm described above. The program uses two nested loops, at (1) and (2). The loop variable `next` in the outer loop runs from index 0 to index `numbers.length-1`. The loop variable `current` in the inner loop runs through the unsorted part of the array, i.e. from index `next+1` to index `numbers.length`. Swapping the smallest value found in each pass with the first element in the unsorted part of the array is done at (3), each time the inner loop has completed (cf. Figure 9.2).

### PROGRAM 9.2    Sorting by selection in an array of integers

```
// Utility class for sorting and searching arrays of integers.
class Utility {
 static void sortBySelection(int[] numbers) {
 // For each pass:
 for (int next=0; next < numbers.length-1; next++) { // (1) outer loop
 // Find the smallest value in the unsorted part of the array
 // Note that we need to find the position of the smallest value
 // in order to swap values later.
 int smallest = next;
 for (int current = next+1;
 current < numbers.length;
 current++) { // (2) inner loop
 if (numbers[current] < numbers[smallest]) {
 smallest = current;
 }
 }

 assert smallest > next && smallest < numbers.length :
 "Index for smallest value " + smallest +
 " should have been between " + (next+1) +
 " and " + numbers.length;

 // Swap the smallest value with the first element
 // of the unsorted part
 int temp = numbers[smallest]; // (3)
 }
 }
}
```

```

 numbers[smallest] = numbers[next];
 numbers[next] = temp;

 printArray("Pass #" + (next+1) + ": ", numbers);
 }
}

static void printArray(String prompt, int[] array) {
 System.out.print(prompt);
 for (int value : array) {
 System.out.print(value + " ");
 }
 System.out.println();
}
}

// Sorting an array using the selection sort algorithm.
public class SortingBySelection {
 public static void main(String[] args) {
 int[] numbers = { 8, 4, 2, 6, 1 };
 Utility.printArray("Unsorted array: ", numbers);
 Utility.sortBySelection(numbers);
 Utility.printArray("Array sorted by selection sort: ", numbers);
 }
}

```

Program output:

```

Unsorted array: 8 4 2 6 1
Pass #1: 1 4 2 6 8
Pass #2: 1 2 4 6 8
Pass #3: 1 2 4 6 8
Pass #4: 1 2 4 6 8
Array sorted by selection sort: 1 2 4 6 8

```

## Analysing the amount of work required by a selection sort

Let's take a step back and review the amount of work involved in sorting an array using selection sort. For an array of  $n$  elements, the outer loop in Program 9.2 is always executed  $n-1$  times, placing one element in the correct position at each pass. For each pass, the algorithm compares the current element with all elements left in the unsorted part of the array. The number of comparisons made will be  $n-1$  in the first pass,  $n-2$  in the second, and so on, until only *one* comparison is made in the last pass. Even if the number of comparisons are reduced by one for each pass, the total number will be proportional to the number of elements in the array. Thus, the amount of work needed by this algorithm will be proportional to  $n^2$ . The algorithm is therefore said to be of order  $n^2$ , denoted by  $O(n^2)$ .





## BEST PRACTICES

Use pseudocode to divide complex operations into smaller, manageable operations. Refine the operations until they are easier to understand and implement. Include the pseudocode as comments in your source code. This approach will document how your solution works, and help in debugging your implementation.

### 9.3 Insertion sort

In an array we can define a conceptual *subarray*, which consists of one or more consecutive elements inside an array. A subarray can be sorted even if the entire array is not sorted.

Let's assume we have a sorted subarray. To place a new value in the correct position in the subarray, we first shift the values of all elements that are larger than the new value one position towards the end of the array. We then place the new value in the correct position, resulting in a subarray that is still sorted and which now contains one more element. This procedure is repeated until all values have been inserted at their correct positions. Not surprisingly, this algorithm is called an *insertion sort*.

#### Sorting an array of integers using an insertion sort

Let's return to the array we sorted in the previous section:

```
int[] numbers = { 8, 4, 2, 6, 1 };
```

We now want to sort it using the insertion sort algorithm described above. Figure 9.3 illustrates how the insertion sort algorithm will successively expand the sorted subarray until the whole array is sorted.

#### Pseudocode for the insertion sort

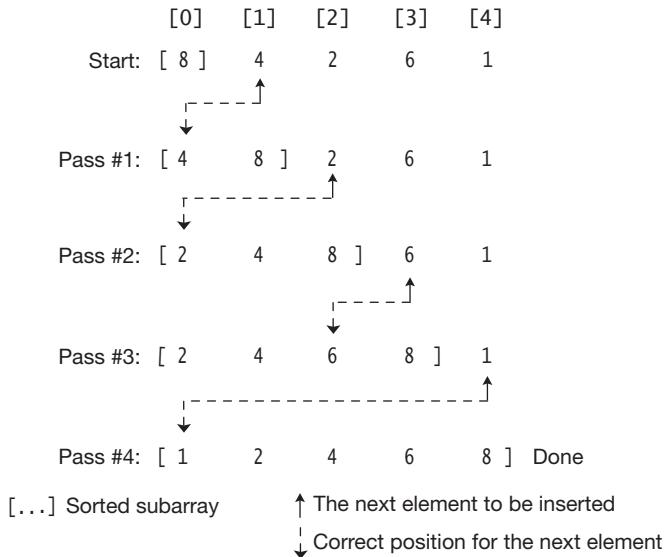
Sorting by insertion can be described by the following pseudocode:

```
For each pass:
 Store the next element to be inserted in an auxiliary variable
 Iterate backwards through the sorted subarray
 If the current element is larger than the next element
 Shift the current element value one position towards the end of the array
 Insert the next element in the vacant position in the array
```

This will sort the array in ascending order. Writing pseudocode for the insertion sort algorithm that sorts the array elements in descending order is left as an exercise.



**FIGURE 9.3** The main steps of insertion sort



## Coding the insertion sort

Program 9.3 shows the implementation of sorting by insertion for an array of integers. The program uses two nested loops, at (1) and (2). The loop variable `next` in the outer loop runs from index 1 to index `numbers.length-1`. In the inner loop, which runs backwards, the element values are shifted towards the end of the array until the correct position for the next element is found (cf. Figure 9.3).

**PROGRAM 9.3** Sorting by insertion in an array of integers

```
// Utility class for sorting and searching arrays of integers.
class Utility {
 static void sortByInsertion(int[] numbers) {
 // For each pass:
 for (int next = 1; next < numbers.length; next++) { // (1) outer loop
 // Store the next element to be inserted in an auxilliary variable
 int temp = numbers[next];
 // Pass through the sorted subarray backwards
 // If the current element is larger than the next element
 // Shift the current element one position towards the end of the array
 int current;
 for (current = next;
 current > 0 && temp < numbers[current-1];
 current--) { // (2) inner loop
 numbers[current] = numbers[current-1]; // (3) shift backwards
 }
 assert current == 0 || numbers[current] < temp :
 }
 }
}
```



```

 "Index for insertion " + current +
 " should have been between 0 and " + (numbers.length-1);

 // Insert the next element in the vacant position in the array
 numbers[current] = temp; // (4)

 printArray("Pass #" + next + ": ", numbers);
}
}

// other methods...
}

// Sorting an array using the insertion sort algorithm.
public class SortingByInsertion {
 public static void main(String[] args) {
 int[] numbers = { 8, 4, 2, 6, 1 };
 Utility.printArray("Unsorted array: ", numbers);
 Utility.sortByInsertion(numbers);
 Utility.printArray("Array sorted by insertion sort: ", numbers);
 }
}

```

Program output:

```

Unsorted array: 8 4 2 6 1
Pass #1: 4 8 2 6 1
Pass #2: 2 4 8 6 1
Pass #3: 2 4 6 8 1
Pass #4: 1 2 4 6 8
Array sorted by insertion sort: 1 2 4 6 8

```

---

### Analysing the amount of work required by an insertion sort

The insertion sort algorithm will pass through the array  $n-1$  times, where  $n$  is the number of elements in the array. In the first pass, up to  $n-2$  comparisons and element value shifts are made. In the second pass, up to  $n-3$  comparisons and element value shifts are made, and so on, until only one comparison and at the most one element value shift is made in the last pass. Assuming that the array is not partially sorted when the algorithm starts, the maximum number of comparisons and element value shifts are found by adding  $(n-2) + (n-3) + \dots + 2 + 1$ . This amounts to a number proportional to  $n$ . Even if the array is sorted beforehand, every comparison still has to be performed. Thus, insertion sort also requires an amount of work proportional to  $n^2$ , and is said to be of order  $O(n^2)$ .



## 9.4 Sorting arrays of objects

### Comparing objects: the Comparable interface

In Chapter 4 we compared objects for reference equality using the `==` operator, and used the `equals()` method to compare objects for value equality. This allowed us to compare two objects for equality only. In Java, the `java.lang.Comparable` interface defines the method needed for ranking objects according to their natural order. This interface specifies a single method called `compareTo`.

For two objects `t1` and `t2` of class `T`, the method call `t1.compareTo(t2)` returns a value:

- `== 0` if the objects are equal according to the same criteria as for the `equals()` method.
- `< 0` if `t1` is smaller than `t2`, i.e. `t1` comes before `t2` in the natural order for class `T`.
- `> 0` if `t1` is larger than `t2`, i.e. `t1` comes after `t2` in the natural order for class `T`.

Every class that needs to represent a data type with a natural order must adhere to the `Comparable` interface by implementing a `compareTo()` method in accordance with the natural order of its objects. The Javadoc documentation for this interface states that the natural order for a class `T` should be consistent with its implementation of the `equals()` method. The expression `(t1.compareTo(t2) == 0)` should therefore yield the same boolean value as `t1.equals(t2)` for all possible objects `t1` and `t2`.

Many classes in the Java standard library implement the `Comparable` interface. We can, for example, compare two `String` objects by calling the `compareTo()` method:

```
String name1 = "Smith";
String name2 = "Jones";
int status = name1.compareTo(name2);
if (status == 0) {
 System.out.println("Participants have the same last name.");
} else if (status < 0) {
 System.out.println(name1 + " comes before " + name2);
} else {
 System.out.println(name1 + " comes after " + name2);
}
```

The `compareTo()` method in the `String` class compares strings based on the Unicode value of the characters at corresponding positions in the two strings. A sort based on method this will order strings in lexicographical order, which is the commonest way of ordering text strings, for example the names in a telephone directory.

All numerical wrapper classes also implement the `Comparable` interface. The natural order of such objects is given by arithmetic comparison of their values, i.e. using the relational operators predefined for numerical data types. See *Relational operators for primitive data types* on page 236. The simple example in Program 9.4 compares two `Integer` objects with values -10 and 10, respectively. The result is what one would expect.

## PROGRAM 9.4 Comparing integer values wrapped as objects

```
// Testing the natural order for wrapper objects that hold integers.
public class TestWrapperComparison {
 public static void main(String[] args) {
 Integer number1 = -10;
 Integer number2 = 10;
 int status = number1.compareTo(number2);

 if (status == 0) {
 System.out.println("EQUAL Integer objects: " + number1 +
 " and " + number2);
 } else if (status < 0) {
 System.out.println("Integer object " + number1 +
 " is smaller than " + number2);
 } else {
 System.out.println("Integer object " + number1 +
 " is larger than " + number2);
 }
 }
}
```

Program output:

Integer object -10 is smaller than 10

## Implementing the natural order for time objects

Let's implement the natural order for objects of the `Time` class introduced in Chapter 7 (see Program 7.10 on page 195). This class defines the time of day as two integers, `hours` (0–23) and `minutes` (0–59). Objects of the `Time` class can obviously be ordered chronologically. We can use this order as the natural order for `Time` objects.

The `compareTo()` method implemented by the `Time` class must return an integer value:

- < 0 if the time represented by `t1` comes before that of `t2`.
- > 0 if the time represented by `t1` comes after that of `t2`.
- == 0 if the `Time` objects represent the same time of day.

Given two `Time` objects `t1` and `t2`, we can determine their natural order by comparing their respective hours and minutes values. If the hours are different, the hour values determine the order. If the hours are the same, but the minutes are different, the minute values determine the order. If both the hours and minutes are the same, the two `Time` objects represent the same time of the day.



The following pseudocode describes the steps for determining the natural order of two `Time` objects `t1` and `t2`:

```
If the hours are different
 Natural order is determined by the natural order of the hour values
Else if the minutes are different
 Natural order is determined by the natural order of the minute values
Else
 The two Time objects represent the same time of day
```

We need to refine the pseudocode above to determine what value the `compareTo()` method should return. The problem boils down to determining the natural order of two integers, representing either hours or minutes. We note that for two integers `i1` and `i2`, their difference (`i1 - i2`) is positive if `i1` is greater than `i2`, negative if `i1` is less than `i2`, and 0 if `i1` and `i2` are equal. The difference is exactly what the `compareTo()` method needs to return. We can now refine the pseudocode above:

```
If the hours are different
 Return the difference between the hour values
Else if the minutes are different
 Return the difference between the minute values
Else
 Return 0
```

The pseudocode above can be further refined if we take into consideration that the *total number of minutes* (i.e. `hours*60 + minutes`) represented by each time of day are also in chronological order, but this is left as an exercise.

Program 9.5 shows how the revised `Time` class implements the `Comparable` interface. The `implements` clause has to be included in the class declaration, as shown in (1). The `compareTo()` method must have the signature shown in (2):

```
class Time implements Comparable { // (1)

 public int compareTo(Object o2) { // (2)
 // implementation of natural order for Time objects
 }
}
```

The `compareTo()` method is passed a `Time` object which is referenced by the parameter `o2` of type `Object`. The construct (`Time`) at (3) is called a *cast*. It tells the compiler that the `obj2` reference passed as parameter will refer to an object of class `Time` at runtime. We cast the parameter to a `Time` reference `t2` at (3) so that we can access the time values of the `Time` object passed as parameter. The rest of the implementation of the `compareTo()` method is based on the pseudocode developed earlier in this subsection.

Note that both the `toString()` and `compareTo()` methods need to be declared as `public` methods. These visibility constraints are enforced by the `Object` class and the `Comparable` interface respectively.

Program 9.5 tests the `compareTo()` method with `Time` objects that represent different times.

## PROGRAM 9.5 Testing the natural order of Time objects

```

// Time is given as hours (0-23) and minutes (0-59).
class Time implements Comparable { // (1)

 // Fields for the time.
 int hours;
 int minutes;

 /** Constructor */
 Time(int hours, int minutes) {
 assert (0 <= hours && hours <= 23 && 0 <= minutes && minutes <= 59) :
 "Invalid hours and/or minutes";
 this.hours = hours;
 this.minutes = minutes;
 }

 /** String representation of the time, TT:MM */
 public String toString() {
 return String.format("%02d:%02d", hours, minutes);
 }

 /** Comparing Time objects */
 public int compareTo(Object obj2) { // (2)
 Time t2 = (Time) obj2; // (3) Cast required, as we are
 // comparing Time objects
 // Check if the hours are different
 if (this.hours != t2.hours) {
 // Order is determined by which of the hours is greater
 return this.hours - t2.hours;
 }
 // Have the same hours. Check if the minutes are different.
 if (this.minutes != t2.minutes) {
 // Order is determined by which of the minutes is greater
 return this.minutes - t2.minutes;
 }
 // Both hours and minutes have the same values.
 return 0; // Objects represent the same time
 }
}

// Testing the natural order for Time objects.
public class TestTimeComparison {
 public static void main(String[] args) {
 Time time1 = new Time(6, 30); // 06:30
 Time time2 = new Time(15, 5); // 15:05
 Time time3 = new Time(20, 45); // 20:45
 compareTimeObjects(time1, time2);
 compareTimeObjects(time3, time2);
 }
}

```



```

 compareTimeObjects(time1, time1);
 }

 static void compareTimeObjects(Time t1, Time t2) {
 int status = t1.compareTo(t2);
 if (status == 0) {
 System.out.println("EQUAL start and end times " + t1 +
 " - CHECK YOUR WATCH!");
 } else if (status < 0) {
 System.out.println("Started working " + t1 + " before ending " + t2
 + " - OK.");
 } else {
 System.out.println("Ended working " + t1 + " before starting " + t2
 + " - YIKES!");
 }
 }
}

```

Program output:

```

Started working 06:30 before ending 15:05 - OK.
Ended working 20:45 before starting 15:05 - YIKES!
EQUAL start and end times 06:30 - CHECK YOUR WATCH!

```

---

## Sorting arrays of comparable objects

Program 9.6 provides an alternative implementation of the `sortBySelection()` method first introduced in Program 9.2. The new method in the `GenericUtility` class can be used to sort arrays of `Comparable` objects.

The `sortBySelection()` method takes an array of `Comparable` objects, as shown in (1). It uses two nested loops, (2) and (3), as before. The loop variable in the outer loop (`next`) runs from index 0 to index `objects.length-2`, while the loop variable in the inner loop (`current`) runs through the unsorted part of the array, i.e. from index `next+1` to index `objects.length-2`. Elements are compared using the `compareTo()` method at (4). The swap at (5) is done for each pass of the inner loop, using the auxiliary variable `temp` of type `Comparable`.

Program 9.6 defines an array with six `Time` objects, which are `Comparable`, and sorts them using the revised selection sort algorithm, printing the sorted array to the terminal window.

In Program 9.6, the array of `Time` objects is already sorted after the fourth pass. However, the selection sort algorithm has no means of detecting this, and will always perform  $n-1$  passes for an array with  $n$  elements. For each pass, all elements in the unsorted part are compared to find the next value. The amount of work is therefore proportional to  $n^2$ .

We can use our implementation of the `sortBySelection()` method to sort *any* array of `Comparable` objects, not just `Time` objects. The `String` class and the numerical wrapper



## PROGRAM 9.6    Sorting arrays of Comparable objects

```

class GenericUtility {
 static void sortBySelection(Comparable[] objects) { // (1)
 // For each pass:
 for (int next = 0; next < objects.length-1; next++) { // (2) outer loop
 // Find the smallest value in the unsorted part of the array
 int smallest = next;
 for (int current = next+1;
 current < objects.length;
 current++) { // (3) inner loop
 if (objects[current].compareTo(objects[smallest]) < 0) { // (4)
 smallest = current;
 }
 }

 assert smallest > next && smallest < objects.length :
 "Index for smallest value " + smallest +
 " should have been between " + (next+1) +
 " and " + objects.length;

 // Swap the smallest value with the first element of the usorted part
 Comparable temp = objects[smallest]; // (5)
 objects[smallest] = objects[next];
 objects[next] = temp;

 printArray("Pass #" + (next+1) + ": ", objects);
 }
 }

 // other methods...

 static void printArray(String prompt, Comparable[] array) {
 System.out.print(prompt);
 for (Comparable element : array) {
 System.out.print(element + " ");
 }
 System.out.println();
 }
}

// Program that sorts an array of Time objects using selection sort
public class SortingObjects {
 public static void main(String[] args) {
 Time[] times = { new Time(10, 0), // 10:00
 new Time(12, 30), // 12:30

```



```

 new Time(6, 30), // 06:30
 new Time(6, 56), // 06:56
 new Time(1, 44), // 01:44
 new Time(15, 5) // 15:05
 };
GenericUtility.printArray("Unsorted array: ", times);
GenericUtility.sortBySelection(times);
GenericUtility.printArray("Array sorted by selection sort: ", times);
}
}

```

Program output:

```

Unsorted array: 10:00 12:30 06:30 06:56 01:44 15:05
Pass #1: 01:44 12:30 06:30 06:56 10:00 15:05
Pass #2: 01:44 06:30 12:30 06:56 10:00 15:05
Pass #3: 01:44 06:30 06:56 12:30 10:00 15:05
Pass #4: 01:44 06:30 06:56 10:00 12:30 15:05
Pass #5: 01:44 06:30 06:56 10:00 12:30 15:05
Array sorted by selection sort: 01:44 06:30 06:56 10:00 12:30 15:05

```

---

## 9.5 Linear search

We often need to look up a value in an array. If the array is not sorted, we are forced to check every element until the value sought is found, or until we have exhausted all elements in the array. This searching technique is called *linear search*, and the value we want to look up is referred to as the *search key*.

Linear search can be described by the following pseudocode:

```

Repeat while elements are left to compare in the array:
 If current element equals the search key
 Return the index of the current element
 If the search key is not found after searching the entire array
 Return a negative index to indicate that the key was not found

```

Program 9.7 implements the linear search algorithm for an array of integers (type `int`). The method `linearSearch()` returns the index of the key value if it is found in the array. If the given key is not found, an invalid index, `-1`, is returned. The loop at (1) runs through the array until the entire array is searched or the key is found.

Because the array is not sorted in advance, the worst-case scenario is for the search to run through the entire array comparing every element to the key. The amount of work needed in linear search is therefore proportional to the number of elements in the array. The algorithm is therefore said to be of  $O(n)$ , i.e. of order  $n$ .

Note the use of an `assert` statement in Program 9.7 to verify that the algorithm has either exhausted all the elements of the array, or that the key has been found. Methods in later



programming examples in this chapter use the `assert` statement in a similar way to verify that the algorithm behaves as expected.

Note that if *multiple* array elements contain the search key, Program 9.7 only reports the index of the *first* element that has the key. The program can be extended to report indices of all elements containing the key. This extension is left as an exercise.

### PROGRAM 9.7 Linear search in an array of integers

```
// Utility class for sorting and searching arrays of integers.
class Utility {
 static int linearSearch(int[] numbers, int key) {
 boolean found = false;
 int i = 0;
 // Repeat while more elements in array:
 while (i < numbers.length && !found) { // (1)
 // If the current element contains the key:
 // Return the index of the current element
 if (numbers[i] == key) {
 found = true; // Found!
 } else {
 i++; // Move to next element
 }
 }
 assert (i >= numbers.length || found) : "Error in linear search.";
 if (found)
 return i; // The key is in element numbers[i]
 else
 return -1; // The key is not found
 }

 // other methods...
}

// Using linear search
import java.util.Scanner;
public class LinearSearch {
 public static void main(String[] args) {
 int[] numbers = { 18, 15, 5, 2, 12, 17, 21, 3, 6, 25, 22, 1, 10 };
 Utility.printArray("Array (unsorted): ", numbers);

 Scanner keyboard = new Scanner(System.in);
 System.out.print("Enter the search key [integer]: ");
 int key = keyboard.nextInt();

 int index = Utility.linearSearch(numbers, key);
 if (index > -1) {
 System.out.println("Linear search found the number " + key
 + " in element with index " + index);
 }
 }
}
```

```

 } else {
 System.out.println("The number " + key + " is not in the array!");
 }
}

```

9



Program output when the given key exists in the array:

```

Array (unsorted): 18 15 5 2 12 17 21 3 6 25 22 1 10
Enter the search key [integer]: 17
Linear search found the number 17 in element with index 5

```

Program output when the given key is not found:

```

Array (unsorted): 18 15 5 2 12 17 21 3 6 25 22 1 10
Enter the search key [integer]: 32
The number 32 is not in the array!

```

---

## 9.6 Binary search

If values in an array are already sorted, we can use a more efficient search algorithm than linear search. For example, the phone directory is sorted according to last name, and it would clearly be a waste of time to start at the beginning of a phone directory to look up a person whose last name is “Smith”.

When an array is already sorted, we can start by comparing the search key with the middle element in the array. (We assume that the array is sorted in ascending order.) If the middle element is *less* than the key, we know that the key has to be in the upper half of the array, if it exists in the array at all. Analogously, the key must be in the lower half of the array if the middle element is *greater* than the key. We then repeat this procedure in the relevant half of the array. With this approach, the number of elements to be searched is halved in each step. The algorithm either finds the key or runs out of elements to compare with the key. This search algorithm is called *binary search*.

Assuming that the array is already sorted in ascending order, we can formulate the binary search algorithm as follows:

```

Repeat while there are elements to compare in the array:
 If the middle element is equal to the key
 Return the index of the middle element
 If the middle element is smaller than the key
 Continue search in the upper half
 If the middle element is larger than the key
 Continue search in the lower half

```

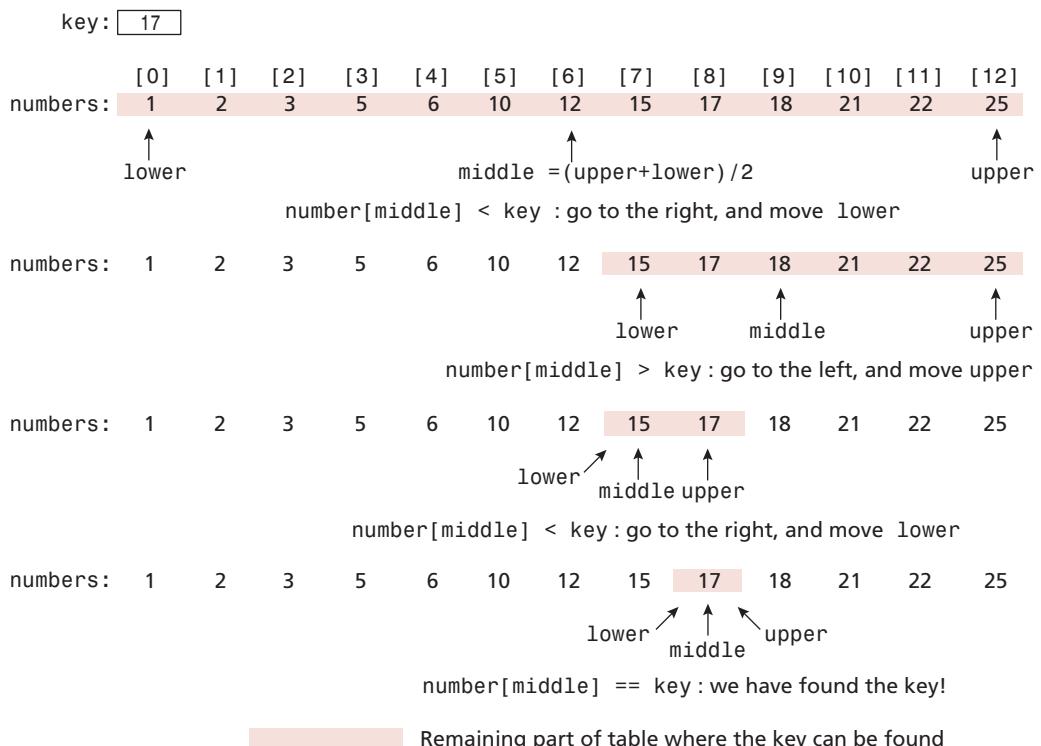
Figure 9.4 shows the steps for binary search in an array of integers when the search key is 17. We start by comparing the key to the middle element, which in this case contains the value 12. The middle element is less than the key, so we know that the key can only be in the upper half of the array. We repeat the procedure in the interval of the array defined by the variables `lower` and `upper`, and thus determine an even smaller part of the array that



can contain the key. Note how the variables `lower` and `upper` are updated to indicate the interval for the search. After repeatedly halving the array, we end up with a middle element whose value is equal to the search key, 17. The search stops and returns the index of the middle element, as the key has been found.

If the array does not contain the key, the algorithm will stop when it runs out of elements to compare with the key, i.e. when the interval in which to search becomes empty. In this case an invalid index (-1) is returned, indicating that the key was not found.

**FIGURE 9.4** Binary search for a key that is contained in the array



Remaining part of table where the key can be found

Program 9.8 implements binary search for arrays of `int` values. The loop at (1) ensures that the number of remaining elements to search is halved until the key is found, or until there are no more elements left to search. The index of the middle element is computed at (2). As long as the `lower` limit is less than or equal to the `upper` limit, the search interval is not empty. See the `if` statement at (3). Note how the interval limits `lower` and `upper` are updated at (4) and (5). In order to search in the *upper* half, the `lower` limit is incremented. In order to search in the *lower* half, the `upper` limit is decremented. Program 9.8 tests the algorithm with different keys.

For each pass, the number of elements that needs to be compared with the search key is halved. If  $n$  is the number of elements in the array, the amount of work is proportional to the natural logarithm of  $n$ ,  $\log_2(n)$ . For an array with 1048576 ( $2^{20}$ ) elements, the binary search algorithm will, in the worst case scenario, require only twenty comparisons to determine the result.

## PROGRAM 9.8 Binary search in arrays of integers

9

```
// Utility class for sorting and searching arrays of objects.
class Utility {
 static int binarySearch(int[] numbers, int key) {
 int lower = 0; // Lower limit of the search interval
 int upper = numbers.length-1; // Upper limit of the search interval
 int middle = -1; // Index of middle element
 boolean found = false; // Indicates whether the key is found
 boolean moreElements = true; // Indicates whether there are
 // more elements to search

 // Repeat while more elements in array:
 while (moreElements && !found) { // (1) Main loop
 middle = (lower + upper)/2; // (2) Middle element in this interval
 if (lower > upper) { // (3) Determine if interval is empty
 moreElements = false; // No more elements left to search
 } else if (numbers[middle] == key) { // Key is in the middle element
 found = true;
 } else if (numbers[middle] < key) {
 lower = middle + 1; // (4) Search in upper half
 } else {
 upper = middle - 1; // (5) Search in lower half
 }
 }
 assert (!moreElements || found) : "Error in binary search.";

 if (found) {
 return middle; // The key is in element numbers[middle]
 } else {
 return -1; // The key is not found
 }
 }

 // other methods...
}

// Looking up a search key in an array of integers using binary search.
import java.util.Scanner;
public class BinarySearch {
 public static void main(String[] args) {
 int[] numbers = { 1, 2, 3, 5, 6, 10, 12, 15, 17, 18, 21, 22, 25 };
 Utility.printArray("Array (sorted): ", numbers);

 Scanner keyboard = new Scanner(System.in);
 System.out.print("Enter the search key [integer]: ");
 int key = keyboard.nextInt();

 int index = Utility.binarySearch(numbers, key);
 }
}
```

```

 if (index > -1)
 System.out.println("Binary search found the number " + key
 + " in element with index " + index);
 else
 System.out.println("The number " + key + " is not in the array!");
}
}

```

9



Program output when the given key exists in the array:

```

Array (sorted): 1 2 3 5 6 10 12 15 17 18 21 22 25
Enter the search key [integer]: 17
Binary search found the number 17 in element with index 8

```

Program output when the given key does not exist:

```

Array (sorted): 1 2 3 5 6 10 12 15 17 18 21 22 25
Enter the search key [integer]: 32
The number 32 is not in the array!

```

---

## 9.7 Sorting and searching a CD collection

A small music store wants to develop a program that allows it to sort and search its inventory of CDs. Each CD is characterised by the artist, the title and the release year. In addition, the store wants to keep track of the price of a CD. The `CD2` class in Program 9.9 implements CDs with these properties.

The `CD2` class overrides the `equals()` method at (1). Two CDs are considered equal if they have the same artist, title and release year. Note the signature of the `equals()` method. The type of the formal parameter must be `Object`. The method tests at (2) whether the two references `this` and `obj` are aliases. At (3), the program checks that the reference `obj` refers to a `CD2` object. The `equals()` method calls the `compareTo()` method at (4) to do the actual comparison of the CDs.

The `CD2` class fulfils the `Comparable` interface by implementing the `compareTo()` method at (5). The collection of CDs are first ranked by the artist, then by the title and finally by the release year. We call the `compareTo()` method on the `String` fields to compare their content.

### PROGRAM 9.9 A simple CD class

```

// A simple CD class that can be ordered by artist, title and release year.
class CD2 implements Comparable {
 // Declaration of field variables
 String artist;
 String title;
 int released;
 double price;

```



```

// Creates a new CD with given artist, title, release year and price.
CD2(String artist, String title, int released, double price) {
 this.artist = artist;
 this.title = title;
 this.released = released;
 this.price = price;
}

// Returns a string representation of the CD.
public String toString () {
 return "CD: " + artist + " - '" + title + "' (" + released + ")"
 + " with price " + price;
}

// Determines if two CDs are equal, i.e. has same artist, title
// and release year.
public boolean equals(Object obj) { // (1)
 if (this == obj) // (2) Aliases?
 return true;
 if (!(obj instanceof CD2)) // (3) Has correct type?
 return false;
 return compareTo(obj) == 0 ; // (4)
}

// Compares two CDs with respect to artist, title and release year.
public int compareTo(Object obj) { // (5)
 CD2 otherCD = (CD2) obj;
 int status = artist.compareTo(otherCD.artist);
 if (status == 0) { // Same artist, rank by title
 status = title.compareTo(otherCD.title);
 if (status == 0) { // Same title, rank by release
 return released - otherCD.released;
 }
 }
 return status; // Ranked either by artist or title
}
}

```

Program 9.10 implements an array of CDs that can be sorted and searched. The array is created at (1), and the CD we want to look up is declared at (2). (In a real world system the initialisation would most likely be done from a file or a database, and the CD to search for being specified in a graphical user interface.)

The array is sorted at (3). To check that sorting is performed correctly, the entire array is printed at (4). Then, a particular CD is searched for at (5). The program reports the result of the search at (6) or (7) depending on whether the CD was found or not.



Note the call to the `sortBySelection()` method in the `GenericUtility` class at (3). This method was originally developed for sorting arrays of strings (see Program 9.6), but since the type of the array elements is `Comparable`, the method can be used to sort arrays of objects that implement this interface.

In Program 9.10, the type of the array elements in the search method `binarySearch()` is `Comparable`. This method can therefore be used for searching any array whose elements are of the type `Comparable`. This may come in handy if the music store decides to extend their inventory to sell, for instance, DVDs.

### PROGRAM 9.10 A simple CD collection with sorting and searching

```
// Sorting a CD selection.
import java.util.Arrays;
public class CDCollection {
 public static void main(String[] args) {
 CD2[] inventory = { new CD2("The Band", "Greatest Hits", 2006, 9.95),
 new CD2("Thorn", "The Album", 2005, 7.55),
 new CD2("The Band", "Greatest Hits", 2004, 8.95),
 new CD2("ZZZ", "ZZZ Songs", 2007, 12.95) }; // (1)
 CD2 key = new CD2("The Band", "Greatest Hits", 2004, 8.95); // (2)
 System.out.println("CD list:");
 for (CD2 cd : inventory) {
 System.out.println(cd);
 }
 GenericUtility.sortBySelection(inventory); // (3)
 System.out.println("Sorted CD list:");
 for (CD2 cd : inventory) { // (4)
 System.out.println(cd);
 }
 System.out.println("Searching:");
 int index = GenericUtility.binarySearch(inventory, key); // (5)
 if (index > -1) {
 System.out.println("Found " + key + " at index " + index); // (6)
 } else {
 System.out.println("Item " + key + " was not found!"); // (7)
 }
 }
}
// Utility class for sorting and searching arrays of objects.
class Utility {
 static int binarySearch(int[] numbers, int key) {
 int lower = 0; // Lower limit of the search interval
 int upper = numbers.length-1; // Upper limit of the search interval
 int middle = -1; // Index of middle element
 boolean found = false; // Indicates whether the key is found
 boolean moreElements = true; // Indicates whether there are
 // more elements to search
```



```

// Repeat while more elements in array:
while (moreElements && !found) { // (1) Main loop
 middle = (lower + upper)/2; // (2) Middle element in this interval
 if (lower > upper) { // (3) Determine if interval is empty
 moreElements = false; // No more elements left to search
 } else if (numbers[middle] == key) { // Key is in the middle element
 found = true;
 } else if (numbers[middle] < key) {
 lower = middle + 1; // (4) Search in upper half
 } else {
 upper = middle - 1; // (5) Search in lower half
 }
}
assert (!moreElements || found) : "Error in binary search.";

if (found) {
 return middle; // The key is in element numbers[middle]
} else {
 return -1; // The key is not found
}
}

// other methods...
}

```

Program output:

```

CD list:
CD: The Band - 'Greatest Hits' (2006) with price 9.95
CD: Thorn - 'The Album' (2005) with price 7.55
CD: The Band - 'Greatest Hits' (2004) with price 8.95
CD: ZZZ - 'ZZZ Songs' (2007) with price 12.95
Sorted CD list:
CD: The Band - 'Greatest Hits' (2004) with price 8.95
CD: The Band - 'Greatest Hits' (2006) with price 9.95
CD: Thorn - 'The Album' (2005) with price 7.55
CD: ZZZ - 'ZZZ Songs' (2007) with price 12.95
Searching:
Found CD: The Band - 'Greatest Hits' (2004) with price 8.95 at index 0

```

## 9.8 Sorting and searching using the Java standard library

The `java.util.Arrays` class in the Java standard library offers a number of methods for sorting and searching arrays. A small subset of these methods are shown in Table 9.2. The `binarySearch()` method requires that the elements are already sorted in *ascending natural order*. The `sort()` methods require that elements of an array of objects implement the `Comparable` interface in order to sort them.



Program 9.11 shows sorting and searching an array of integers using the methods from the `Arrays` class. The method `printArray()` from Program 9.2 is used to print array values in order to verify that the sorting algorithm behaves as expected.

**TABLE 9.2 Selected methods from the `Arrays` class**

| <code>java.util.Arrays</code>                                                                               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|-------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>static int binarySearch(<br/>    elementTypeName[] array,<br/>    elementTypeName key)</code>         | Uses binary search to look up the given key in the array.<br>Returns the index to the key in the array, if the key exists.<br>If not, a negative index is returned, corresponding to $(-\text{insertionpoint}) - 1$ , where <code>insertionpoint</code> is the index of the element where the key would have been if it had been in the array.<br><br>The array elements must already be sorted in <i>ascending natural order</i> , otherwise the result is undefined.<br>Permitted element types include <code>byte</code> , <code>char</code> , <code>double</code> , <code>float</code> , <code>int</code> , <code>long</code> , <code>short</code> and <code>Object</code> . |
| <code>static void sort(<br/>    elementTypeName[] array)</code>                                             | Sorts an array of elements with type <code>elementTypeName</code> in <i>ascending natural order</i> .<br><br>Only elements from the index <code>fromIndex</code> (inclusive) and the index <code>toIndex</code> (inclusive) are sorted, if these limits are specified. Permitted element types are <code>byte</code> , <code>char</code> , <code>double</code> , <code>float</code> , <code>int</code> , <code>long</code> , <code>short</code> and <code>Object</code> .<br><br>Elements in an array of objects must implement the <code>Comparable</code> interface.                                                                                                           |
| <code>static void sort(<br/>    elementTypeName[] array,<br/>    int fromIndex,<br/>    int toIndex)</code> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

**PROGRAM 9.11 Sorting and searching using methods in the `Arrays` class**

```
// Using the Java standard library for sorting and searching arrays.
import java.util.Arrays;
public class SortingByArrays {
 public static void main(String[] args) {
 int[] numbers = { 8, 4, 2, 6, 1 };
 Utility.printArray("Unsorted array: ", numbers);
 Arrays.sort(numbers);
 Utility.printArray("Array sorted by Java library: ", numbers);
 int key = 4;
 int index = Arrays.binarySearch(numbers, key);
 System.out.println("Key " + key + " has index " + index);
 key = 5;
 index = Arrays.binarySearch(numbers, key);
 System.out.println("Key " + key + " has index " + index);
 }
}
```

Program output:

```
Unsorted array: 8 4 2 6 1
Array sorted by Java library: 1 2 4 6 8
Key 4 has index 2
Key 5 has index -4
```

---

9





## 9.9 Review questions

- Values of all numerical data types in Java, such as `int` and `double`, have a \_\_\_\_\_. The Java programming language also defines a set of \_\_\_\_\_ operators that can be used to compare numerical values.
- The only primitive data type in Java whose values cannot be compared is \_\_\_\_\_. Values of this data type can only be compared for \_\_\_\_\_.
- What will be the output from statements (1) to (6) below, given that the program is run with the following command:

```
> java -ea ComparePrimitiveValues

// Compare values of primitive data types.
public class ComparePrimitiveValues {
 public static void main(String[] args) {
 // Tests for negative integers.
 assert -10 == -10; // (1)
 assert -10 <= -9; // (2)
 assert -10 < -9; // (3)
 assert -10 > -11; // (4)
 assert -10 >= -11; // (5)
 assert -10 >= -9; // (6)
 }
}
```

- Write at least three assertions verifying that the letter 'a' comes before the letter 'z' according to the lexicographical order of characters defined by Java.

- Write an assertion verifying that the Unicode standard defines an equal number of lowercase and uppercase letters in the English alphabet. The first and last lowercase and uppercase letters are 'a' and 'z', and 'A' and 'Z', respectively.

Hint: you will need to use both arithmetic and relational operators in the assertion.

- Write an assertion that verifies that a test score is between 0 and 100. Assume that the score is kept in a variable called `testScore`, and that the score limits are defined as constants names `MIN_SCORE` and `MAX_SCORE` respectively. Let your `assert` statement print a explanatory message if it fails.

- To enable comparison of two *objects*, their class has to implement the \_\_\_\_\_ interface, which defines the \_\_\_\_\_ method used to compare objects.

- A simple class for CDs is defined as follows:

```
// A simple CD class that can be ordered by title and # of tracks.
class CD {
 // Declaration of field variables
 String title;
 int noOfTracks;
```



```
// Creates a new CD with given title and # of tracks.
CD(String title, int noOfTracks) {
 this.title = title;
 this.noOfTracks = noOfTracks;
}

// Prepares a string representation of the CD.
public String toString () {
 return "The CD entitled '" + title + "' has " + noOfTracks + " tracks.";
}
}
```

The objects of this class can be ordered by their titles (lexicographically) and, if two titles are the same, by the number of tracks. In this case the CD with the highest number of tracks is considered the “largest”.

Which of the code snippets below provides a correct implementation of the natural order of `CD` objects? How must the class header for the `CD` class be modified?

- a**

```
public int compareTo(CD otherCD) {
 if (title == otherCD.title) {
 return noOfTracks - otherCD.noOfTracks;
 } else {
 return title.compareTo(otherCD.title);
 }
}
```
- b**

```
public int compareTo(Object otherCD) {
 if (title.equals(otherCD.title)) {
 return noOfTracks - otherCD.noOfTracks;
 } else {
 return title.compareTo(otherCD.title);
 }
}
```
- c**

```
public int compareTo(Object obj) {
 CD otherCD = (CD) obj;
 if (title.equals(otherCD.title)) {
 return noOfTracks - otherCD.noOfTracks;
 } else {
 return title.compareTo(otherCD.title);
 }
}
```

- 9.** The revised `CD` class from Exercise 9.8 is used in the following program:

```
// Sorting a CD selection.
import java.util.Arrays;
public class CDSorter {
 public static void main(String[] args) {
 CD[] newCDs = { new CD("Java Jive", 5),
 new CD("T Cup Blues", 7),
```



```

 new CD("Another cup of Joe", 6),
 new CD("T Cup Blues", 8) };
System.out.println("CD list:");
for (CD nextCD : newCDs) { System.out.println(nextCD); }
Arrays.sort(newCDs);
System.out.println("Sorted CD list:");
for (CD nextCD : newCDs) { System.out.println(nextCD); }
}
}

```

What will be the output from this program?

**10.** Which of the following statements are true?

- a** All objects whose fields have a natural order can be ordered using the relational operators offered by Java.
- b** An array of objects whose fields have a natural order can be sorted by the `sort()` method provided by the `java.util.Arrays` class.
- c** An array of objects whose class implements the `Comparable` interface can be sorted using the `sort()` method in the `java.util.Arrays` class.
- d** Only an array of objects whose class implements the `Comparable` interface can be sorted using the `sort()` method in the `java.util.Arrays` class.

**11.** Which of the following statements are true?

- a** Selection sort will run faster if the array is partially sorted in advance.
- b** Insertion sort will run faster if the array is partially sorted in advance.
- c** Selection sort requires fewer comparisons per pass than insertion sort.
- d** Selection sort requires fewer assignment operations per pass than insertion sort.

## 9.10 Programming exercises

**1.** We want to sort an array of integers in *descending* order. Write pseudocode for a selection sort algorithm that does this.

Simulate by hand on a piece of paper how the algorithm will work for the first pass on the following array:

{ 7, 2, 6, 1, 3, 4, 10, 8 }

Simulate sorting of the array after each pass.

**2.** Implement a `sortBySelectionDescending()` method in the `Utility` class, based on the pseudocode from Exercise 9.1. Test your program on the following array:

{ 7, 2, 6, 1, 3, 4, 10, 8 }

Verify that the program behaves as expected by printing the array after each pass. You can use the `printArray()` method of the `Utility` class to print the array values.



- 3.** Extend the program from Exercise 9.2 to allow the user to enter the values for an array of integers.

Example of user dialogue:

```
Sorting an array of integers in descending order
Enter the number of values in the array [integer]: 6
Enter element value #1: 11
Enter element value #2: 7
Enter element value #3: 23
Enter element value #4: 1
Enter element value #5: 16
Enter element value #6: 15
Unsorted array: 11 7 23 1 16 15
Pass #1: 23 7 1 11 16 15
Pass #2: 23 16 1 11 7 15
Pass #3: 23 16 15 11 7 1
Pass #4: 23 16 15 11 7 1
Pass #5: 23 16 15 11 7 1
Array sorted by selection sort: 23 16 15 11 7 1
```

- 4.** Extend Program 9.2 to allow the user to enter the values for the array. The user should also be able to specify the number of elements in the array.

Example of user dialogue:

```
Sorting an array of integers
Enter the number of values in the array [integer]: 6
Enter element value #1: 11
Enter element value #2: 7
Enter element value #3: 23
Enter element value #4: 1
Enter element value #5: 16
Enter element value #6: 15
Unsorted array: 11 7 23 1 16 15
Pass #1: 1 7 23 11 16 15
Pass #2: 1 7 23 11 16 15
Pass #3: 1 7 11 23 16 15
Pass #4: 1 7 11 15 16 23
Pass #5: 1 7 11 15 16 23
Array sorted by selection sort: 1 7 11 15 16 23
```

- 5.** Write a new program that allows the user to enter the values for an array of integers, and select between sorting in ascending or descending order. The new program should use the sorting algorithms from Exercise 9.3 and Exercise 9.4.

Example of user dialogue:

```
Sorting an array of integers
Enter the number of values in the array [integer]: 6
Enter element value #1: 11
Enter element value #2: 7
Enter element value #3: 23
Enter element value #4: 1
```



```

Enter element value #5: 16
Enter element value #6: 15
Enter sorting order [A for ascending, D for descending]: A
Unsorted array: 11 7 23 1 16 15
Pass #1: 1 7 23 11 16 15
Pass #2: 1 7 23 11 16 15
Pass #3: 1 7 11 23 16 15
Pass #4: 1 7 11 15 16 23
Pass #5: 1 7 11 15 16 23
Array sorted in ascending order: 1 7 11 15 16 23

```

- 6.** The following pseudocode for selection sort is meant to sort an array in descending order:

For each pass:

Find the largest value in the unsorted part of the array

Swap the largest value with the last element in the unsorted partSimulate by hand on a piece of paper how this algorithm will work for the following array:

{ 7, 2, 6, 1, 3, 4, 10, 8 }

Modify the pseudocode so that it correctly sorts the array in descending order.

- 7.** Implement the algorithm for insertion sort from Exercise 9.6. Verify that it sorts the array in descending order by writing a client program that uses the algorithm to sort the same array that was used in the hand simulation.
- 8.** A sorting algorithm called *bubble sort* works by letting smaller values *bubble* up towards the beginning of the array, while larger values *sink* towards the end of the array. Figure 9.5 illustrates how bubble sort works during the first pass of the array

{ 8, 4, 2, 6, 1 }

The algorithm makes the largest element sink all the way down to the end of the array, by comparing pairs of neighbouring values. It swaps the values in a pair if they are not in correct order.

The pseudocode for the bubble sort algorithm is given below. Use this pseudocode as a starting point for implementing the algorithm.

Pseudocode for bubble sort (in ascending order):

For each pass of the array:

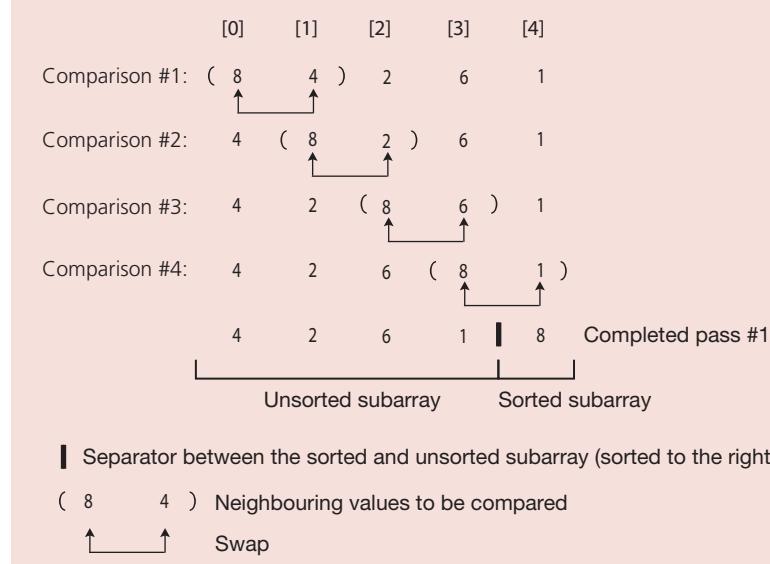
For each pair of neighbouring values in the unsorted part of the array:

If values in the pair are in the wrong order

Swap the values in the pair



**FIGURE 9.5** Sorting an array of integers using bubble sort



9. Modify the `Time` class in Program 9.5 so that the fields `hours` and `minutes` are declared as `Integer` references. Show how we can use this fact in the implementation of the `compareTo()` method.
  10. Refine the `compareTo()` method for the `Time` class in Program 9.5 by taking into consideration that the *total number of minutes* (`hours*60 + minutes`) represented by each time of the day are also in chronological order.
  11. In Scrabble, each player has a set of pieces with a letter on each piece. The pieces are used to make words on a game board. If a player for example has the pieces 'e', 'c', 's', 't', 'k' and 'a', they can make the word "cake" by using four of their pieces. Alternatively, they can make the word "steak" by using all but one piece. When playing Scrabble it may be helpful to sort the pieces beforehand to make it easier to find letters that can be used to form new words. Write a client that reads the letters on the pieces, sorts them and prints them out in the terminal window. Use the data type `char` for the letter on each piece. Select a suitable number of pieces for this purpose.
- Hint: read the characters as a string using the `Scanner` class, then loop through the string to extract individual characters.
12. Assume that a linear search for the value 4 is performed on the following array using the linear search algorithm in Program 9.7.

{ 2, 6, 4, 3, 7, 4 }

Which index is returned by the algorithm in Program 9.7?

Write pseudocode for an extended version of the linear search algorithm that reports indices of *all* elements that are equal to the search key. Modify Program 9.7 to report indices of all elements equal to the search key. Run the revised program using 4 as



the search key again, and verify that it behaves as expected. Also, run the program with a key that only occurs once in the array, and with keys that are not in the array.

- 13.** Extend the program from Exercise 9.12 to allow the user to specify an array of values and the search key from the keyboard.

An example of a dialogue for reading the array length and the element values can be found in Exercise 9.4.

- 14.** Extend the program in Exercise 9.4 to allow binary search in the sorted array.

Example of user dialogue:

```
Sorting and searching an array of integers
Enter the number of values in the array [integer]: 6
Enter element value #1: 11
Enter element value #2: 7
Enter element value #3: 23
Enter element value #4: 1
Enter element value #5: 16
Enter element value #6: 15
Unsorted array: 11 7 23 1 16 15
Pass #1:1 7 23 11 16 15
Pass #2:1 7 23 11 16 15
Pass #3:1 7 11 23 16 15
Pass #4:1 7 11 15 16 23
Pass #5:1 7 11 15 16 23
Array sorted by selection sort: 1 7 11 15 16 23
Enter the search key [integer]: 11
Binary search found the number 11 in element with index 2
```

- 15.** Based on the `EmployeeV7` class from Chapter 7, implement the natural order for employees in a small company, where the criteria for determining their order is their last and first names respectively. Use the new `EmployeeV7` class to sort the following array:

```
Employee[] salesmen = {
 new Employee("Bart", "Simpson", 20.55, Employee.MALE),
 new Employee("John", "Doe", 16.23, Employee.MALE),
 new Employee("Jane", "Doe", 12.95, Employee.FEMALE),
 new Employee("Peggy", "Sue", 13.42, Employee.FEMALE),
};
```

Select a sorting method of your choice from the `GenericUtility` class, and use the `printArray()` method to print both the unsorted and sorted `EmployeeV7` array in the terminal window.

- 16.** Implement a simple lottery game that does the following:

- a** Randomly selects seven numbers out of a set of integers ranging from one to thirty-four.



- b** Allows the user to enter their ticket numbers (with only one game) and compare it to the drawn lottery numbers. Assume that the numbers entered and the numbers drawn are sorted in ascending order.
- c** Reports whether the user has won first prize or not. Winning first prize requires that all seven numbers match.

Before you start programming, write pseudocode that defines the steps needed to solve the problem.

- 17.** Hint: You can use the `java.util.Random` class to generate the lottery numbers.
- 18.** Make the program in Exercise 9.16 more realistic by allowing the user to enter their lottery ticket numbers in any order. The program should sort both the drawn numbers and the user ticket numbers before comparing the two number series. Extend the pseudocode from Exercise 9.16 to clarify the changes that are needed to incorporate the new features before writing any new code. Be sure to include your updated pseudocode in the source code, to document your program.



## Exception handling

### LEARNING OBJECTIVES

By the end of this chapter, you will understand the following:

- How to use exception handling to create programs that are reliable and robust.
- How methods are executed by the JVM, and how it handles error situations.
- Major scenarios of program execution when using the **try-catch** statement.
- How exceptions are thrown, propagated, caught and handled.
- The difference between checked and unchecked exceptions.

### INTRODUCTION

A program must be able to handle error situations gracefully when they occur at runtime. Java provides exception handling for this purpose. In this chapter we take a closer look at how such error situations can occur, and how they can be handled by the program. Further aspects of exception handling are discussed in Chapter 18.

#### 10.1 What is an exception?

A program should be able to handle error situations that might occur at runtime. Error situations can be divided into two main categories:

- 1 *Programming errors.* For example, using an invalid index to access an array element, attempting to divide by zero, calling a method with illegal arguments, or using a reference with the `null` value to access members of an object.
- 2 *Runtime environment errors.* For example, opening a file that does not exist, a read or write error when using a file, or a network connection going down unexpectedly.



Programming errors are error situations that occur because of *logical errors* in the program, while runtime environment errors are errors over which the program has little control. A program must be able to handle both kinds of errors. Ideally, programming errors should not occur, and the program should handle runtime environment errors gracefully.

An *exception* in Java signals that an error or an unexpected situation has occurred during program execution. *Exception handling* is the mechanism for dealing with such situations. It is based on the "*throw and catch*" principle. An exception is *thrown* when an error situation occurs during program execution. It is *propagated* by the JVM and *caught* by an *exception handler* that takes an appropriate action to *handle* the situation. This principle is embedded in the try-catch statement. All exceptions are objects, and the Java standard library provides classes that represent different types of exceptions.

The examples in this chapter show how exceptions can be thrown, propagated, caught and handled. These examples are intentionally simplified to highlight the important concepts in exception handling.

## 10.2 Method execution and exception propagation

We will use Program 10.1 as our running example in this chapter, and will write several versions of Program 10.1 to illustrate exception handling. We want to calculate speed when distance and time are given. The program uses three methods:

- 1 The method `main()`, that calls the `printSpeed()` method with parameter values for distance and time, (1).
- 2 The method `printSpeed()`, that in turn calls the `calculateSpeed()` method, (2).
- 3 The method `calculateSpeed()`, that calculates the expression `(distance/time)` and returns the result in a `return` statement, (3).

Observe that integer division is performed in the evaluation of the expression `(distance/time)`, as both operands are integers, and that integer division by 0 is an *illegal* operation in Java. Attempt at integer division by 0 will result in a runtime error. (On the other hand, floating-point division in an expression like `10.0/0.0` will result in an infinitely large number, denoted by the constant `Double.POSITIVE_INFINITY`.)

### Method execution

During program execution, the JVM uses a program stack to control the execution of methods (see Section 16.2 about stacks and Chapter 17 about recursion). Each stack frame on the program stack corresponds to one method call. Each method call results in the creation of a new stack frame that has storage for local variables (including parameters) in the method. The method whose stack frame is on top of the program stack, is the one currently being executed. When the method returns (i.e. has finished executing), its stack frame is removed from the top of the stack. Program execution continues in the method whose stack frame is now uncovered on top of the program stack. This execution behaviour is called *normal execution*.



An important aspect of program execution is that if method A() calls method B(), method A() cannot continue its execution until method B() completes execution. At any given time during execution, the program stack will contain stack frames of methods that are *active*, i.e. methods that have been called but whose execution has not completed. If we at any given time print the information about all the active methods on the program stack, we obtain what is called a *stack trace*. The stack trace shows which methods are active while the current method on top of the stack is executing.

Execution of Program 10.1 is illustrated in the sequence diagram in Figure 10.1. Execution of a method is shown as a box with local variables. The length of the box indicates how long a method is active. Just before the return statement in (3) is executed, we see that the program stack at this particular time contains three active methods: `main()`, `printSpeed()` and `calculateSpeed()`. The return statement in (3) returns the result from the `calculateSpeed()` method and completes the execution of this method. We see that the output from the program corresponds to the sequence of method calls in Figure 10.1.

### PROGRAM 10.1 Method entry and return

```
public class Speed1 {

 public static void main(String[] args) {
 System.out.println("Entering main().");
 printSpeed(100, 20); // (1)
 System.out.println("Returning from main().");
 }

 private static void printSpeed(int kilometers, int hours) {
 System.out.println("Entering printSpeed().");
 int speed = calculateSpeed(kilometers, hours); // (2)
 System.out.println("Speed = " +
 kilometers + "/" + hours + " = " + speed);
 System.out.println("Returning from printSpeed().");
 }

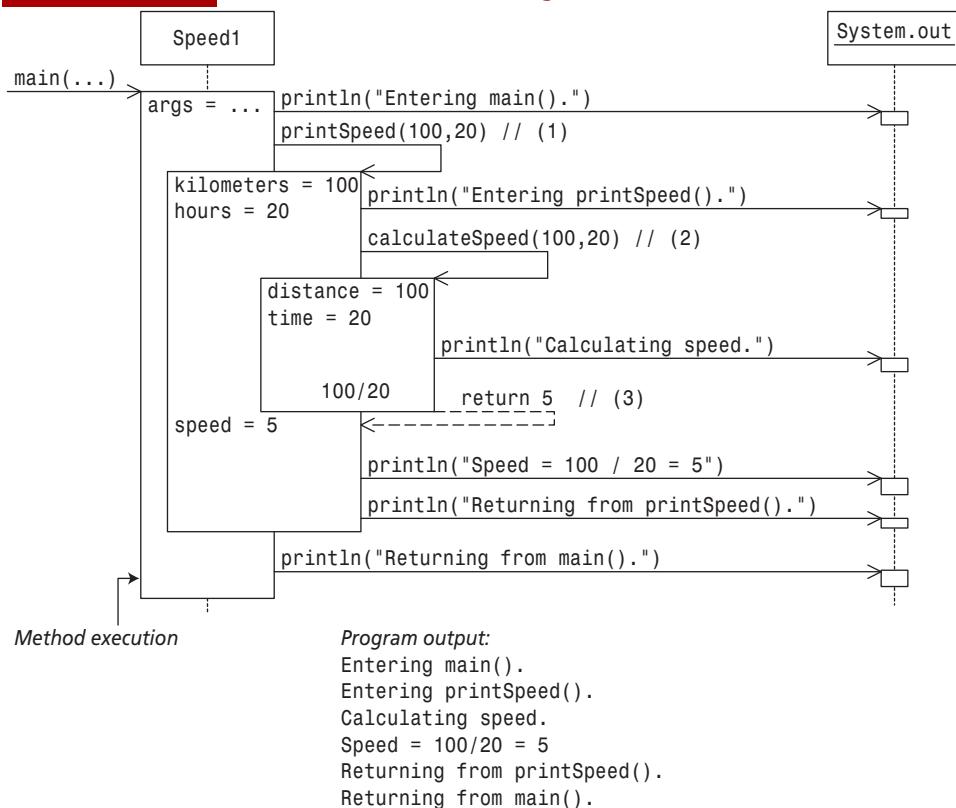
 private static int calculateSpeed(int distance, int time) {
 System.out.println("Calculating speed.");
 return distance/time; // (3)
 }
}
```

Program output:

```
Entering main().
Entering printSpeed().
Calculating speed.
Speed = 100/20 = 5
Returning from printSpeed().
Returning from main().
```



**FIGURE 10.1** Method execution (Program 10.1)



### BEST PRACTICES

Use of print statements as illustrated in Program 10.1 is a useful debugging technique in tracking program execution.

### Stack trace

If we replace the following call in the `main()` method from Program 10.1:

```
printSpeed(100, 20); // (1)
```

with

```
printSpeed(100, 0); // (1) The second parameter is 0.
```

and run the program, we get the following output in the terminal window:

```
Entering main().
Entering printSpeed().
Calculating speed.
Exception in thread "main" java.lang.ArithmetricException: / by zero
```



```
at Speed1.calculateSpeed(Speed1.java:20)
at Speed1.printSpeed(Speed1.java:12)
at Speed1.main(Speed1.java:6)
```

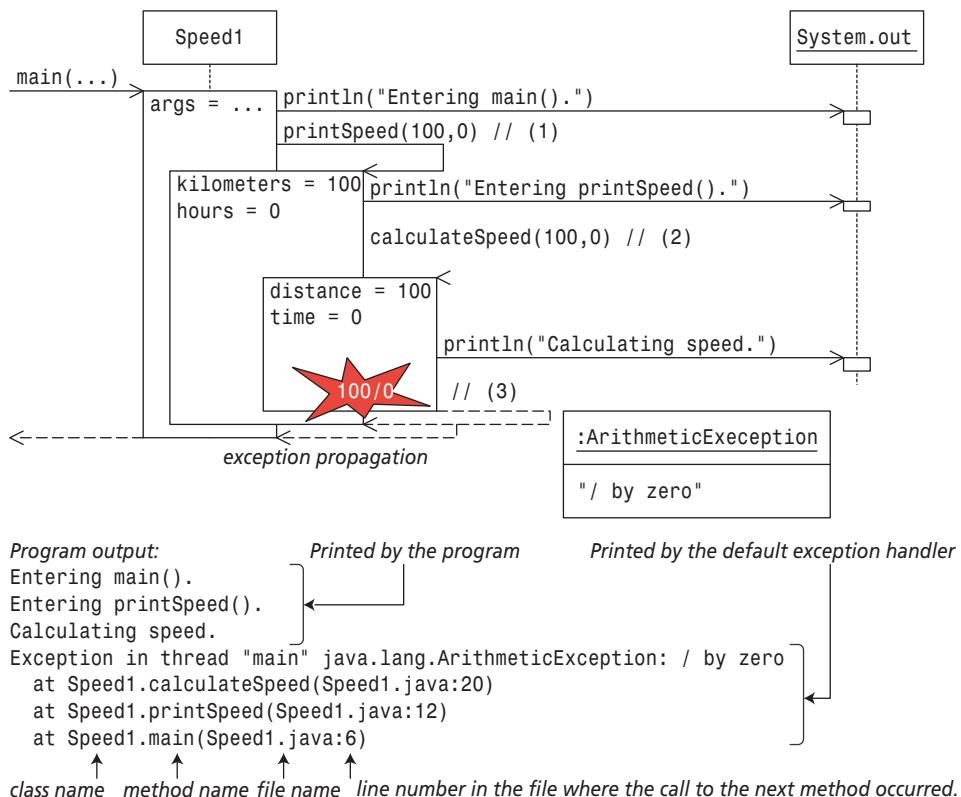
The execution of the program is illustrated in the sequence diagram in Figure 10.2. We see that all goes well right until the execution of the statement at (3), as corroborated by the output from the program. An error situation occurs at (3) during the evaluation of the expression `distance/time`, because the variable `time` has the value 0. This error situation is signalled by *throwing* an `ArithmeticException` (i.e. an object of the class `ArithmeticException`), which is sent back (we say *propagated*) through the stack frames on the program stack. The JVM takes care of forwarding an exception to the active methods on the program stack in the right order.

## Exception propagation

Propagation of an exception takes place in the following way: the exception is offered to the method whose stack frame is on top of the program stack, i.e. the method in which exception occurred. In this case it is the method `calculateSpeed()`. Since this method does not have any code to deal with this exception, the `calculateSpeed()` method is terminated, and its stack frame on top of the program stack is removed. The exception is next offered to the active method now on top of the program stack: the `printSpeed()` method. It also does not have any code to handle the exception, and consequently it is also terminated, and its stack frame removed. The exception is next offered to the `main()` method, which is also terminated and its stack frame removed, since it does not have any code for exception handling either. Now the exception has propagated to the *top level*, and here it is handled by a *default exception handler* in the JVM. This exception handler prints information about the exception, together with the stack trace at the time when the exception occurred (see Figure 10.2). The execution of the program is then terminated.

By comparing Figure 10.1 and Figure 10.2, we see that the error situation at runtime had great influence on the behaviour of the program. Program execution does not continue in the normal way under exception propagation, and the exception is not forwarded by any `return` statement. The execution of each active method on the program stack is successively terminated when an exception is propagated, unless an active method catches the exception and thereby stops its propagation.

For terminal window-based applications that we have developed so far, the program is terminated after the exception is handled by a default exception handler. For applications with a GUI (Chapter 20), the program continues after the exception has been handled by a default exception handler. In both cases a default exception handler prints information from the program stack to the terminal window.


**FIGURE 10.2** Exception propagation (integer division by 0)


## 10.3 Exception handling

In many cases it is not advisable to let exceptions be handled by a default exception handler. The consequences of terminating the program execution too early can be drastic, for example, data can be lost. The language construct try-catch can be used for exception handling in Java. Figure 10.3 shows a try block followed by a catch block. A try block consists of a block with the keyword try in front. A try block can contain arbitrary code, but normally it contains statements that can potentially result in an exception being thrown during execution. A catch block is associated with a try block. A catch block constitutes an exception handler. An exception can be thrown as a result of executing the code in the try block, and this exception can be caught and handled in an associated catch block. The block notation, {}, is required for the try and catch blocks, even if a block only contains a single statement.

A catch block resembles a method declaration. The head of a catch block consists of the keyword catch and the declaration of a single parameter specifying which type of exceptions this catch block can handle. A catch block can contain arbitrary code, but its main purpose is to execute actions for handling the error situation that the exception caught by the catch block represents.



Since a try block is a local block, the local variables declared in the block can only be used in the block itself. The same is also true for a catch block, where the exception parameter is considered a local variable, as is the case for formal parameters of a method.

**FIGURE 10.3 try-catch statement**

The try block contains the code that can lead to an exception being thrown.

```

try {
 int speed = calculateSpeed(kilometers, hours);
 System.out.println("Speed = " +
 kilometers + "/" + hours + " = " + speed);
}

```

try block

catch block

```

catch (ArithmaticException exception) { one catch block parameter
 System.out.println(exception + " (handled in printSpeed())");
}

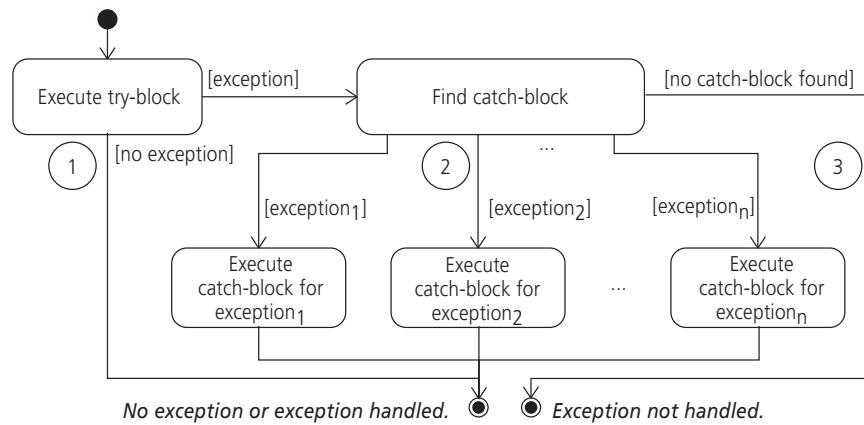
```

A catch block can catch an exception and handle it, if it is of the right type.

Figure 10.4 illustrates three typical scenarios when using the try-catch statement. These scenarios comprise the following situations during execution:

- 1 The code in the try block is executed, and no exception is thrown.
- 2 The code in the try block is executed, and an exception is thrown. This exception is caught and handled in a corresponding catch block.
- 3 The code in the try block is executed, an exception is thrown, but no catch block is found for handling the exception.

**FIGURE 10.4 try-catch scenarios**



Normal execution continues after the try-catch-blocks.

Exception not handled.

Execution aborted and exception propagated.

For the scenarios 2 and 3 in Figure 10.4 the execution of the try block is terminated when an exception is thrown, skipping the rest of the try block.

For the scenarios 1 and 2 in Figure 10.4 normal execution continues after the try-catch blocks, as there is no exception to be propagated. For scenario 3 the exception will be propagated as described in Section 10.2.



If the exception is caught and handled during its propagation, normal execution is resumed from that moment onwards. This means that an exception can be handled in a different method than the one it was thrown in, and a catch block is only executed if it catches an exception. The sequence in which statements are executed, and thereby the sequence of active methods on the program stack, is called the *runtime behaviour* of the program. It determines which exceptions can be thrown, and how these are handled.

### try-catch scenario 1: no exception

The method `printSpeed()` in Program 10.2 uses a try-catch statement to handle exceptions of the type `ArithmaticException`. This method calls the `calculateSpeed()` method in the try block at (2). The corresponding catch block, (4), is declared to catch exceptions of the type `ArithmaticException`. The handling of such an exception in the catch block consists of printing the exception. This type of exception can occur in the method `calculateSpeed()` during the evaluation of the arithmetic expression in (5). This means that if such an exception is thrown in the method `calculateSpeed()`, it will be caught in the method `printSpeed()`.

The runtime behaviour for Program 10.2 is shown in Figure 10.5. This behaviour corresponds to scenario 1 in Figure 10.4, where no exception occurs in the execution of the try block. The entire try block in the method `printSpeed()` is executed, while the catch block is skipped, as no exception is thrown in the method `calculateSpeed()`. We note that the execution of Program 10.2 shown in Figure 10.5 corresponds with the execution of Program 10.1 shown in Figure 10.1, and we get the same output in the terminal window in both programs.

#### PROGRAM 10.2    Exception handling

```
public class Speed2 {

 public static void main(String[] args) {
 System.out.println("Entering main().");
 printSpeed(100, 20); // (1)
 System.out.println("Returning from main().");
 }

 private static void printSpeed(int kilometers, int hours) {
 System.out.println("Entering printSpeed().");
 try { // (2)
 int speed = calculateSpeed(kilometers, hours); // (3)
 System.out.println("Speed = " +
 kilometers + "/" + hours + " = " + speed);
 }
 catch (ArithmaticException exception) { // (4)
 System.out.println(exception + " (handled in printSpeed())");
 }
 System.out.println("Returning from printSpeed().");
 }
}
```

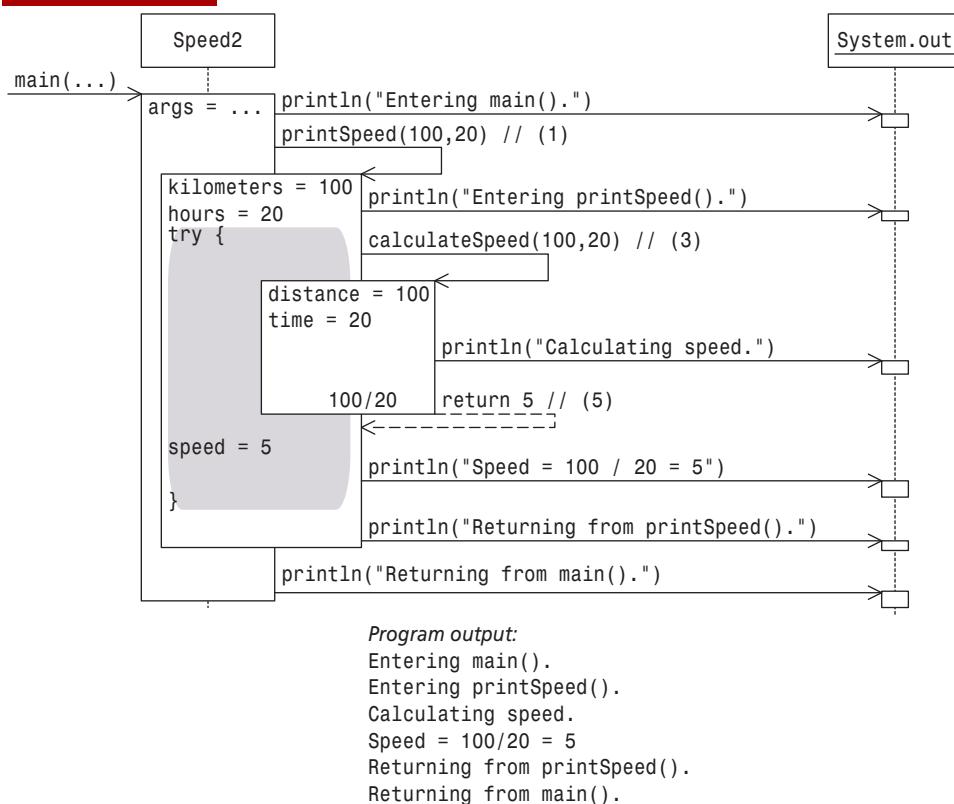


```

private static int calculateSpeed(int distance, int time) {
 System.out.println("Calculating speed.");
 return distance/time; // (5)
}
}
Entering main().
Entering printSpeed().
Calculating speed.
Speed = 100/20 = 5
Returning from printSpeed().
Returning from main().

```

**FIGURE 10.5** Exception handling (Program 10.2) (scenario 1 in Figure 10.4)



## try-catch scenario 2: exception handling

If we again replace the following call in the `main()` method from Program 10.2:

```
printSpeed(100, 20); // (1)
```

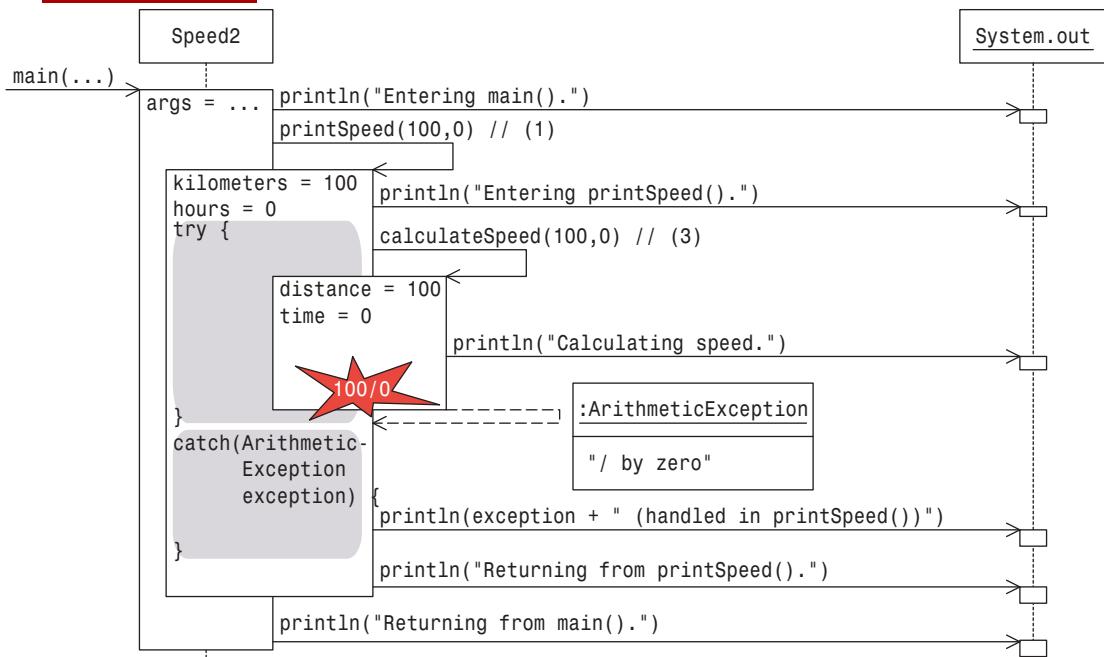
with

```
printSpeed(100, 0); // (1) Second parameter is 0.
```



the program behaviour is as shown in Figure 10.6. It corresponds to scenario 2 in Figure 10.4. Integer division by 0 results in an `ArithmeticException` being thrown at (5) in the `calculateSpeed()` method. The execution of this method is terminated, and the exception propagated. It is caught by the catch block in the method `printSpeed()`. After handling the exception, normal execution of the program is resumed, as shown by the output in Figure 10.6.

**FIGURE 10.6** Exception handling (Program 10.2) (scenario 2 in Figure 10.4)



#### Program output:

```

Entering main().
Entering printSpeed().
Calculating speed.
java.lang.ArithmaticException: / by zero (handled in printSpeed())
Returning from printSpeed().
Returning from main().

```

### try-catch scenario 3: exception propagation

Program 10.3 illustrates scenario 3 from Figure 10.3. Scenario 3 shows what can happen when an exception is thrown during the execution of a try block and no corresponding catch block is found to handle the exception. The scenario shows that the exception is propagated in the usual way, as we have seen in Section 10.2.

In Program 10.3 both the `main()` and the `printSpeed()` methods use the try-catch statement at (1) and (4), respectively. The `main()` method has a catch block to catch an exception of the type `ArithmaticException`, (3), while the `printSpeed()` method has a catch block to catch an exception of the type `IllegalArgumentExeption`, (6).



The runtime behaviour of the program (Figure 10.7) shows that integer division by 0 again results in an `ArithmaticException` being thrown at (7) in the method `calculateSpeed()`. The execution of this method is terminated, and the exception is propagated. It is *not* caught by the catch block in the method `printSpeed()`, since this catch block is declared to catch exceptions of the type `IllegalArgumentExpection`, and not exceptions of the type `ArithmaticException`. The execution of the method `printSpeed()` is terminated (statements after (5) are not executed), and the exception is propagated further. The exception `ArithmaticException` is now caught by the catch block in the `main()` method at (3). After handling of the exception in the catch block in the `main()` method, normal execution of the program is resumed, as can be seen from the output in Program 10.3.

### PROGRAM 10.3 Exception handling (scenario 3 in Figure 10.4)

```
public class Speed3 {

 public static void main(String[] args) {
 System.out.println("Entering main().");
 try { // (1)
 printSpeed(100,20); // (2)
 }
 catch (ArithmaticException exception) { // (3)
 System.out.println(exception + " (handled in main())");
 }
 System.out.println("Returning from main().");
 }

 private static void printSpeed(int kilometers, int hours) {
 System.out.println("Entering printSpeed().");
 try { // (4)
 int speed = calculateSpeed(kilometers, hours); // (5)
 System.out.println("Speed = " +
 kilometers + "/" + hours + " = " + speed);
 }
 catch (IllegalArgumentExpection exception) { // (6)
 System.out.println(exception + " (handled in printSpeed())");
 }
 System.out.println("Returning from printSpeed().");
 }

 private static int calculateSpeed(int distance, int time) {
 System.out.println("Calculating speed.");
 return distance/time; // (7)
 }
}
```

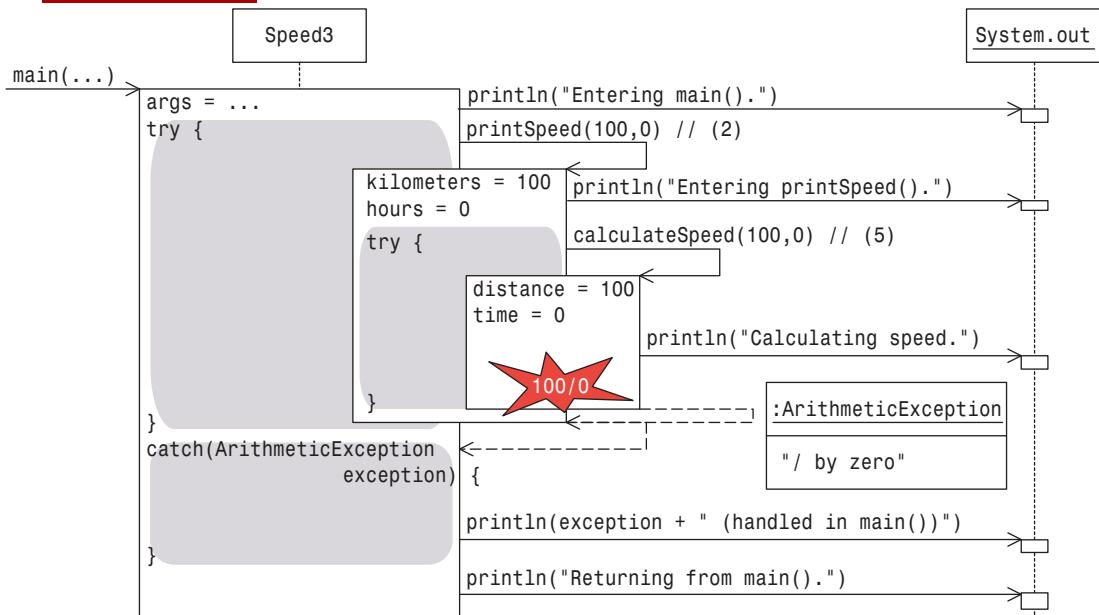
Program output:

```
Entering main().
Entering printSpeed().
Calculating speed.
```



```
java.lang.ArithmeticException: / by zero (handled in main())
Returning from main().
```

**FIGURE 10.7** Exception handling (Program 10.3) (scenario 3 in Figure 10.4)



#### Program output:

```
Entering main().
Entering printSpeed().
Calculating speed.
java.lang.ArithmeticException: / by zero (handled in main())
Returning from main().
```

#### BEST PRACTICES

Start designing your exception handling strategy from the beginning, rather than tacking it on when the implementation is done.

#### BEST PRACTICES

Exception handling carries a performance penalty. For that reason, avoid placing a try-catch statement inside a loop.



## 10.4 Checked exceptions

Exception handling allows the program to deal with error situations during program execution. It is possible for a method that throws an exception to let the exception propagate further, without doing anything about it. However, Java defines some special exceptions that a program simply cannot ignore when they are thrown. Such an exception is called a *checked exception*, because the compiler will complain if the method in which it can occur does not deal with it explicitly. Checked exceptions thus force the code in which they can occur to take explicit action to deal with them, resulting in programs that are *robust*, i.e. are able to handle error situations appropriately.

The Java standard library defines classes whose objects represent exceptions. Table 10.1 shows a selection of *checked* exceptions. The `Exception` class represents the category of all checked exceptions. The `ClassNotFoundException` is thrown when the class that a program uses cannot be found. You have mostly likely seen it when you misspelled the name of the class with the `main()` method on the command line while starting a program. The last three checked exceptions are in the `java.io` package and signal error situations that occur when dealing with files and streams (see Chapter 11 and Chapter 19). We will take a closer look at the main categories of exceptions in Section 18.1 on page 569.

**TABLE 10.1 Selected checked exceptions**

| Checked exception class (in the <code>Description</code> <code>java.lang</code> package, unless otherwise noted) |                                                                                                                                                                                                                                                                                                         |
|------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Exception</code>                                                                                           | This class represents the category of all checked exceptions.                                                                                                                                                                                                                                           |
| <code>ClassNotFoundException</code>                                                                              | Attempt to load a class during execution, but the class cannot be found.                                                                                                                                                                                                                                |
| <code>java.io.IOException</code>                                                                                 | Signals error during reading and writing of data. For example, the <code>read()</code> methods in the interface <code>InputStream</code> and the <code>write()</code> methods in the interface <code>OutputStream</code> throw this exception. (See Chapter 11 and Chapter 19 on files and streams).    |
| <code>java.io.EOFException</code>                                                                                | Signals unexpected end of input. For example, the <code>read()</code> methods in the interface <code>InputStream</code> throw this exception. (See Chapter 11 and Chapter 19 on files and streams).                                                                                                     |
| <code>java.io.FileNotFoundException</code>                                                                       | Attempt to refer to a file that does not exist. For example, the constructors in the classes <code>FileInputStream</code> , <code>FileOutputStream</code> and <code>RandomAccessFile</code> throw this exception, if the file cannot be assigned. (See Chapter 11 and Chapter 19 on files and streams). |



## Dealing with checked exceptions using the **throws** clause

A method that can throw a checked exception, must satisfy *one* of the following two conditions:

- 1** Catch and handle the exception in a try-catch statement, as we have discussed earlier.
- 2** Allow further propagation of the exception with a throws clause specified in its method declaration, which we will discuss here.

A throws clause is specified in the method header, between the parameter list and the method body:

```
... method name (...) throws exception class1, ..., exception classn { ... }
```

Let us say that a method `B()` can throw a checked exception `K`, and that method `B()` chooses to propagate exception `K` further with the help of a throws clause in its method declaration. If a method `A()` calls method `B()`, then method `A()` must now take a stand on how to deal with exception `K`, because method `A()` can now indirectly throw exception `K` that it can receive from method `B()`. This means that each client of a method that can propagate a checked exception in a throws clause, must decide how to deal with this exception. The compiler will check that a method that can throw a checked exception, satisfies one of the two conditions listed above. If a checked exception that is specified in a throws clause is propagated to the top level (i.e. not caught by any catch block), it will be handled by a default exception handler in the usual way (Section 10.2).

## Programming with checked exceptions

In Program 10.4 the `calculateSpeed()` method can throw a checked exception of the type `Exception` in an if statement:

```
if (distance < 0 || time <= 0) // (6)
 throw new Exception("distance and time must be > 0");
```

We can use a `throw` statement to throw an exception explicitly, by specifying the exception object to be thrown in the statement. In this case, we call the constructor of the exception class and pass a suitable message to explain the error situation. An object of the class `Exception`, with the string "distance and time must be > 0", is thrown by the `throw` statement above. More details on using the `throw` statement can be found in Section 18.2 on page 571.

With the setup in Program 10.4, the `calculateSpeed()` method must decide how to deal with a checked `Exception`. It chooses to throw this exception further in a throws clause, (5). The `printSpeed()` method, that calls the `calculateSpeed()` method, must therefore take a stand on this checked exception as well. It also chooses to throw it further in a throws clause, (4). Since the `main()` method calls the `printSpeed()` method, the `main()` method must also decide what to do with this exception. The `main()` method chooses to catch and handle this exception in a try-catch block, (1) and (3). Any attempt to leave out this exception from the throws clauses at (5) and (6) will result in a compile-time error.



We will see several examples that use checked exceptions in the chapters on files and streams (Chapter 11 and Chapter 19).

#### PROGRAM 10.4 Handling checked exceptions

```
public class Speed6 {

 public static void main(String[] args) {
 System.out.println("Entering main().");
 try {
 // printSpeed(100, 20); //(1)
 printSpeed(-100,20); //(2a)
 // printSpeed(-100,20); //(2b)
 }
 catch (Exception exception) { //(3)
 System.out.println(exception + " (handled in main())");
 }
 System.out.println("Returning from main().");
 }

 private static void printSpeed(int kilometers, int hours)
 throws Exception { //(4)
 System.out.println("Entering printSpeed().");
 double speed = calculateSpeed(kilometers, hours);
 System.out.println("Speed = " +
 kilometers + "/" + hours + " = " + speed);
 System.out.println("Returning from printSpeed().");
 }

 private static int calculateSpeed(int distance, int time)
 throws Exception { //(5)
 System.out.println("Calculating speed.");
 if (distance < 0 || time <= 0) //(6)
 throw new Exception("distance and time must be > 0");
 return distance/time;
 }
}
```

Output from Program 10.4 when (2a) is in the program, and (2b) is commented out:

```
Entering main().
Entering printSpeed().
Calculating speed.
Speed = 100/20 = 5.0
Returning from printSpeed().
Returning from main().
```

Output from Program 10.4 when (2b) is in the program, and (2a) is commented out:

```
Entering main().
Entering printSpeed().
Calculating speed.
```



```
java.lang.Exception: distance and time must be > 0 (handled in main())
Returning from main().
```

## 10.5 Unchecked exceptions

Since Java provides checked exceptions, it begs the question: does Java provide *unchecked exceptions*? The answer is "yes". Unchecked exceptions are exceptions typically concerning unforeseen errors, such as *programming errors*. Table 10.2 shows some common unchecked exceptions that you are likely to come across when you program in Java. We have already seen one example where an `ArithmaticException` and an `IllegalArgumentEx-`ception can be thrown (Program 10.3).

In contrast to checked exceptions, the compiler does *not* check whether unchecked exceptions can be thrown. This means that a method does not have to deal with unchecked exceptions, but then it must also face the consequences if the program is terminated due to an unchecked exception.

The best solution for handling such situations is to correct the cause of the error in the program, so that they do *not* occur during program execution. For example, selection statements or assertions can be used to check the conditions that a method imposes on its actual parameters, before we call the method.

Since the compiler ignores unchecked exceptions and a method is not forced to handle them, such exceptions are usually not specified in the `throws` clause of a method.

**TABLE 10.2 Selected unchecked exceptions**

| Unchecked exception class (in the <code>java.lang</code> package) | Description                                                                                                                                                                                                                                                                                         |
|-------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>RuntimeException</code>                                     | This class represents one category of unchecked exceptions.                                                                                                                                                                                                                                         |
| <code>NullPointerException</code>                                 | Attempt to use a reference that has the value <code>null</code> , i.e. the reference does not refer to an object. For example, the expression <code>new String(null)</code> throws this exception, since the parameter has the value <code>null</code> , instead of being a reference to an object. |
| <code>ArithmaticException</code>                                  | Illegal arithmetic operation, for example integer division with 0, e.g. <code>10/0</code> .                                                                                                                                                                                                         |



| Unchecked exception class (in the <code>java.lang</code> package) | Description                                                                                                                                                                                                         |
|-------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ClassCastException</code>                                   | Attempt to convert an object's reference value to a type that it does not belong to. For example:<br><code>Object ref = new Integer(0);</code><br><code>String str = (String) ref; // Integer is not String.</code> |
| <code>IllegalArgumentException</code>                             | Attempt to pass an illegal actual parameter value in a method call.                                                                                                                                                 |
| <code>NumberFormatException</code>                                | Indicates problem converting a value to a number, for example, attempt to convert a string with characters that cannot constitute a legal integer, e.g. <code>Integer.parseInt("4u2")</code> .                      |
| <code>IndexOutOfBoundsException</code>                            | The index value is not valid.                                                                                                                                                                                       |
| <code>ArrayIndexOutOfBoundsException</code>                       | The index value is not valid. The index value in an array is either less than 0 or greater than or equal to the array length, e.g. <code>array[array.length]</code> .                                               |
| <code>StringIndexOutOfBoundsException</code>                      | The index value is not valid. The index value in a string is either less than 0 or greater than or equal to the string length, e.g. <code>str.charAt(-1)</code> .                                                   |
| <code>AssertionError</code>                                       | Indicates that the condition in an <code>assert</code> statement has evaluated to the value <code>false</code> , i.e. the assertion failed. See Section 3.4 on page 64 and Section 14.3 on page 406.                |

### BEST PRACTICES

The `AssertionError` should never be caught in a catch block, as it signals that an assertion about the program does not hold, and therefore the logic of the program must be corrected.

## 10.6 Review questions

- Execution handling in Java is built on the principle of \_\_\_\_\_ and \_\_\_\_\_.  
Execution handling in Java is built on the principle of *throw* and *catch*.
- The sequence in which actions are executed determines which exceptions are thrown, and how these will be handled. This is called the \_\_\_\_\_ of the program.



The sequence in which actions are executed determines which exceptions are thrown, and how these will be handled. This is called the *execution behaviour* of the program.

- 3.** Local variables for a method call are stored on a \_\_\_\_\_ during execution.

Local variables for a method call are stored on a *program stack* during execution.

- 4.** If a method A() calls a method B(), the execution of method \_\_\_\_\_ will complete before method \_\_\_\_\_.

If a method A() calls a method B(), the execution of method B() will complete before method A().

- 5.** What is meant by a method being active during execution?

That a method is active means that a call to the method has not completed. There is a stack frame for the method on the program stack.

- 6.** What is wrong with the following try-catch statement?

```
try
 int i = Integer.parseInt("onetwothree");
 catch (NumberFormatException x)
 System.out.println("Not a valid integer.");
```

Parentheses for the try and catch blocks cannot be omitted.

```
try {
 int i = Integer.parseInt("onetwothree");
}
catch (NumberFormatException x) {
 System.out.println("Not a valid integer.");
}
```

- 7.** Which statement are true about the try-catch statement?

- a** A catch block must have a corresponding try block.
  - b** The parameter list of a catch block can be empty, then the catch block can catch any exception.
  - c** The parameter list of a catch block always has only one parameter that specifies the type of exceptions the catch block can catch.
  - d** A try block can contain code that does not throw an exception.
- (a), (c), (d)

- 8.** Which statements are true when an exception occurs?

- a** If the exception is not caught and handled by a catch block, it will be handled by a default exception handler.
- b** If the exception is caught and handled by a catch block, normal execution will continue.
- c** The execution of the try block is terminated, regardless of whether the exception is later caught or not.



- d** Normal execution can never be resumed after an exception has occurred in a try block.

(a), (b), (c)

- 9.** A method can specify exceptions it will rethrow in a \_\_\_\_\_ clause in the method header.

A method can specify exceptions it will rethrow in a throws clause in the method header.

- 10.** Which statement about checked exceptions is true?

- a** Checked exceptions that can be thrown in a method must be listed in a throws clause if the exceptions are not caught and dealt with in the method body.
- b** A method that calls another method that has a throws clause with checked exceptions need not deal with these exceptions.
- c** A method that calls another method that has a throws clause with checked exceptions must explicitly deal with these exceptions.

(a), (c).

- 11.** Which statement is true about checked exceptions?

- a** Checked exceptions that can be thrown in a method, must be listed in a throws clause of the method if these exceptions are not handled by a try-catch statement in the method body.
- b** A method that calls another method that has a throws clause with checked exceptions, need not handle these exceptions.
- c** Since the compiler checks the use of checked exceptions, such exceptions are not handled by a default exception handler.

(a)

A method that calls another method that has a throws clause with checked exceptions, must handle these exceptions. The compiler will check this. Any exception not explicitly caught and handled by the program, will be handled by a default exception handler.

- 12.** Which exceptions will the following code throw during execution?

- a** String str = null; int in = str.length();
- b** Object objRef = new String("aba"); Integer intRef = (Integer) objRef;
- c** int[] array = new int[10]; array[array.length] = 100;
- d** int j = Integer.parseInt("1two1");

(a) `NullPointerException`, the reference str has the value null and therefore does not refer to any object.

(b) `ClassCastException`, Integer cannot be converted to String.

(c) `ArrayIndexOutOfBoundsException`, array[array.length] does not exist.



(d) `NumberFormatException`, "1two1" has characters that are not legal in an integer.

Note that it is the runtime environment that throws these exceptions during execution of the code given above.

- 13.** A method specifies checked exceptions it can throw to its caller in a \_\_\_\_\_ clause.

A method specifies checked exceptions it can throw to its caller in a `throws` clause.

- 14.** Which code will compile?

- a** `try { }`
- b** `try { } catch(Exception x) { }`
- c** `catch(Exception y) { }`
- d** `catch(Exception y) { } try { }`

(b)

## 10.7 Programming exercises

- 1.** Write a class `Temperature` that converts temperature between Fahrenheit (F) and Celsius (C). The formulas for conversion between the two units are:
- $fahrenheit = (9.0 * celsius) / 5.0 + 32.0$
  - $celsius = (fahrenheit - 32.0) * 5.0 / 9.0$

The program reads the degrees from the command line, for example:

```
> java Temperature 50.0 C
50.0 C is 122.0 F
> java Temperature 122.0 F
122.0 F is 50.0 C
```

The program should throw an exception of the type `NumberFormatException` if the number of degrees is not a legal floating-point number. The program should check the data from the command line and give suitable feedback.

- 2.** Write a class `Area` that calculates the area of a rectangle, using the following formula:

$$\text{area} = \text{length} * \text{breadth}$$

The program reads the length (L) and the breadth (B) from the command line, for example:

```
> java Area L 10.5 B 5.0
Length: 10.5
Breadth: 5.0
Area: 52.5
> java Area B 5.0 L 10.5
Length: 10.5
Breadth: 5.0
Area: 52.5
```



The program should throw a `NumberFormatException` if the sides are not legal values.  
The program should check the information read from the command line and give suitable feedback.



## Text file I/O and simple GUI dialogs

### LEARNING OBJECTIVES

By the end of this chapter, you will understand the following:

- Organising values in data records for storing in text files.
- Writing string representations of primitive values and objects to text files.
- Reading characters from text files and converting them to appropriate values.
- Designing simple GUI dialogues using the **JOptionPane** class.
- Creating message dialog boxes to present information to the user.
- Creating input dialog boxes to acquire data from the user.
- Creating confirmation dialog boxes to get confirmation from the user.

### INTRODUCTION

A program must communicate with its environment to obtain the data it requires and to publish the results it computes. Data that a program reads is called *input*, and the data a program writes is called *output*. Input is read from a *source*, and output is written to a *destination* (or *sink*). In the examples so far, the keyboard has functioned as the source and the terminal window has been the destination for data.

This chapter discusses how data can be written to and read from text files, and how to create simple graphical dialog boxes to exchange information between the program and the user.

Chapter 19 further explores reading from and writing to different kinds of files. Appendix E contains customised classes for reading values from the terminal window and creating simple GUI dialog boxes.



## 11.1 File handling

Data that a program manipulates is no longer available after the program terminates, unless the data is stored somewhere and can be read again by the program. Data files can be used for this purpose. A data file offers persistent storage of data and can function as both a source and destination for data.

A *file* refers to a specific storage area on media where data can be stored, for example on a hard drive. Data in a file is stored as a sequence of *bytes* (i.e. an information unit of 8 *bits*), but can be interpreted in different ways during reading and writing. If the data in a file is interpreted as a *sequence of bytes*, the file is called a *binary file*. If the data is interpreted as a *sequence of characters*, the file is called a *text file*.

In general the way in which primitive data values are stored is platform-specific. It is not at all certain that a binary file created on one platform can be moved to another and be interpreted correctly. Java solves the problem of moving binary data by defining the *size* of primitive data types. The size specifies how many bytes are needed to represent values of a particular primitive data type (see Table 11.1). A binary file containing such values will be interpreted correctly by a Java program on any platform.

A *char* value in a Java program always occupies two bytes, but this might not be the case when the character is stored in a text file. How many bytes a character occupies in a file depends on the *character encoding* used to store the characters in the file. Java provides classes that facilitate reading and writing characters from text files and take into account the encoding used for storing the characters.

A text editor will interpret the contents of a text file as a sequence of characters. The compiler `javac` interprets the contents of a Java source file as a long string of characters, but creates a binary file that contains the resulting Java byte code. Some character encodings are quite common, for example UTF-8 and ISO 8859-1, so that text files are usually interpreted correctly when moved between platforms.

**TABLE 11.1 Default size of primitive data types**

| Primitive data type  | Size in bytes |
|----------------------|---------------|
| <code>boolean</code> | 1             |
| <code>char</code>    | 2             |
| <code>int</code>     | 4             |
| <code>long</code>    | 8             |
| <code>float</code>   | 4             |
| <code>double</code>  | 8             |



## Data records

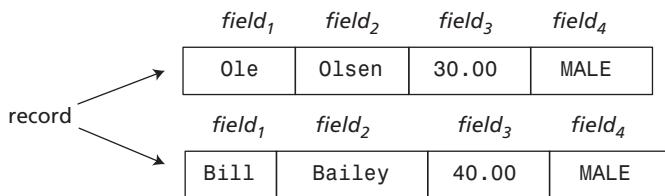
A company wants to store information about their employees in a text file so that the information can be reused. The class `Employee` is shown at (1) in Program 11.1. We would like to store the following information about an employee in a text file:

```
String firstName; // Field variable 1
String lastName; // Field variable 2
double hourlyRate; // Field variable 3
Gender gender; // Field variable 4
```

It is quite common to store information as data records. A *record* consists of one or more data fields. A *data field* in a record usually contains a primitive value, but we allow a string literal as well. A data record for an employee will consist of four data fields, as illustrated in Figure 11.1, corresponding to the four field variables in the `Employee` class. Note that the string representation of an enum constant is its name, as depicted for the gender field in Figure 11.1.

Program 11.1 also shows the class `PersonnelRegister` at (2), which uses an array to keep track of all employees. In the next section we use the classes from Program 11.1 to write information in the `Employee` object as records in a text file, to read back records from the text file, and to create appropriate `Employee` objects from the records read.

**FIGURE 11.1 Data records**



**PROGRAM 11.1 Registering employees**

```
// Constants to represent gender
enum Gender {FEMALE, MALE};

// Class representing an employee
class Employee { // (1)

 // Static variable
 final static double NORMAL_WORKWEEK = 37.5;

 // Field variables
 String firstName;
 String lastName;
 double hourlyRate;
 Gender gender;
```



```
// Constructors
Employee() { }
Employee(String firstName, String lastName,
 double hourlyRate, Gender gender) {
 this.firstName = firstName;
 this.lastName = lastName;
 this.hourlyRate = hourlyRate;
 this.gender = gender;
}

// Determines whether an employee is female.
boolean isFemale() { return (this.gender == Gender.FEMALE); }

// Computes the salary of an employee, based on the number of hours
// worked during the week.
double computeSalary(double numOfHours) {
 assert numOfHours >= 0 : "Number of hours must be >= 0";
 double weeklySalary = hourlyRate * Employee.NORMAL_WORKWEEK;
 if (numOfHours > Employee.NORMAL_WORKWEEK) {
 weeklySalary += 2.0 * hourlyRate * (numOfHours - NORMAL_WORKWEEK);
 }
 return weeklySalary;
}

// Return string representation of the field values of an employee.
public String toString() {
 return String.format(
 "First name: %-6s Last name: %-8s Hourly rate: %6.2f Gender: %-6s",
 this.firstName, this.lastName, this.hourlyRate, this.gender);
}

// Personnel register of employees
class PersonnelRegister { // (2)

 final static int MAX_NUM_EMPLOYEES = 100;

 // Fields
 Employee[] employees;
 int numOfEmployees;
 int numOfFemales;

 // Default constructor
 PersonnelRegister() {
 employees = new Employee[MAX_NUM_EMPLOYEES];
 }

 // Create a personnel register from an array of employees.
 PersonnelRegister(Employee[] suppliedEmployeeArray) {
```

```

employees = new Employee[MAX_NUM_EMPLOYEES];
for(int i = 0; i < suppliedEmployeeArray.length; i++) {
 registerEmployee(suppliedEmployeeArray[i]);
}
}

// Create a personnel register with field values from parameters.
PersonnelRegister(Employee[] employees, int numEmployees,
 int numFemales) {
 this.employees = employees;
 this.numEmployees = numEmployees;
 this.numFemales = numFemales;
}

// Register an employee in the employee array.
void registerEmployee(Employee newEmployee) {
 assert numEmployees != employees.length:
 "No room for more employees.";
 employees[numEmployees] = newEmployee;
 numEmployees++;
 if (newEmployee.gender == Gender.FEMALE) {
 numFemales++;
 }
}

// Selectors
int getNumEmployees() { return numEmployees; }
int getNumFemales() { return numFemales; }
Employee[] getEmployeeArray() { return employees; }

// Return employee with specified index.
Employee getEmployee(int index) {
 assert 0 <= index && index <= numEmployees :
 "Index not valid";
 return employees[index];
}

// Replace an employee at index with another employee.
void replaceEmployee(int index, Employee newEmployee) {
 assert 0 <= index && index <= numEmployees :
 "Index not valid";
 employees[index] = newEmployee;
}

// Compute percentage of females in the company.
double getFemalePercentage() {
 assert numEmployees > 0 : "Personnel register is empty.";
 return 100.0 * numFemales / numEmployees;
}

```



```
// Returns statistics about the company.
public String toString() {
 return String.format("The company has %d employees, " +
 " where %.2f%% are women.",
 getNumOfEmployees(), getFemalePercentage());
}
}
```

---

11



## 11.2 Text files

A text file contains lines of text. Each *text line* consists of a sequence of characters and is terminated by a *line terminator string*. This line terminator string is platform-specific. For example, it is the string "\r\n" in Windows, but consists of a single character '\n' in Unix. However, the methods for file handling in Java ensure that the line terminator string is interpreted correctly.

The `java.io` package provides extensive support for reading data from various sources and writing data to various destinations. Here we concentrate on one approach to reading and writing text files. Essentially, there are four steps to file handling, which we will discuss in detail in subsequent subsections:

- 1 We need to *open* the file, thereby creating a connection between the program and the file.
- 2 We need to choose the appropriate class to convert the values, depending on whether we are reading from or writing to a file.
- 3 Data can now be written to or read from the designated file.
- 4 We *close* the file when we are finished with it, thereby freeing any resources that were used for handling the file.

We would like to store information about employees in a text file. Information about an employee is stored as a data record consisting of the field values in an `Employee` object, translated to a sequence of characters, and each record is terminated by the line terminator string. When reading, we can read a text line that corresponds to a record, and create an employee with the extracted field values.

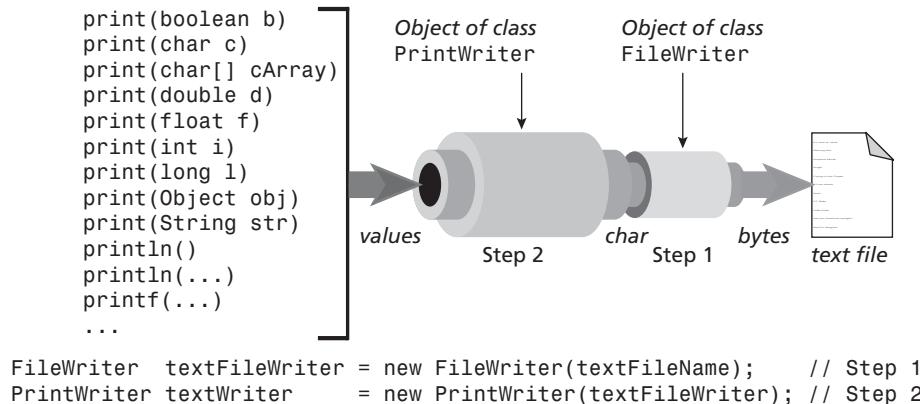
### Writing to text files

We need the following classes to write to a text file (see Figure 11.2):

- The class `java.io.FileWriter` opens the file and ensures that the characters are stored in the default character encoding for the platform.
- The class `java.io.PrintWriter` provides methods for converting values to their respective string representations.

FIGURE 11.2

Writing to a text file



Here's how the four steps for file handling apply to writing to text files. The procedure for writing string representations of values to text files is illustrated by Figure 11.2 and Program 11.2:

- 1 Create a `FileWriter` object to open the file for writing:

```
FileWriter textFileWriter = new FileWriter(dataFileName); // (5)
```

The constructor accepts the name of the file. If a file with the designated name does not exist, a new file with this name is created. If the file with the designated name exists, it is reset, meaning that the writing will start at the beginning of the file and any previous content will be overwritten. The call to the `FileWriter` constructor can throw an `IOException` if the file cannot be opened for some reason.

The class also provides a constructor that allows appending to an existing file:

```
FileWriter textFileWriter = new FileWriter(dataFileName, append);
```

If the `boolean` value of the parameter `append` is `true`, this constructor will open the file for appending. Otherwise writing will start from the beginning of the file.

- 2 Create a `PrintWriter` object that is connected to the `FileWriter` from step 1:

```
PrintWriter textWriter = new PrintWriter(textFileWriter); // (6)
```

Now the `PrintWriter` delivers characters to the `FileWriter`, which in turn ensures that the characters it receives are correctly stored as bytes in the file, using the default character encoding for the platform.

The class `PrintWriter` also provides a constructor that accepts a file name as parameter:

```
PrintWriter textWriter = new PrintWriter(dataFileName);
```

This constructor creates the underlying file writer to encode the characters for storing in the specified file. This constructor also creates the file if necessary, and always opens it for writing from the beginning of the file.

- 3 Write text representations of values using the `print` methods of the `PrintWriter` class. For example, the method `writeEmployeeData()` at (10) in Program 11.2 uses the `printf()` method in the `PrintWriter` class to write the values of the different fields in an `Employee` object:

```
textWriter.printf(
```

```
%s" + FIELD_TERMINATOR + "%s" + FIELD_TERMINATOR +
 "%.2f" + FIELD_TERMINATOR + "%s%n",
 employee.firstName, employee.lastName,
 employee.hourlyRate, employee.gender);
```

Since all primitive values are written as characters, it is important to mark the *end* of each field in the record, so that the values can be interpreted correctly when read from the file. We terminate all fields with the field terminator character, except for the last field, which is terminated by the line terminator string. The format specification "%n" in the format string ensures that the correct platform-specific line terminator string is printed.

We have used comma (,) as the field terminator character in a record. This format is called the *Comma Separated Values (CSV) file format*. It is quite popular, and is used by many spreadsheet programs to import and export values.

- 4 Finish by closing the file. This is done by calling the `close()` method of the `PrintWriter`:

```
textWriter.close(); // (9)
```

Calling the `close()` method is strongly recommended, as this ensures that not only is the connection between the program and the file closed, but also that any resources that were used for handling the file are freed and that no data is lost.

Program 11.2 uses the classes from Program 11.1, and at (2) calls the method `writeAllEmployeesToFile()` to write the employee information to the specified file. The method creates the necessary writer objects at (5) and (6), as explained above. The method first writes the number of records to be stored in the text file at (7), and at (8) calls the method `writeEmployeeData()` to write the information about each employee. The method `writeEmployeeData()` at (10) writes this information as explained in step 3 above. The file `employeeFile.txt` will contain the following four text lines, which can be verified by opening it in a text editor:

```
3
Ole,Olsen,30.00,MALE
Bill,Bailey,40.00,MALE
Liv,Larsen,50.00,FEMALE
```

The contents of the file are read into a personnel register at (3), and subsequently the contents of the personnel register are printed to the terminal window.

## PROGRAM 11.2 Text files

```
import java.io.IOException;

public class CompanyAdmin {

 public static void main(String[] args) throws IOException { // (1)
 // Create an array of employees.
 Employee[] employeeInfo = {
 new Employee("Ole", "Olsen", 30.00, Gender.MALE),
 new Employee("Bill", "Bailey", 40.00, Gender.MALE),
 new Employee("Liv", "Larsen", 50.00, Gender.FEMALE)}
```



```
};

// Create a personnel register.
PersonnelRegister register = new PersonnelRegister(employeeInfo);
// Create a company.
CompanyUsingTextFiles company = new CompanyUsingTextFiles(register);

// Print employee info in personnel register to a text file.
company.writeAllEmployeesToTextFile("employeeFile.txt"); // (2)

// Read employee info from a text file.
System.out.println("Read from text file.");
company.readAllEmployeesFromTextFile("employeeFile.txt"); // (3)

// Print employee info to the terminal window.
company.printAllEmployeesToTerminalWindow();
company.printReport();
}

}

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

class CompanyUsingTextFiles {

 final static char FIELD_TERMINATOR = ','; // Comma

 // Field
 private PersonnelRegister register;

 // Constructors
 CompanyUsingTextFiles() {
 register = new PersonnelRegister();
 }

 CompanyUsingTextFiles(PersonnelRegister register) {
 this.register = register;
 }

 // Print statistics
 void printReport() {
 System.out.println(register);
 }

 void printAllEmployeesToTerminalWindow() {
 Employee[] employees = register.getEmployeeArray();
 int numOfEmployees = register.getNumOfEmployees();
 }
}
```



```
System.out.println("Number of employees: " + numOfEmployees);
for (int i = 0; i < numOfEmployees; i++) {
 System.out.println(employees[i]);
}

void writeAllEmployeesToTextFile(String dataFileName)
 throws IOException { // (4)

 FileWriter textFileWriter = new FileWriter(dataFileName); // (5)
 PrintWriter textWriter = new PrintWriter(textFileWriter); // (6)

 // Write the number of employees in the register.
 int numOfEmployees = register.getNumOfEmployees();
 textWriter.println(numOfEmployees); // (7)

 // Write info about each employee.
 Employee[] employees = register.getEmployeeArray();
 for (int i = 0; i < numOfEmployees; i++) {
 writeEmployeeData(textWriter, employees[i]); // (8)
 }

 textWriter.close(); // (9)
}

void writeEmployeeData(PrintWriter textWriter,
 Employee employee) // (10)
// Fields separated by a field terminator character.
textWriter.printf(
 "%s" + FIELD_TERMINATOR + "%s" + FIELD_TERMINATOR +
 "%.2f" + FIELD_TERMINATOR + "%s%n",
 employee.firstName, employee.lastName,
 employee.hourlyRate, employee.gender);
}

void readAllEmployeesFromTextFile(String dataFileName)
 throws IOException { // (11)

 FileReader textFileReader = new FileReader(dataFileName); // (12)
 BufferedReader textReader = new BufferedReader(textFileReader); // (13)

 // Read how many employees are in the text file.
 String firstLine = textReader.readLine(); // (14)
 int totalNumOfEmployees = Integer.parseInt(firstLine);

 // Create a personnel register.
 register = new PersonnelRegister();

 // Read info about all employees in the file.
 for (int i = 0; i < totalNumOfEmployees; i++) {
```



```

// Read an employee.
Employee employee = readEmployeeData(textReader); // (15)
// Register the employee.
register.registerEmployee(employee);
}

textReader.close(); // (16)
}

Employee readEmployeeData(BufferedReader textReader)
 throws IOException { // (17)
// Read a record, i.e. a text line.
String record = textReader.readLine(); // (18)

// Find the index of the field terminator characters in the record.
int fieldTerminatorIndex1 = record.indexOf(FIELD_TERMINATOR); // (19)
int fieldTerminatorIndex2 = record.indexOf(FIELD_TERMINATOR,
 fieldTerminatorIndex1 + 1);
int fieldTerminatorIndex3 = record.indexOf(FIELD_TERMINATOR,
 fieldTerminatorIndex2 + 1);

// Extract the values of the fields from the data record.
String firstName = record.substring(0, fieldTerminatorIndex1); // (20)
String lastName = record.substring(fieldTerminatorIndex1 + 1,
 fieldTerminatorIndex2);

String doubleStr = record.substring(fieldTerminatorIndex2 + 1,
 fieldTerminatorIndex3);
double hourlyRate = Double.parseDouble(doubleStr); // (21)

String genderStr = record.substring(fieldTerminatorIndex3 + 1);
Gender gender;
if (genderStr.equals(Gender.MALE.toString())) { // (22)
 gender = Gender.MALE;
} else {
 gender = Gender.FEMALE;
}

return new Employee(firstName, lastName, hourlyRate, gender);
}
}

```

Program output:

```

Number of employees: 3
First name: Ole Last name: Olsen Hourly rate: 30.00 Gender: MALE
First name: Bill Last name: Bailey Hourly rate: 40.00 Gender: MALE
First name: Liv Last name: Larsen Hourly rate: 50.00 Gender: FEMALE
The company has 3 employees, where 33.33% are women.

```



### Exception handling when writing to a text file

At (5) in Program 11.2 the constructor call in the `writeAllEmployeesToTextFile()` method can throw an `IOException`. This is an exception that cannot be ignored, i.e. it is a checked exception. The method is therefore declared with a `throws` clause in its header:

```
void writeAllEmployeesToTextFile(String dataFileName)
 throws IOException { // (4)
 ...
 FileWriter textFileWriter = new FileWriter(dataFileName); // (5)
 ...
}
```

Since the `main()` method calls the `writeAllEmployeesToTextFile()` method, the `main()` method *must* also deal with the `IOException` that can result from the call to the `writeAllEmployeesToTextFile()` method. The `main()` method also uses the `throws` clause to rethrow the exception:

```
public static void main(String[] args) throws IOException { // (1)
 ...
 company.writeAllEmployeesToTextFile("employeeFile.txt"); // (2)
 ...
}
```

If an `IOException` is thrown in the `writeAllEmployeesToTextFile()` method at runtime, it will be passed to the method that called the `writeAllEmployeesToTextFile()` method. In Program 11.2, this is the `main()` method. This method rethrows the exception, resulting in the JVM having to deal with it, stopping program execution and printing information about the exception to the terminal window. Removing any of the `throws` clauses in the two methods above will result in a compile time error.

## Reading from text files

We need the following classes to read from a text file (see Figure 11.3 on page 306):

- The class `java.io.FileReader` opens the file and ensures that the bytes in the file are interpreted correctly as characters, according to the default character encoding for the platform.
- The class `java.io.BufferedReader` provides the ability to read a whole line of text, rather than a character at a time. We can achieve a significant improvement in the performance of our program if it is possible to store characters temporarily in memory, and move several characters at a time from the external media to this temporary storage. This approach minimises the number of accesses necessary to read characters from the external media. Such a temporary storage is called a *buffer*, and it is used by the `BufferedReader` class.

The procedure for reading text from files and converting it to values of appropriate data types is outlined below (see also Figure 11.3, Figure 11.4 and Program 11.2):

- 1 Create a `FileReader` object to open the file for reading:

```
FileReader textFileReader = new FileReader(dataFileName); // (12)
```



The constructor accepts the name of the file to open. If a file with the designated name does not exist, an exception is thrown. If the file with the designated name exists, reading will start at the beginning of this file.

The call to the `FileReader` constructor will also throw an `IOException` if for some reason the file cannot be opened.

- 2** Create a `BufferedReader` object that is connected to the `FileReader` from step 1:

```
BufferedReader textReader = new BufferedReader(textFileReader); // (13)
```

The `BufferedReader` class provides the method `readLine()` for reading a whole line of text efficiently, as explained above. The `FileReader` reads the bytes in the file as characters, and the `BufferedReader` can be used to read these as lines of text.

- 3** Read one text line at a time using the `readLine()` method of the `BufferedReader` class. This method returns a `String` object:

```
String record = textReader.readLine(); // (18)
```

Each successive call to the `readLine()` method reads the next line of text in the file. The `readLine()` method returns a null when there are no more characters to read, i.e. when we have reached the *end of the file* (often abbreviated as *EOF*).

If it is not the end of file, the string returned will contain the current line of text. The line terminator string at the end of each text line is *not* a part of the returned string.

The `readLine()` method can throw an `IOException` if it cannot read from the file.

In our case, the returned string corresponds to a record. We can now extract the characters that comprise each field value from the record, as shown in Figure 11.4. Since each field is terminated by the field terminator character, we first find the index of this character in the record (step 1 in Figure 11.4):

```
int fieldTerminatorIndex1 = record.indexOf(FIELD_TERMINATOR); // (19)
```

Given this index, we can extract the substring that comprises the characters in the field (step 2 in Figure 11.4):

```
String firstName = record.substring(0, fieldTerminatorIndex1); // (20)
```

If the field value is an integer or another type of value, we must convert the substring to the corresponding value. Examples below show how we can convert a string to a double value or to an enum constant (step 3 in Figure 11.4):

```
String doubleStr = record.substring(fieldTerminatorIndex2 + 1,
 fieldTerminatorIndex3);
```

```
double hourlyRate = Double.parseDouble(doubleStr); // (21)
```

```
String genderStr = record.substring(fieldTerminatorIndex3 + 1);
```

```
Gender gender;
```

- if** (genderStr.equals(Gender.MALE.toString())) { // (22)
 gender = Gender.MALE;
 } **else** {
 gender = Gender.FEMALE;
 }

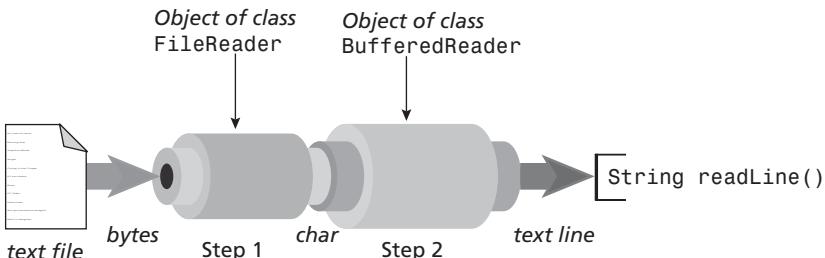
An alternative solution for reading field values is given in Exercise 11.1.

- 4** Close the file. This is done by calling the `close()` method of the `BufferedReader` class:

textReader.close(); // (9)

The call ensures that all resources connected with reading from the file are freed. The `close()` method will throw an `IOException` if it cannot close the file.

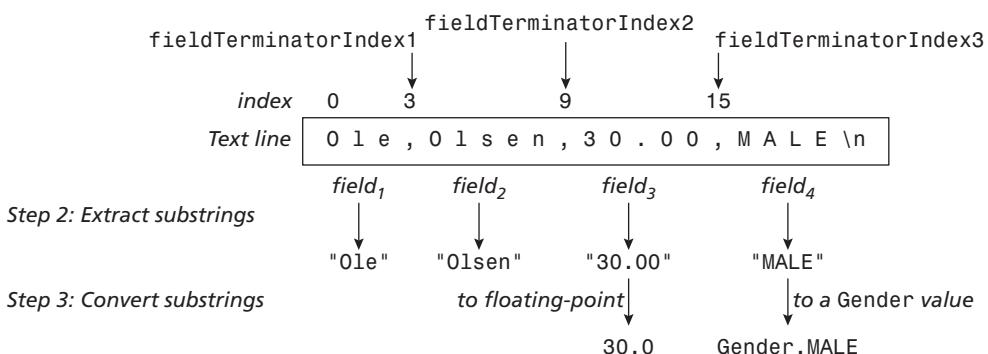
**FIGURE 11.3** Reading from a text file



```
FileReader textFileReader = new FileReader(textFileName); // Step 1
BufferedReader textReader = new BufferedReader(textFileReader); // Step 2
```

**FIGURE 11.4** Reading records and converting fields to appropriate values

*Step 1: Find the index of the field terminator characters*



Program 11.2 on page 300 illustrates reading from a text file. The `main()` method in the `CompanyAdmin` class calls the method `readAllEmployeesFromTextFile()` in the `CompanyUsingTextFiles` class at (2), passing it the name of the file as parameter. This method creates the necessary reader objects at (12) and (13), as explained in steps 2 and 3 above. At (14), the method reads the number of records stored in the text file:

```
String firstLine = textReader.readLine(); // (14)
int totalNumOfEmployees = Integer.parseInt(firstLine);
```

At (15), the `readAllEmployeesFromTextFile()` method calls the `readEmployeeData()` method to read the information about each employee. The `readEmployeeData()` method declared at (17) reads the record of an employee from the text file and extracts the field values to create an `Employee` object, as explained in step 4 above. To verify the information read from the text file, the `main()` method prints it to the terminal window.



Note that the data is written sequentially from the beginning of the file, and must also be read sequentially from the beginning of the file. That is why such files are called *sequential files*. The interpretation of the characters as different types of values when reading from the file is the responsibility of the program.

### **Exception handling when reading from a text file**

Exception handling for reading from a text file is analogous to that for writing to a text file. Calls to the method `readLine()` at (18) can throw an `IOException`, therefore the method `readEmployeeData()` is declared with a `throws` clause to deal with this checked exception:

```
Employee readEmployeeData(BufferedReader textReader)
 throws IOException { // (17)
 ...
 String record = textReader.readLine(); // (18)
 ...
}
```

The `readAllEmployeesFromTextFile()` method has several lines of code that can throw an `IOException`, and is therefore declared with a `throws` clause:

```
void readAllEmployeesFromTextFile(String dataFileName)
 throws IOException { // (11)
 ...
 FileReader textFileReader = new FileReader(dataFileName); // (12)
 ...
 Employee employee = readEmployeeData(textReader); // (15)
 ...
 textReader.close(); // (16)
}
```

Since the `main()` method calls the `readAllEmployeesFromTextFile()` method at (3), the `main()` method must also be declared with a `throws` clause:

```
public static void main(String[] args) throws IOException { // (1)
 ...
 company.readAllEmployeesFromTextFile("employeeFile.txt"); // (3)
 ...
}
```

Note the following chain of calls: the `main()` method calls the `readAllEmployeesFromTextFile()` method, which in turn calls the `readEmployeeData()` method. They all have to deal with any `IOException`. Removing any of the `throws` clauses will result in a compile time error.

## **11.3 Simple GUI dialogue design**

The `java.swing.JOptionPane` class is useful for creating simple graphical user interfaces (GUIs). This section presents examples of how this class can be used for this purpose.



## Overview of the `JOptionPane` class

The `java.swing.JOptionPane` class provides predefined simple GUI dialog boxes that can be customised for exchange of input and output with the user.

We will look at the designing of dialog boxes for three purposes, using the `JOptionPane` class:

- 1 How information can be presented to the user.
- 2 How the user can enter input required by the program.
- 3 How the program can ask the user to confirm information.

The `JOptionPane` class defines three static methods (see Table 11.2) that we will use to create dialog boxes for the purposes mentioned above. These methods are `showTypeDialog`, where *Type* can be replaced by `Message`, `Input` or `Confirm`, depending on which type of dialog box is required. These methods have many parameters in common (see Table 11.3). Some of the parameters need not be specified, in which case default values are used.

All these dialog boxes are *modal*, meaning that if a program displays such a box, all user input is directed to it. Only when interaction with the dialog box has concluded is user input directed back to the program.

Many of the examples and programming exercises in the previous chapters can be modified to use dialog boxes. Some suggestions are given as programming exercises at the end of this chapter.

### ***Ending programs that use GUI dialog boxes***

A program usually ends when the `main()` method has finished executing. However, when using GUI components, the JVM starts an additional task (a *thread*) to monitor the interaction between the program and the GUI. Even though the `main()` method has ended, this GUI task continues in the JVM and must be stopped explicitly. This is done by calling the `exit()` method in the `System` class:

```
System.exit(0);
```

This method is called when we want to stop *all* execution, usually as the last statement in the `main()` method. The method requires an integer value as parameter, which is passed back to the operative system to indicate whether the program executed successfully (the value 0) or not (a non-zero value).



**TABLE 11.2** Summary of methods from the **JOptionPane** class

| Method                                                                                                                               | Description                                                                                                                                                                                                                                                                                              |
|--------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>static void showMessageDialog( Component parentComponent, Object message)</code>                                               | The <code>showMessageDialog()</code> method is used to present information to the user.<br>It always shows an "OK" button.                                                                                                                                                                               |
| <code>static void showMessageDialog( Component parentComponent, Object message, String title, int messageType)</code>                | The first method uses the title "Message" in the title bar of the dialog box, and the message type is <code>JOptionPane.INFORMATION_MESSAGE</code> (see Table 11.4).                                                                                                                                     |
| <code>static String showInputDialog( Object message)</code>                                                                          | The <code>showInputDialog()</code> method is used to ask the user for input. Whatever is entered in the text field of the dialog box is returned as a string.<br>It always shows an "OK" button and a "Cancel" button.<br>If the dialog box has been cancelled, the <code>null</code> value is returned. |
| <code>static String showInputDialog( Component parentComponent, Object message, String title, int messageType)</code>                | The first and the second method use the title "Input" in the title bar of the dialog box, and the message type is <code>JOptionPane.QUESTION_MESSAGE</code> (see Table 11.4).                                                                                                                            |
| <code>static int showConfirmDialog( Component parentComponent, Object message)</code>                                                | The <code>showConfirmDialog()</code> method is used to ask the user a question in order to confirm some information. The integer value returned by the methods indicates the action taken by the user (see Table 11.5).                                                                                  |
| <code>static int showConfirmDialog( Component parentComponent, Object message, String title, int optionType)</code>                  | The first method uses the title "Select an option" in the title bar of the dialog box, the option type is <code>JOptionPane.YES_NO_CANCEL_OPTION</code> (see Table 11.6) and the message type is <code>JOptionPane.QUESTION_MESSAGE</code> (see Table 11.4).                                             |
| <code>static int showConfirmDialog( Component parentComponent, Object message, String title, int optionType, int messageType)</code> | The option type (Table 11.6) specifies which combination of an "OK" button, a "Yes" button, a "No" button and a "Cancel" button can be used in a confirmation dialog box.                                                                                                                                |

**TABLE 11.3** Common parameters for the `showTypeDialog()` methods

| Parameter name                         | Description                                                                                                                                                                                                                                                                   |
|----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Component parentComponent</code> | If the value is <code>null</code> or no value is specified, the dialog box is placed in the middle of the screen, otherwise it is placed below the parent component.                                                                                                          |
| <code>Object message</code>            | Specifies what should be presented to the user, for example, a message or a prompt. Usually this object is a string. If that is not the case, the <code>toString()</code> method of the object is called to create a string representation, which is shown in the dialog box. |
| <code>String title</code>              | Specifies the title to set in the title bar of the dialog box.                                                                                                                                                                                                                |
| <code>int messageType</code>           | This value specifies which icon should be used in the dialog box. These values are specified in Table 11.4.                                                                                                                                                                   |

**TABLE 11.4** Specifying the message type in a dialog box

| Message type                     | Description                                                                                                      |
|----------------------------------|------------------------------------------------------------------------------------------------------------------|
| <code>ERROR_MESSAGE</code>       | These constants in the <code>JOptionPane</code> class represent standard icons that can be used in a dialog box. |
| <code>INFORMATION_MESSAGE</code> |                                                                                                                  |
| <code>WARNING_MESSAGE</code>     |                                                                                                                  |
| <code>QUESTION_MESSAGE</code>    |                                                                                                                  |
| <code>PLAIN_MESSAGE</code>       |                                                                                                                  |

**TABLE 11.5** Values returned by the `showConfirmDialog()` method

| Constants                  | Which action the user has performed            |
|----------------------------|------------------------------------------------|
| <code>YES_OPTION</code>    | Clicked on the “Yes” button.                   |
| <code>NO_OPTION</code>     | Clicked on the “No” button.                    |
| <code>CANCEL_OPTION</code> | Clicked on the “Cancel” button.                |
| <code>OK_OPTION</code>     | Clicked on the “OK” button.                    |
| <code>CLOSED_OPTION</code> | Clicked on the close button of the dialog box. |

TABLE 11.6 Specifying the option type in the `showConfirmDialog()` method

| Constants for option buttons | Option buttons shown in the dialog box |
|------------------------------|----------------------------------------|
| DEFAULT_OPTION               | "OK" button only                       |
| YES_NO_OPTION                | "Yes" and "No" buttons                 |
| YES_NO_CANCEL_OPTION         | "Yes", "No" and "Cancel" buttons       |
| OK_CANCEL_OPTION             | "Yes" and "Cancel" buttons             |

### Message dialogs – presenting information to the user

This type of dialog box usually consists of a message to the user and an "OK" button that the user can click after having read the message. The method `showMessageDialog()` is used for such dialog boxes, as illustrated in Program 11.3. The dialog boxes created at (1) and (2) are shown in Figure 11.5.

11



FIGURE 11.5 Dialog windows with the `showMessageDialog()` method



```
JOptionPane.showMessageDialog(// (1)
 null,
 "How ya'doin!");
```

(a)

```
JOptionPane.showMessageDialog(// (2)
 null,
 "You have hit the jackpot!",
 "Important Message",
 JOptionPane.WARNING_MESSAGE);
```

(b)

PROGRAM 11.3 Using the `showMessageDialog()` method

```
import javax.swing.JOptionPane;

public class MessageDialog {
 public static void main(String[] args) {

 JOptionPane.showMessageDialog(// (1)
 null,
 "How ya'doin!" // Message
);

 JOptionPane.showMessageDialog(// (2)
 null,
 "You have hit the jackpot!",
 "Important Message", // Title in the window
 JOptionPane.WARNING_MESSAGE
);
 }
}
```

```

 JOptionPane.WARNING_MESSAGE // Message type
);
}

System.exit(0); // (3) Terminate the program.
}
}

```

---

11



## Input dialogs – reading data from the user

This type of dialog box usually consists of a text field where the user can enter text, and two buttons: an “OK” button and a “Cancel” button to deliver the input entered in the text field to the program, or to cancel the dialog box without supplying any input, respectively. The method `showInputDialog()` provides this functionality. Program 11.4 creates input dialog boxes at (1), (2) and (4), and they are illustrated in Figure 11.6.

The `showInputDialog()` method returns the contents of the text field as a string. This string may be explicitly converted to another type of value if necessary, as shown at (3) in Program 11.4. At (5), the input is presented to the user in a message dialog box.

**FIGURE 11.6** Dialog windows with the `showInputDialog()` method



```
JOptionPane.showInputDialog(// (1) JOptionPane.showInputDialog(// (2)
 "Name:" null,
); "Zipcode:"
);
```

(a)

(b)



```
JOptionPane.showInputDialog(// (4)
 null, "City:",
 "Input data", JOptionPane.PLAIN_MESSAGE);

```

(c)

## PROGRAM 11.4 Using the `showInputDialog()` method

```
import javax.swing.JOptionPane;

public class InputDialog {
 public static void main(String[] args) {

 String name = JOptionPane.showInputDialog(// (1)
 "Name:", // Prompt
);

 String zipcodeStr = JOptionPane.showInputDialog(// (2)
 null, // No parent window
 "Zipcode:", // Prompt
);
 int zipcode = Integer.parseInt(zipcodeStr); // (3)

 String city = JOptionPane.showInputDialog(// (4)
 null, // No parent window
 "City:", // Prompt
 "Input data", // Title in the window
 JOptionPane.PLAIN_MESSAGE // Message type
);

 JOptionPane.showMessageDialog(// (5) Message dialogue
 null,
 name + "\n" + zipcode + " " + city,
 "Information",
 JOptionPane.PLAIN_MESSAGE
);

 System.exit(0);
 }
}
```

11



## Confirmation dialogs – getting confirmation from the user

This type of dialog usually consists of a question about some fact that the user must confirm. The confirmation dialog box usually has two buttons, a “Yes” button and a “No” button, to reply to the question. The method `showConfirmDialog()` provides this functionality. Interpretation of the value returned by the method, which indicates the action taken by the user, is shown in Table 11.5 on page 310.

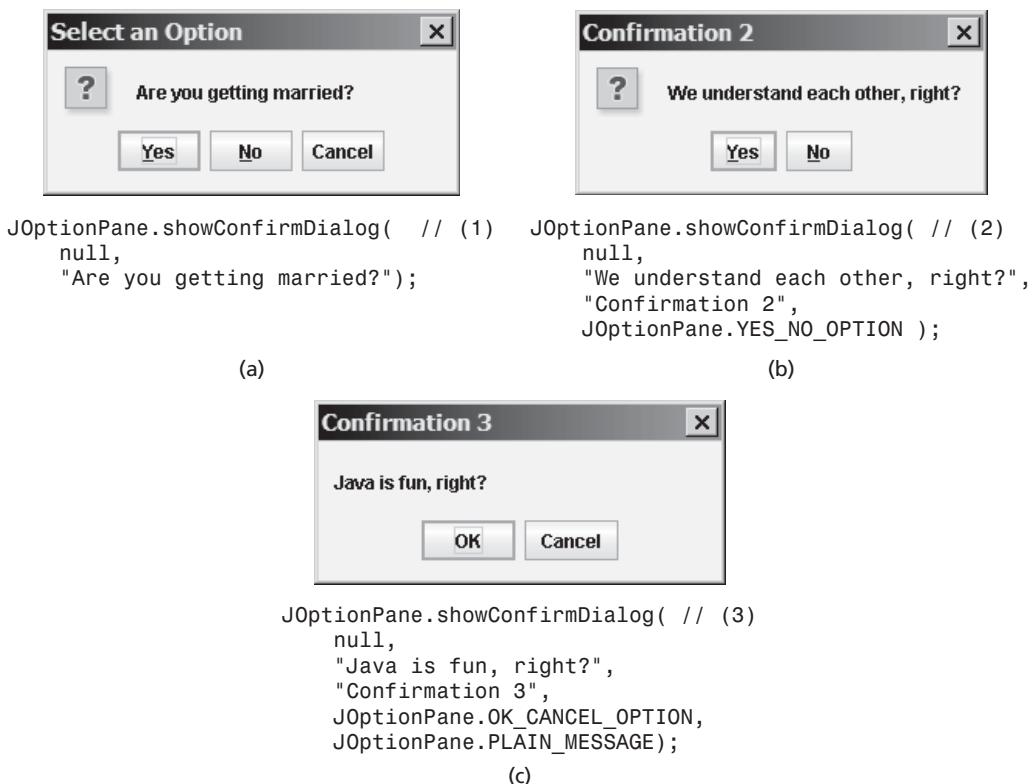
The `showConfirmDialog()` method can also take a parameter that specifies the *option type*. The class `JOptionPane` defines valid values for the option type that indicate what combination of “Yes”, “No” and “Cancel” buttons will be used in the confirmation dialog box (see Table 11.6).



Program 11.5 provides examples of dialog boxes for confirmation of miscellaneous information at (1), (2) and (3). These dialog boxes are shown in Figure 11.7.

It is also possible to change the text that is shown on the “Yes”, “No” and “Cancel” buttons, but we leave it to the reader to find the details in the Java standard library.

**FIGURE 11.7** Dialog windows with the **showConfirmDialog()** method



**PROGRAM 11.5** Using the **showConfirmDialog()** method

```

import javax.swing.JOptionPane;

public class ConfirmDialog {
 public static void main(String[] args) {

 int answer1 = JOptionPane.showConfirmDialog(// (1)
 null, // No parent window
 "Are you getting married?" // Prompt
); // YES, NO and CANCEL buttons
 String answerStr1 = null;
 switch (answer1) {
 case JOptionPane.YES_OPTION:
 answerStr1 = "Congratulations!";

```

```

 break;
 case JOptionPane.NO_OPTION:
 answerStr1 = "All right.";
 break;
 case JOptionPane.CANCEL_OPTION:
 case JOptionPane.CLOSED_OPTION:
 answerStr1 = "Sorry I asked.";
 break;
 default:
 assert false;
}
JOptionPane.showMessageDialog(null, answerStr1);

int answer2 = JOptionPane.showConfirmDialog(// (2)
 null, // No parent window
 "We understand each other, right?", // Prompt
 "Confirmation 2", // Title in the window
 JOptionPane.YES_NO_OPTION // YES and NO buttons
);
String answerStr2 = null;
switch (answer2) {
 case JOptionPane.YES_OPTION:
 answerStr2 = "Good!";
 break;
 case JOptionPane.NO_OPTION:
 case JOptionPane.CLOSED_OPTION:
 answerStr2 = "All right.";
 break;
 default:
 assert false;
}
JOptionPane.showMessageDialog(null, answerStr2);

int answer3 = JOptionPane.showConfirmDialog(// (3)
 null, // No parent window
 "Java is fun, right?", // Prompt
 "Confirmation 3", // Title in the window
 JOptionPane.OK_CANCEL_OPTION, // OK and CANCEL buttons
 JOptionPane.PLAIN_MESSAGE // Message type
);
String answerStr3 = null;
switch (answer3) {
 case JOptionPane.OK_OPTION:
 answerStr3 = "We agree!";
 break;
 case JOptionPane.CANCEL_OPTION:
 case JOptionPane.CLOSED_OPTION:
 answerStr3 = "Pity you won't confirm.";
 break;
 default:

```



```

 assert false;
 }
JOptionPane.showMessageDialog(null, answerStr3);

 System.exit(0);
}
}

```

---

11



## 11.4 Review questions

1. Which of these files would you characterise as a text file?

- a A file with Java byte code.
  - b A file with Java source code.
  - c A file with image data.
  - d A file with audio data.
  - e A file with HTML (Hypertext Markup Language) content.
- (b), (e).

Only (b) and (e) are text files. The others are binary files.

2. Which statements about sequential files are true?

- a When we open a sequential file for writing, we can specify whether writing will start immediately after the contents that are already in the file, or that the previous contents can be overwritten.
  - b When we open a sequential file for reading, and the file does not exist, an unchecked error is thrown.
  - c We can open a sequential file for both reading and writing operations.
- (a), (b).

3. Which code lines will ensure that writing will start immediately after the contents that are already in the file?

- a `FileWriter textFileWriter = new FileWriter(dataFileName);`
- b `FileWriter textFileWriter = new FileWriter(dataFileName, true);`
- c `FileWriter textFileWriter = new FileWriter(dataFileName, false);`

(b).

(a) and (c) are equivalent.

4. Which classes have a constructor that accepts a file name as a parameter?

- a `FileWriter`
- b `PrintWriter`



- c** `FileReader`  
**d** `BufferedReader`
- (a), (b), (c).
- 5.** What methods are found in the `PrintWriter` class for writing string representations of primitive data values?
- a** `printInt(int i)`  
**b** `print(char i)`  
**c** `println(int i)`  
**d** `printIntln(int i)`
- (a), (c).
- 6.** What value does the method `readLine()` in the class `BufferedReader` return when it comes to the end of file?
- a** It returns the `null` value.  
**b** It returns the string "EOF".  
**c** It returns the value -1.  
**d** It throws a checked exception.
- (a).
- 7.** Assume that the variable `textReader` refers to an object of the class `BufferedReader`, which is connected to a text file. Which code will you choose to read from the file? Why?
- a** Use a do-while loop to read from the file.
- ```
String line = null;
do {
    line = textReader.readLine();
    System.out.println(line);
} while (line != null);
```
- b** Use a while loop to read from the file.
- ```
String line = textReader.readLine();
while (line != null) {
 System.out.println(line);
 line = textReader.readLine();
}
```
- (b).

Alternative (b) will avoid printing "null" when the end of file is reached.

- 8.** What is the recommended way of closing the file, given the following declarations:

```
FileWriter textFileWriter = new FileWriter(dataFileName);
PrintWriter textWriter = new PrintWriter(textFileWriter);
```



**a** `textFileWriter.close();  
textWriter.close();`

**b** `textFileWriter.close();  
textWriter.close();`

**c** `textWriter.close();`

**d** `textWriter.close(textFileWriter);`

(c).

(c) will ensure that all other connected resources are freed.

- 9.** Which methods must specify a `throws` clause with an `IOException` in order for the following code to compile?

```
class Writing {
 public static void main(String[] args) {
 PrintWriter textWriter = openFileForWrite("preciousData.txt");
 writeToFile(textWriter);
 closeWriteFile(textWriter);
 }

 static PrintWriter openFileForWrite(String dataFileName) {
 FileWriter textFileWriter = new FileWriter(dataFileName);
 PrintWriter textWriter = new PrintWriter(textFileWriter);
 return textWriter;
 }

 static void writeToFile(PrintWriter textWriter) {
 textWriter.println("To file or not to file.");
 }

 static void closeWriteFile(PrintWriter textWriter) {
 textWriter.close();
 }
}

a main(), openFileForWrite()
b main(), openFileForWrite(), writeToFile()
c main(), openFileForWrite(), writeToFile(), closeWriteFile()

(a).
```

The statements in the `writeToFile()` and `closeWriteFile()` methods do not throw an `IOException`. A call to the `FileWriter` constructor in the `openFileForWrite()` method can throw an `IOException`. The `openFileForWrite()` method must specify an `IOException` in a `throws` clause, and so must the `main()` method, as it calls the `openFileForWrite()` method.

- 10.** Which methods must specify a `throws` clause with an `IOException` in order for the following code to compile?

```
class Reading {
```



```
public static void main(String[] args) {
 BufferedReader textReader = openFileForRead("preciousData.txt");
 readFromFile(textReader);
 closeReadFile(textReader);
}

static BufferedReader openFileForRead(String dataFileName) {
 FileReader textFileReader = new FileReader(dataFileName);
 BufferedReader textReader = new BufferedReader(textFileReader);
 return textReader;
}

static void readFromFile(BufferedReader textReader) {
 System.out.println(textReader.readLine());
}

static void closeReadFile(BufferedReader textReader) {
 textReader.close();
}
}

a main(), openFileForRead()
b main(), openFileForRead(), readFromFile()
c main(), openFileForRead(), readFromFile(), closeReadFile()

(c).
```

The `main()` method calls the other three methods, which all contain code that can throw an `IOException`. All *four* methods must specify an `IOException` in a `throws` clause.

- 11.** Which statements are true about the following methods in the `JOptionPane` class?
- a** The method `showMessageDialog()` always shows an “OK” button.
  - b** The method `showInputDialog()` always shows an “OK” button and a “Cancel” button.
  - c** The method `showConfirmDialog()` can show any combination of an “OK” button, a “Yes” button, a “No” button and a “Cancel” button.
- (a), (b).

The `showConfirmDialog()` method can only show combinations specified by the following constants in the `JOptionPane` class: `DEFAULT_OPTION` (i.e. “OK” button), `YES_NO_OPTION`, `YES_NO_CANCEL_OPTION`, `OK_CANCEL_OPTION`.

- 12.** Which statements are true about the following constants in the `JOptionPane` class?
- a** The message type constants specify which icon is used in the dialog box.

- 11
- 13.** Which statement will show the following dialog box?
- 
- (a), (b), (c).
- a** `JOptionPane.showMessageDialog(null, "Shall we move on?", "Dialog", JOptionPane.PLAIN_MESSAGE);`
- b** `JOptionPane.showConfirmDialog(null, "Shall we move on?", "Dialog", JOptionPane.DEFAULT_OPTION, JOptionPane.PLAIN_MESSAGE);`
- (a), (b).

However, (a) will not return any value to indicate which button the user clicked.

## 11.5 Programming exercises

- 1.** Write a new version of the method `readEmployeeData()` in the class `CompanyUsingTextFiles` from Program 11.2 on page 300 that uses the `Scanner` class to extract the values from an employee record.

The example below shows how we can use a `Scanner` to read a record and extract the field values that are terminated by the ',' character, i.e. CSV format. Note how the field terminator character ',' is specified by the delimiter string "\\",," when using the `Scanner`.

```
import java.util.Scanner;

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



```

// The input record:
String record = "123,45.67,false,1949,786,BERGEN";
// Scanner takes the record as input string:
Scanner lexer = new Scanner(record);
// Uses the specified delimiter string,
// which represents the character ',' in this case:
lexer.useDelimiter("\\,");

// Read values sequentially using appropriate next() method:
int field1 = lexer.nextInt();
double field2 = lexer.nextDouble();
boolean field3 = lexer.nextBoolean();
int field4 = lexer.nextInt();
long field5 = lexer.nextLong();
String field6 = lexer.next();
System.out.printf("%5d%6.2f%8s%5d%4d%8s",
 field1, field2, field3, field4, field5, field6);

// Close Scanner when done:
lexer.close();
}
}

```

Output from the program:

123 45.67 false 1949 786 BERGEN

- 2.** Given two text files each containing a sorted sequence of integers in ascending order, write a program that merges the sequences into one sorted sequence in a third file. You can assume that the input file contains one integer per line. The output file should also be written with one value per line. The program should not store any of the sequences in memory. It should be possible to run the program with the following command:

> **java Merge file1 file2 destinationFile**

Be sure to write the pseudocode for the main steps of the program.

- 3.** Write a program that reads a sequence of integers from a text file and prints a report of how many times each digit from 0, 1, ... 9 occurs in the sequence. You can assume that the text file contains one integer per line, for example:

2006  
786  
1949  
1972  
...

The program should write the frequency of each digit to the terminal window, for example:

0 19  
1 23



```

2 8
3 5
...
9 16

```

- 4.** Modify the program from Exercise 11.3 so that the frequency of each digit is written to a result file.
- 5.** Write a new version of Program 6.4 on page 140 that reads the values of the two-dimensional array `weeklyData` from a text file. Choose a suitable data record format for storing the values of the two-dimensional array `weeklyData` in the text file.
- 6.** Write a new version of Exercise 6.1 and Exercise 6.2 on page 157 so that the sequence of digits is read from a text file and the histogram shown in a dialog box.
- 7.** Write a new version of Exercise 3.7 on page 69 in which the student ID and the number of points are read from a text file. Choose a suitable data record format for storing the values in the text file. The program should print a list with student ID and grade to the terminal window.
- 8.** Write a new version of Program 9.2 on page 240 to read the integers from a text file.
- 9.** Write the following programs for handling text files. The programs should read and check the program arguments from the command line before executing any operations on the files.
  - a** A program that counts the number of lines and characters in a file:  
`> java LineCounter myFile`
  - b** A program that reads from a text file and prints the average length of the text lines in the file.  
`> java AverageLineLength sourceFile`
  - c** A program that copies a text file to another file:  
`> java CopyFile sourceFile destinationFile`
  - d** A program that splits a file. The text lines in the source file should be split among several files depending on the number of text lines in the source file. The maximum number of text lines allowed in each file should be specified as a program argument. Choose appropriate names for the new files, for example, `file1.txt`, `file2.txt` and so on.  
`> java SplitFile 100 sourceFile`
  - e** A program that concatenates the contents of two files to a destination file, i.e. the destination file should contain the contents of the first file, followed by the contents of the second file:  
`> java ConcatFiles sourceFile1 sourceFile2 destinationFile`

Extend the program to concatenate any number of files to a destination file.

- f** A program that prints those lines in a file that containing a specified string (see the method `indexOf(String substring)` in the `String` class):



> **java FindString searchString sourceFile**

- g** A program that replaces a string in a file with another string (see the method `replaceAll(String oldStr, String newStr)` in the `String` class):

> **java ReplaceString oldSt newStr sourceFile destinationFile**

- h** A program that removes space and tab characters at the beginning and end of each text line in a file and writes the result to a new file (see the method `trim()` in the `String` class):

> **java TrimLines sourceFile destinationFile**

- i** A program that replaces all sequences of space and tab characters with a single space in each text line of a file:

> **java ShrinkFile sourceFile destinationFile**

- 10.** Write a program that encrypts and decrypts a file.

For example, the following command should encrypt the source file using the specified code file, and placing the encrypted text in the destination file:

> **java Crypto -C codeFile sourceFile destinationFile**

The following command should decrypt the source file using the specified code file, creating the original text in the destination file:

> **java Crypto -D codeFile sourceFile destinationFile**

The code file is a text file in which each line consists of two characters. The first character is replaced by the second character during encrypting, and vice versa during decrypting. The coding information can, for example, be represented by two arrays of characters, in which the same index in both arrays indicates the pair of characters that can replace each other. Assume that the characters in each array are unique, i.e. the first character is unique for all pairs, as is the second character.

- 11.** Write a program that converts the temperature from the Celsius scale to the Fahrenheit scale. Use the following formula, where `fTemp` and `cTemp` represent the temperature in degrees Fahrenheit and Celsius respectively:

`fTemp = (9.0/5.0) * cTemp + 32.0;`

The program should use dialog boxes for all interaction with the user.

- 12.** Modify the program from Exercise 5.6 on page 120 to read the temperature interval using dialog boxes and present the result in a message box.

- 13.** Modify the programs from Exercise 11.9 so that they use dialog boxes for input from the user.

- 14.** Modify the program from Exercise 3.1 on page 67 to read the duration of a time interval in seconds from an input dialog box and present the result in a message dialog box.

- 15.** Modify the program from Exercise 4.3 on page 97 to read the text from an input dialog box and present the result in a message dialog box.



16. Modify the program from Exercise 5.3 on page 120 to read the text from an input dialog box and present the result in a message dialog box.
17. Modify the program from Exercise 9.14 on page 268 to read the integers from a text file, to specify the key in an input dialog box, and to present the result in a message box.
18. Write a program that keeps track of items in a museum. Each item (class `Item`) has the following information:
  - A unique reference number for the item
  - Name of the item
  - Description of the item

The program should let the user execute the following operations in the museum (class `Museum`):

- Register a new item in the museum
- Change the information about an existing item in the museum
- Delete an item from the museum collection

First write pseudocode for the operations that can be performed, then implement the operations.

The collection of items should be stored in a suitable data structure during execution, for example in an array.

All interaction between the program and the user should be through a text-based user interface (class `TextBasedMuseumInteraction`). The part of the program that implements the text-based interaction should be separate from the rest of the program. All other classes must be independent of the text-based interaction. It should be possible to replace the text-based interaction of the program with, for example, a graphical user interface without having to modify any of the other classes.

The program should store the collection of items in the museum in a text file. When the program starts it should read any information about existing items from a file, if such information exists. Before the program quits, information about items should be written out to a file. Choose an appropriate record format to represent an item in the file.

19. Implement a simple GUI dialog-based user interface (class `GUIBasedMuseumInteraction`) to replace the interaction between the program and the user in Exercise 11.18.







## P A R T   F O U R

**Object-oriented Programming (OOP)**



## Inheritance

### LEARNING OBJECTIVES

By the end of this chapter, you will understand the following:

- What inheritance is and how it promotes reuse of code.
- The *is-a* relationship between a superclass and a subclass.
- How to extend properties and behaviour in a subclass.
- How to access inherited members from within a subclass.
- Conversion between reference types (upcasting, downcasting).
- Overriding a superclass method.
- The difference between overriding and overloading a method.
- Making classes and methods final to disallow design changes.

### INTRODUCTION

This chapter introduces some fundamental concepts in object-oriented programming (OOP) and illustrates them by using UML diagrams and code examples. *Inheritance* allows new classes to be defined based on existing ones. OOP extends object-based programming (OBP) with inheritance. We illustrate the use of inheritance by calculating the salary of different kinds of employees in a fictional company called DOT-COM. Subsequent chapters provide more examples of OOP application.



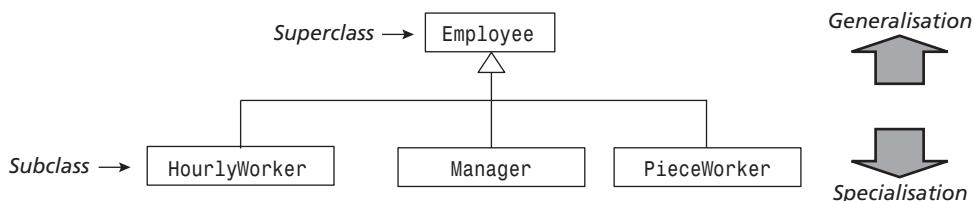
## 12.1 Main concepts

### Superclasses and subclasses

Figure 12.1 shows some central concepts related to inheritance. An existing class turns out to be a suitable basis for several other classes, that need similar properties and behaviour. The new classes can inherit members from the existing class and can, in addition, extend the existing class with new members. The existing class is called a *superclass* (a.k.a. *basic class* or *parent class*), and each of the new classes inheriting from this class, is called a *subclass* (a.k.a. *derived class* or *child class*).

Even in a small company like DOT-COM there can be several types of employees. The various employee types have some common properties and behaviour, such as name, hourly salary and methods for calculating the weekly salary. There are also differences between the different types of employees. For instance, a manager has a bonus that is added to the salary, while an hourly worker, who is only paid by the hour, does not get paid for overtime. We can define a new class for each type of employee from scratch, but these classes will have much overlap and repeated code. This can be avoided by using inheritance, which allows subclasses to share common fields and methods from the superclass. The superclass declaration thus defines a *contract* that specifies what properties and behaviour all objects of this class and its subclasses will provide.

**FIGURE 12.1 Inheritance hierarchy**



### Specialisation and generalisation

Since subclasses can extend properties and/or behaviour, we can consider them to be special cases of the superclass. For this reason, the process of defining subclasses is often called *specialisation*. The superclass represents the common aspects for all subclasses, and can be seen as a more general case. Thus, we say that we perform a *generalisation* when going from a group of subclasses to their common superclass. The relationship between a subclass and its superclass is called *is-a* relationship, a.k.a. *inheritance relationship*. For instance, an hourly worker *is-a* employee since the class `HourlyWorker` is a subclass of the class `Employee`. Such *is-a* relationships can be shown in an *inheritance hierarchy*, as illustrated in Figure 12.1.

The purpose of inheritance is to promote reuse of source code. In a program where many classes resemble each other, common properties and behaviour can be placed in one class, which becomes the superclass, and then the other classes are defined as special cases, i.e. as subclasses.



For an inheritance relationship to serve any purpose, there must be some aspects that differentiate the classes. For instance, a manager will need a field for the manager bonus. This amount is to be added to the salary every week, and consequently there is also a need to change the calculation of weekly salary for this type of employee.

Changes in properties and behaviour are closely related. Often we want to change both of these aspects in a subclass, but there may also be cases where only the behaviour needs to be modified. For instance, we can define a subclass for hourly workers, where the salary will be calculated based on hourly salary and number of hours worked in the current week, not taking into account any overtime. In this case the subclass will not need new properties, but the behaviour for the calculation of the salary must be modified.

### BEST PRACTICES

Capture common properties and behaviour of related classes in a contract, and define a super-class that implements this contract. The subclasses can then specialise this contract as necessary.

### BEST PRACTICES

A class B should only inherit from a class A *if and only if all* the behaviour and properties of class A are relevant for implementing class B. This is called the litmus test for *design by inheritance*.

## Inheritance in Java

The Java standard library uses inheritance extensively. It is common to find libraries of classes organised in inheritance hierarchies for most object-oriented programming languages, either as libraries distributed with the compiler, or as third-party packages. For example, libraries for developing graphical user interfaces contain classes that are typically organised in inheritance hierarchies.

In Java all classes inherit from the `Object` class. The following class declaration

```
class Employee { ... }
```

is thus equivalent with:

```
class Employee extends Object { ... }
```

where the keyword `extends` shows the relationship between the superclass and the subclass.

In Java each subclass inherits from only *one* superclass. This is called *single inheritance*. Since the `Object` class is always the superclass of all classes, these classes and thereby all their objects inherit some common behaviour from the `Object` class. This behaviour includes, among other things, a method that creates a string representation of the object



state (`toString()`), as well as a method that tests whether two objects are equal (`equals()`).

## 12.2 Extending properties and behaviour

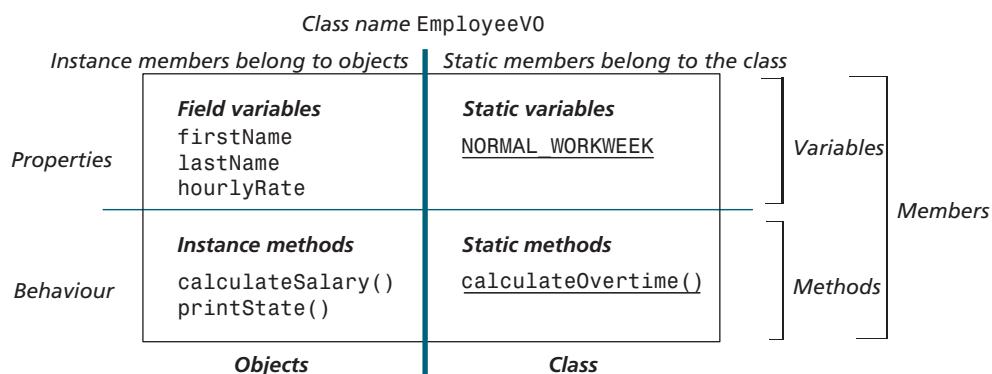
We will use a simple employee class to illustrate extending properties and behaviour. The members of this `EmployeeV0` class are shown in Figure 12.2.

The `EmployeeV0` class contains only first name, last name and hourly rate for an employee. The class has methods for the calculation of the weekly salary and for printing the state of an `EmployeeV0` object. In addition, we have defined a constant for the number of hours in a normal working week as a static variable, and a static method that calculates the number of hours worked overtime.

All members of the `EmployeeV0` class are *visible* to other classes in the same directory, as no *access modifier* has been specified to limit their access. These members will therefore be inherited by all subclasses extending the `EmployeeV0` class. If we do not want a member to be inherited by subclasses, we can declare it with the keyword `private` in the member declaration. This applies also to methods. A private method cannot be called outside its class, and will not be inherited by the subclasses. Private methods typically perform a well-defined task that is needed by other methods in the class, but it is not relevant or desirable to allow access to the method from other classes. Hiding properties and behaviour in this manner is explained in the section *Access modifiers for class members* on page 443. To keep things simple, we will not use any access modifiers for class members in the rest of this chapter.

The complete source code for the `EmployeeV0` class is shown in Program 12.1. Note the use of assertions to control that the actual parameter to methods `calculateSalary()` and `calculateOvertime()` has a valid value. In the `calculateOvertime()` method the computed result is also verified. Assertions are used throughout the chapter to verify the values of parameters and computed values.

**FIGURE 12.2** Overview of members in the `EmployeeV0` class





## PROGRAM 12.1 The superclass EmployeeVO

```

/**
 * An employee is paid by the hour at a fixed hourly rate.
 * However, for each hour worked overtime during the week,
 * the hourly rate is doubled.
 */
class EmployeeVO {
 // Static variable (constant)
 final static double NORMAL_WORKWEEK = 37.5;

 // Field variables
 String firstName;
 String lastName;
 double hourlyRate;

 // Constructors
 /**
 * Creates an employee with default values for all fields.
 */
 EmployeeVO() { // explicit default constructor
 }
 /**
 * Creates an employee with the specified name and hourly rate.
 * @param firstName First name (and middle names, if any) of the employee.
 * @param lastName Last name of the employee.
 * @param hourlyRate Payment per hour in GBP.
 */
 EmployeeVO(String firstName, String lastName, double hourlyRate) {
 this.firstName = firstName;
 this.lastName = lastName;
 this.hourlyRate = hourlyRate;
 }

 // Instance methods
 /**
 * Calculates the weekly salary for the employee.
 * @param numHours Number of hours worked in current week.
 * @return Weekly salary in GBP for the current week.
 */
 double calculateSalary(double numHours) {
 // Using assert to check the value of the actual parameter
 assert numHours >= 0 : "Number of hours must be >= 0";
 double fixedSalary = hourlyRate * NORMAL_WORKWEEK;
 if (numHours <= NORMAL_WORKWEEK) {
 return fixedSalary;
 } else {
 return fixedSalary + 2.0 * hourlyRate * calculateOvertime(numHours);
 }
 }
}

```



```

 }
 /**
 * Prints the name and hourly rate for the employee.
 */
 void printState() {
 System.out.printf("Employee %s %s has an hourly rate of %.2f GBP%n",
 firstName, lastName, hourlyRate);
 }

 // Static method
 /**
 * Calculates number of hours overtime an employee has worked this week.
 * @param numHours Total number of hours worked by an employee.
 * @return Number of overtime hours.
 */
 static double calculateOvertime(double numHours) {
 // Using assert to check the value of the actual parameter
 assert numHours >= 0 : "Number of hours must be >= 0";
 double overtime = 0.0; // assume no overtime
 if (numHours > NORMAL_WORKWEEK) {
 overtime = numHours - NORMAL_WORKWEEK; // has worked overtime
 }
 assert overtime >= 0 : "Number of overtime hours must be >= 0";
 return overtime;
 }
}

```

---

## Initialising object state

In our company, DOT-COM, there are some employees that are in charge of other employees. These are called *managers* and they receive a fixed bonus each week as compensation for the additional work they do in leading a group. To represent this type of employees, we create a subclass `ManagerVO` with a new field to hold the manager bonus. Program 12.2 shows the definition of this subclass.

A class must make sure that any object created from the class is initialised to a valid initial state. The `ManagerVO` class inherits the fields `firstName`, `lastName` and `hourlyRate`. These fields can either be initialised directly in the subclass or by calling one of the superclass constructors. The call `super()` in (1) results in the superclass constructor with the signature `EmployeeVO()` to be executed. Similarly the constructor call `super(firstName, lastName, hourlyRate)` at (2) and (3) results in the execution of the superclass constructor with the signature `EmployeeVO(String, String, double)`. This means that the superclass constructor is called before the rest of the subclass constructor is executed. This chaining of constructors is performed upwards in the inheritance hierarchy until the `Object` class is reached.

The call to the superclass constructor must always be the *first* statement in the subclass constructor. This ensures that all fields in the superclass have a valid value before the subclass starts using them.



If the subclass constructor does not explicitly call a superclass constructor, then the default superclass constructor with the signature `super()` will be called implicitly.

The types of the actual parameters in the call to the superclass constructor must be identical with the types of the formal parameters declared in one of the superclass constructors. In Program 12.2 the first call to the superclass constructor is without parameters, and will result in a call to the explicit default constructor of the superclass. At lines (2) and (3) the types of the actual parameters match the types of the formal parameters in the second constructor of the `EmployeeV0` superclass.

If a superclass has only non-default constructors, the compiler will issue an error message if the subclass calls `super()` (without parameters).

The next important step in the subclass constructor is initialising fields that are declared in the subclass itself. In Program 12.2 the default constructor does not initialise the manager bonus, therefore the field is set to the default value (0.0) for `double` variables. The same is the case for the second constructor. It simply ignores the bonus value. The third constructor initialises the manager bonus with the amount passed in the last parameter.

### BEST PRACTICES

It is a good idea to call a superclass constructor explicitly in each subclass constructor. Also make sure that fields declared in the subclass are properly initialised in the subclass constructors. This way every object of the subclass will have a valid initial state.

## Extending behaviour

The `ManagerV0` class also declares a new instance method, `calculateManagerSalary()`, that calculates the weekly salary for a manager. Clients can call this method on objects of the `ManagerV0` class. This is illustrated in Program 12.3. A `ManagerV0` object is created at (1):

```
ManagerV0 manager = new ManagerV0("John D.", "Boss", 60.0); // (1)
```

The constructor call at (1) will result in the second superclass constructor (at (2) in Program 12.2) being executed. We can assign the manager bonus for this object and calculate the manager salary:

```
manager.managerBonus = 105.0; // (2)
System.out.printf("Salary: %.2f GBP\n",
 manager.calculateManagerSalary(50.5)); // (3)
```

Figure 12.3 shows an object of the class `ManagerV0` that has inherited all fields from the superclass `EmployeeV0`, and that has, in addition, a field called `managerBonus` that is declared in the subclass itself. Figure 12.3 shows the state of the object referred to by the reference `manager` after its creation at (1) and modification of the field `managerBonus` at (2). Figure 12.3 also shows some of the members the `manager` object inherits from the `Object` class. Note also the relation between the object and the inheritance hierarchy.



Extending behaviour by defining a new method, `calculateManagerSalary()`, enables us to calculate the weekly salary for a manager, but at the expense of breaking the contract defined by the `EmployeeVO` class for salary calculation. In fact, a `ManagerVO` object can calculate the weekly salary in two ways: by calling the method `calculateManagerSalary()` in the `ManagerVO` class and by calling the inherited method `calculateSalary()` from the superclass `EmployeeVO`. In the section *Extending behaviour in the subclass* on page 344 we will show how to extend the behaviour of a subclass *without* breaking the contract of the superclass.

### PROGRAM 12.2 A simple subclass for managers

```
/*
 * A manager is paid by the hour at a fixed hourly rate.
 * For each hour worked overtime during the week,
 * the hourly rate is doubled.
 * In addition, the manager always receives a fixed manager bonus.
 */
class ManagerVO extends EmployeeVO {
 // Field variable
 double managerBonus;

 // Constructors
 /**
 * Creates a manager with default values for all fields.
 */
 ManagerVO() { // explicit default constructor
 super(); // (1)
 }
 /**
 * Creates a manager with the specified name and hourly rate.
 * The manager bonus is set to the default value (0.0).
 * @param firstName First name (and middle names, if any) of the manager.
 * @param lastName Last name of the manager.
 * @param hourlyRate Payment per hour in GBP.
 */
 ManagerVO(String firstName, String lastName, double hourlyRate) {
 super(firstName, lastName, hourlyRate); // (2)
 }
 /**
 * Creates a manager with the specified name, hourly rate and manager bonus.
 * @param firstName First name (and middle names, if any) of the manager.
 * @param lastName Last name of the manager.
 * @param hourlyRate Payment per hour in GBP.
 * @param managerBonus A fixed bonus that is added to the weekly salary.
 */
 ManagerVO(String firstName, String lastName,
 double hourlyRate, double bonus) {
 super(firstName, lastName, hourlyRate); // (3)
 managerBonus = bonus; // Stores the extended state
 }
}
```



```

}

// Instance method
/**
 * Calculates the weekly salary of the manager.
 * @param numHours Number of hours worked in the current week.
 * @return Weekly salary in GBP for the current week.
 */
double calculateManagerSalary(double numHours) { // (4)
 // Using assert to check the value of the actual parameter
 assert numHours >= 0 : "Number of hours must be >= 0";
 double fixedSalary = hourlyRate * NORMAL_WORKWEEK; // (5)
 if (numHours <= NORMAL_WORKWEEK) { // (6)
 return fixedSalary + managerBonus;
 } else {
 return fixedSalary
 + 2.0 * hourlyRate * calculateOvertime(numHours) // (7)
 + managerBonus;
 }
}
}

```

---

### PROGRAM 12.3 A client using the ManagerVO subclass

```

/**
 * Calculates the weekly salary of a manager.
 */
class TestManagerVO {
 public static void main(String[] args) {
 ManagerVO manager = new ManagerVO("John D.", "Boss", 60.0); // (1)
 manager.managerBonus = 105.0; // (2)
 System.out.printf("Salary: %.2f GBP%n",
 manager.calculateManagerSalary(50.5)); // (3)
 }
}

```

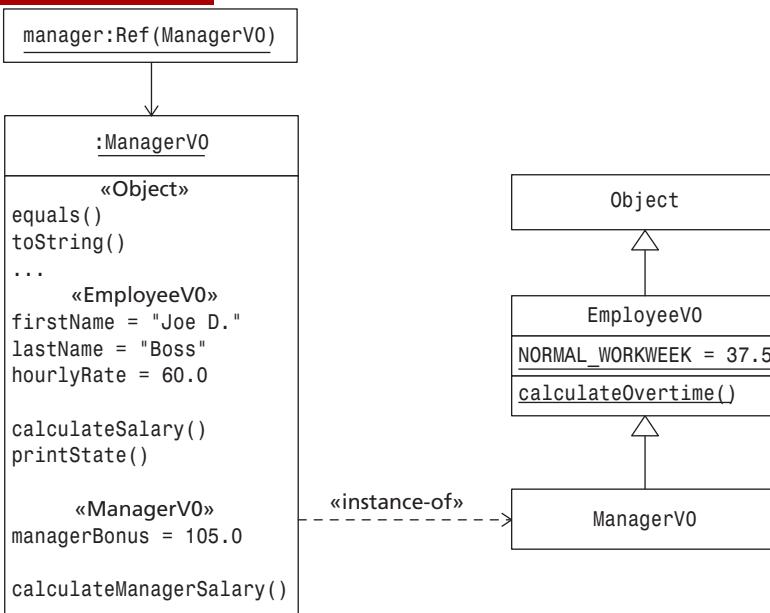
Program output:

Salary: 3915.00 GBP

---



**FIGURE 12.3** Object diagram for a manager



(a) After creation of a ManagerV0 object and changing properties

## 12.3 Using inherited members

### Accessing inherited members in the subclass

Program 12.2 shows how an object of a subclass can refer to members it inherits through its inheritance hierarchy. For instance, the instance method `calculateManagerSalary()` at (4) accesses the inherited field `hourlyRate` at (5), the inherited static variable `NORMAL_WORKWEEK` at (6) and the inherited static method `calculateOvertime()` at (7). These three members are inherited by the subclass `ManagerV0` from the superclass `EmployeeV0` shown in Program 12.1. The subclass can refer to the inherited members as if they were declared in the subclass itself.

We can also use the `this` reference to refer to an inherited field in the subclass:

```
hourlyRate = 7.5; // access by simple field name, implicit this reference
this.hourlyRate = 7.5; // access by explicit this reference
```

The subclass can call inherited instance methods in the usual way:

```
printState(); // call by simple method name, implicit this reference
this.printState(); // call by explicit this reference
```

### Accessing inherited members using subclass references

Clients of a subclass can also access the inherited members of the subclass as if they were declared in the subclass. Program 12.4 shows how a client uses the `ManagerV0` class. The client uses the subclass reference `manager` to access instance members, both those that are



declared in the subclass, as well as those inherited by the subclass. The client also uses the subclass manager reference to access inherited static members, as shown at (1). Finally, the client uses the subclass name and the superclass name to access static variables in the superclass as shown at (2) and (3), respectively.

#### PROGRAM 12.4 Accessing inherited members using subclass references

```
/**
 * Retrieves inherited state and uses inherited behaviour in the subclass.
 */
class UseInheritance {
 public static void main(String[] args) {
 double numOfHoursWorked = 50.5;
 // Creating a manager object
 ManagerVO manager = new ManagerVO("John D.", "Boss", 60.0);

 // Refers to subclass field by field name
 manager.managerBonus= 650.0;

 // Refers to inherited and subclass fields by field names
 System.out.printf("Manager %s has a bonus of %.2f GBP%n",
 manager.lastName, manager.managerBonus);

 // Calls subclass instance methods by method names
 System.out.printf("Manager salary is %.2f GBP%n",
 manager.calculateManagerSalary(numOfHoursWorked));

 // Calls inherited instance method by method name
 manager.printState();

 // Refers to inherited static variable
 System.out.printf("Normal work week is %.1f hours%n",
 manager.NORMAL_WORKWEEK); // (1)
 if (numOfHoursWorked > ManagerVO.NORMAL_WORKWEEK) { // (2)
 System.out.printf("Overtime: %.1f hours%n",
 (numOfHoursWorked - EmployeeVO.NORMAL_WORKWEEK)); // (3)
 } else {
 System.out.println("No overtime.");
 }
 }
}
```

Program output:

```
Manager Boss has a bonus of 650.00 GBP
Manager salary is 4460.00 GBP
Employee John D. Boss has an hourly rate of 60.00 GBP
Normal work week is 37.5 hours
```



Overtime: 13.0 hours

## Using superclass references to refer to subclass objects

A reference of a superclass type is called a *superclass reference*. Obviously such a reference can refer to superclass objects, i.e. objects of the superclass. But it can also refer to subclass objects, since a subclass object *is-a* superclass object. It can be used to access *inherited* members in subclass objects. In Program 12.5, the `employee` reference is a reference of the superclass `EmployeeV0`. At runtime, it will refer to an object of the subclass `ManagerV0`. Members inherited by this subclass object from the superclass `EmployeeV0` can be accessed using the superclass reference `employee`:

```
employee.firstName = "Hugo";
employee.printState();
```

However, it is *only* inherited members that can be accessed by a superclass reference. If we try to retrieve the value of a field declared in the subclass using a superclass reference, the compiler will report an error:

```
employee.managerBonus = 105.0; // Compile-time error!
```

This is understandable in the above statement as the compiler only sees a reference named `employee` of type `EmployeeV0`, and the `EmployeeV0` class does *not* declare a field called `managerBonus`. The compiler has no way of knowing what type of object the superclass reference will refer to at runtime. If it turned out at runtime that the subclass object referred to by the `employee` reference was not a `ManagerV0` object after all, say it was an `HourlyWorker`, the above statement would execute an illegal operation. In other words, the compiler cannot guarantee the runtime behaviour in this case, and therefore reports an error.

If we want to access members in the subclass object that are declared in the subclass, we need a subclass reference. In Figure 12.4, the length of the grey bar shows which members can be accessed in an object, depending on the type of the reference used.

### PROGRAM 12.5 Using superclass references

```
/*
 * Uses a superclass reference to refer to a subclass object.
 */
class SuperIsDuper {
 public static void main(String[] args) {
 // A superclass reference can refer to a subclass object.
 EmployeeV0 employee = new ManagerV0("John D.", "Boss", 60.0);

 // Inherited fields and methods can be accessed by a superclass reference.
 employee.firstName = "Hugo";
 employee.printState();

 // But a superclass reference cannot refer to a subclass member:
 }
}
```

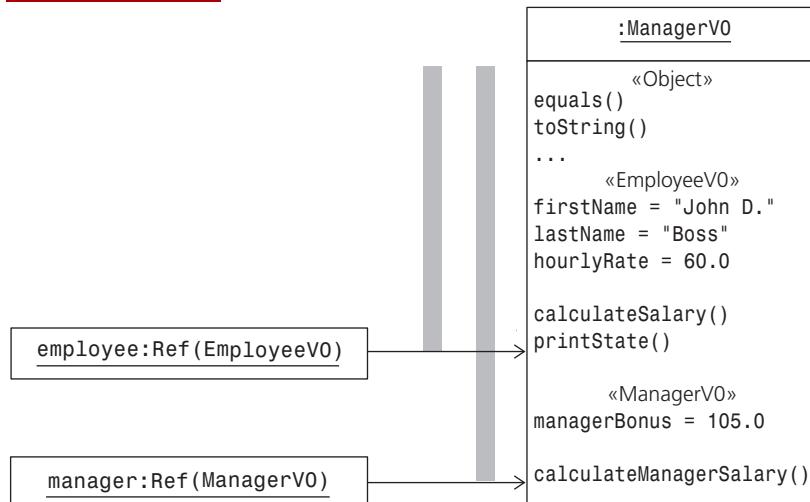


```
// employee.managerBonus = 105.0; // Compile-time error!
}
}
```

Program output:

Employee Hugo Boss has an hourly rate of 60.00 GBP

**FIGURE 12.4** Superclass and subclass references



## Conversion between reference types

A superclass reference of type `EmployeeVO` can refer to both objects of the `EmployeeVO` class and objects of the `ManagerVO` class. Since a subclass object can be used where a superclass object can, it is legal to allow superclass references to refer to subclass objects. Converting reference types from subclass to superclass references is called *upcasting*, since the conversion is performed upwards in the inheritance hierarchy.

```
employee = new ManagerVO("John D.", "Boss", 60.0, 105.0); // (1)
```

In the assignment statement above, the reference value of a subclass object of type `ManagerVO` is assigned to the superclass reference `employee` of type `EmployeeVO`. Upcasting is automatic (implicit), because a subclass object can behave like a superclass object.

If we then want to refer to properties that are specific to the subclass object, it is tempting to use a subclass reference:

```
ManagerVO manager = employee; // Compile-time error
```

However, the compiler will not accept the assignment above. It cannot guarantee that the superclass reference `employee` will refer to an object of the subclass `ManagerVO` at runtime. For instance, the reference `employee` can refer to an object of the superclass `EmployeeVO` that obviously is not a `ManagerVO`. We have to explicitly state the conversion of the super-



class reference using the *cast operator*, (*typename*), before the compiler will allow the assignment:

```
ManagerVO manager = (ManagerVO) employee;
```

During program execution this will work as long as the superclass reference `employee` refers to an object of the subclass `ManagerVO`. If the superclass reference `employee` refers to other types of objects, the execution will be aborted. If we are certain that the object is a `ManagerVO` object, the assignment above poses no problems. If we are not certain, then we have to check that the *type of the object* referred to by the superclass reference is that of the appropriate subclass, before attempting the type conversion. Java offers the `instanceof` operator for determining the type of the object referred to by a reference. The following code from Program 12.6 illustrates the use of this operator:

```
if (employee instanceof ManagerVO) { // (3)
 ManagerVO manager = (ManagerVO) employee; // (4)
 System.out.printf("Manager bonus is %.2f GBP%n", manager.managerBonus); // (5)
}
```

If the reference `employee` refers to an object of the `ManagerVO` class, the `instanceof` operator will return the value `true`. In that case the type conversion is safe, and we can use the converted reference to access members declared in the subclass. If the reference `employee` refers to an object of the `EmployeeVO` class, the `instanceof` operator will return the value `false`. In that case, the actions in the `if` body, lines (4) and (5), will not be executed.

Converting from a superclass reference to a subclass reference is called *downcasting*, as the conversion is downwards in the inheritance hierarchy, from the superclass to the subclass. Downcasting requires the cast operator, and it is only done after ensuring that the superclass reference actually refers to an object of the appropriate subclass.

### BEST PRACTICES

Always verify that the object referred to by a superclass reference is of the correct type before downcasting its reference value. This is the recommended practice in the `equals()` method, where the reference value of the second object is passed in a reference of the `Object` type.

## Using superclass and subclass references

In Program 12.6 the superclass reference `employee` can be used to either refer to an object of the `EmployeeVO` class or to an object of the `ManagerVO` class. Regardless of the object type, it can be used to access members from the superclass `EmployeeVO`, and an example of this is the call to the `printState()` method at (2).

In Program 12.6, the subclass reference `manager` is only used to refer to members declared specifically in the `ManagerVO` class. After downcasting the superclass reference `employee` at (4), the value of the manager bonus is retrieved at (5) using the subclass reference `manager`. Upcasting takes place in (1) where a `ManagerVO` object is created and its reference value is assigned to the superclass reference `employee`.



## PROGRAM 12.6 Using superclass and subclass references

```

import java.util.Scanner;
/**
 * Calculates the weekly salary of an employee or a manager, depending on
 * user input.
 */
class ReferenceConversion {
 public static void main(String[] args) {
 EmployeeVO employee;
 Scanner keyboard = new Scanner(System.in);

 // Chooses which type of employee to create
 System.out.println("Types of employees:");
 System.out.println(" 1 - Employee");
 System.out.println(" 2 - Manager");
 System.out.print("Choose a category: ");
 int empCode = keyboard.nextInt();

 // Creates an employee of the chosen type
 switch(empCode){
 case 1: employee = new EmployeeVO("John", "Doe", 30.0);
 break;
 case 2: employee = new ManagerVO("Hector", "Vee", 55.0, 95.0); // (1)
 break;
 default: System.out.println("Invalid category - creating an Employee");
 employee = new EmployeeVO("John", "Smith", 25.0);
 break;
 }

 // Prints a report for the employee
 System.out.println("Printing information:");
 // Prints properties common to all types of employees
 employee.printState(); // (2)
 // Prints subclass specific data for managers
 if (employee instanceof ManagerVO) { // (3)
 ManagerVO manager = (ManagerVO) employee; // (4)
 System.out.printf("Manager bonus is %.2f GBP%n",
 manager.managerBonus); // (5)
 }
 }
}

```

Program output when an EmployeeVO object is created:

Types of employees:

- 1 - Employee
- 2 - Manager

Choose a category: 1

Printing information:



Employee John Doe has an hourly rate of 30.00 GBP

Program output when a ManagerV0 object is created:

Types of employees:

- 1 - Employee
- 2 - Manager

Choose a category: 2

Printing information:

Employee Hector Vee has an hourly rate of 55.00 GBP

Manager bonus is 95.00 GBP

---

## 12.4 Extending behaviour in the subclass

### Overriding instance methods in the subclass

A subclass usually wants to extend the behaviour from the superclass, but without breaking the contract specified by the superclass. This means that the subclass will have to offer the same services as the superclass. For instance, our company needs to calculate weekly salary for managers as well as for other types of employees. Clients for salary calculation are simpler to write if we can just use the same method call to calculate the right salary for any type of employee, without having to differentiate between the different types of employees.

To demonstrate this style of programming, we now write a new version of the subclass for managers. Program 12.7 shows the new subclass ManagerV1 in which we have defined a new declaration for the `calculateSalary()` method that has the same signature and return type as in the superclass. We say that the subclass method *overrides* the superclass method.

The client in Program 12.8 uses the ManagerV1 class to calculate the weekly salary for a manager. It calls the overridden method in the usual way:

```
System.out.printf("Weekly salary: %.2f GBP%n",
 manager.calculateSalary(50.5)); // (3)
```

#### PROGRAM 12.7 A subclass overriding an instance method

```
/*
 * A manager is paid by the hour at a fixed hourly rate.
 * For each hour worked overtime during the week,
 * the hourly rate is doubled.
 * In addition, the manager always receives a fixed manager bonus.
 */
class ManagerV1 extends EmployeeV0 {
 // Field variable
 double managerBonus;
```



```
// Constructors as for ManagerV0

// Instance method
/**
 * Calculates the weekly salary of the manager.
 * This method overrides the method in the EmployeeV0 class.
 * @param numHours Number of hours worked in the current week.
 * @return Weekly salary in GBP for the current week.
 */
double calculateSalary(double numHours) { // (4)
 // Using assert to control values of actual parameters
 assert numHours >= 0 : "Number of hours must be >= 0";
 double fixedSalary = hourlyRate * NORMAL_WORKWEEK; // (5)
 if (numHours <= NORMAL_WORKWEEK) {
 return fixedSalary + managerBonus;
 } else {
 return fixedSalary
 + 2.0 * hourlyRate * calculateOvertime(numHours)
 + managerBonus;
 }
}
```

### PROGRAM 12.8 Using the overridden instance method

```
/**
 * Calculates the weekly salary of a manager.
 */
class TestManagerV1 {
 public static void main(String[] args) {
 ManagerV1 manager = new ManagerV1("John D.", "Boss", 60.0, 105.0); // (1)
 System.out.println("Information about a manager:");
 manager.printState(); // (2)
 System.out.printf("Weekly salary: %.2f GBP\n",
 manager.calculateSalary(50.5)); // (3)
 }
}
```

Program output:

```
Information about a manager:
Employee John D. Boss has an hourly rate of 60.00 GBP
Weekly salary: 3915.00 GBP
```



## Using an overridden method from the superclass

The calculation for the weekly salary of a manager is almost identical to that of an `EmployeeV0`, except for the fact that a manager gets a weekly bonus in addition. We can therefore call the superclass method first to calculate the employee salary, and then add the manager bonus. We call the overridden method to calculate the employee salary by means of the keyword `super`, that can be used to refer to all visible members in the superclass.

The `calculateSalary()` method in the `ManagerV2` class is greatly simplified, as shown in Program 12.9. Here, we first call the superclass `calculateSalary()` method at (5), and then add the manager bonus at (6):

```
double fixedSalary = super.calculateSalary(numHours); // (5)
fixedSalary += managerBonus; // (6)
```

Compared to Program 12.7, the method for calculating the weekly salary for managers is far easier to maintain, since we reuse the superclass method and only need one new line of code to take the manager bonus into account. Note the use of the keyword `super` at (5). This is required to call the method in the superclass, since the subclass has overridden this method.

We can also use `super` to refer to inherited members that are *not* overridden from the superclass. In Program 12.9, the `printState()` method retrieves values for the name and the hourly salary using the keyword `super` at (8), as well as obtaining the value of the static variable `NORMAL_WORKWEEK` at (9). Note that all of these members are inherited by the subclass, and can therefore be accessed without the keyword `super`.

Note that it is only the *implementation* of the `calculateSalary()` and `printState()` methods that has changed between the `ManagerV1` and the `ManagerV2` classes. The contract of the `ManagerV2` class is the same as that of the `ManagerV1` class, the only difference being their names. It is not surprising that the clients in Program 12.8 and Program 12.10 are identical, except for the version of the manager class they use.

### BEST PRACTICES

Avoid changing the superclass contract. Let subclasses offer refined or new services by specialising the superclass contract. This way the impact of any changes is minimised on the clients.

### BEST PRACTICES

Reuse the services of the superclass by means of the keyword `super`, allowing your extended services in the subclass to call superclass methods to perform as much of the work as possible.



## PROGRAM 12.9 A subclass using an overridden method from the superclass

```
/**
 * A manager is paid by the hour at a fixed hourly rate.
 * For each hour worked overtime during the week,
 * the hourly rate is doubled.
 * In addition, the manager always receives a fixed manager bonus.
 */
class ManagerV2 extends EmployeeV0 {
 // Field variable
 double managerBonus;

 // Constructors as for ManagerV0

 // Instance method
 /**
 * Calculates the weekly salary of the manager.
 * This method overrides the method in the EmployeeV0 class.
 * @param numHours Number of hours worked in the current week.
 * @return Weekly salary in GBP for the current week.
 */
 double calculateSalary(double numHours) { // (4)
 // Using assert to control values of actual parameters
 assert numHours >= 0 : "Number of hours must be >= 0";
 double fixedSalary = super.calculateSalary(numHours); // (5)
 fixedSalary += managerBonus; // (6)
 return fixedSalary;
 }

 /**
 * Prints name, hourly salary and bonus for the manager.
 * This method overrides the method in the EmployeeV0 class.
 */
 void printState() { // (7)
 System.out.printf("Manager %s %s has an hourly rate of %.2f GBP%n",
 super.firstName, super.lastName, super.hourlyRate); // (8)
 System.out.printf("and a bonus of %.2f GBP for %.1f hours work%n",
 managerBonus, super.NORMAL_WORKWEEK); // (9)
 }
}
```

## PROGRAM 12.10 A client using the new subclass for managers

```
/**
 * Calculates the weekly salary of a manager.
 */
class TestManagerV2 {
```



```

public static void main(String[] args) {
 ManagerV2 manager = new ManagerV2("John D.", "Boss", 60.0, 105.0); // (1)
 System.out.println("Information about a manager:");
 manager.printState(); // (2)
 System.out.printf("Weekly salary: %.2f GBP%n",
 manager.calculateSalary(50.5)); // (3)
}
}

```

Program output:

```

Information about a manager:
Manager John D. Boss has an hourly rate of 60.00 GBP
and a bonus of 105.00 GBP for 37.5 hours work
Weekly salary: 3915.00 GBP

```

## Using a shadowed field

If the subclass declares a field with the same name as a field in the superclass, we say that the field in the superclass is a *shadowed field*. Such fields cannot be referred to by the field name or through the `this` reference, since these forms will refer to the subclass field of the same name. The subclass must then use the keyword `super` to read the value of or assign a new value to the shadowed field from the superclass. In the following code, the field declaration at (1) shadows the field with the same name in the superclass `EmployeeV0`. The field in the superclass is accessed at (2) using the keyword `super`.

```

class ManagerV3 extends EmployeeV0 {
 int hourlyRate; // (1) Shadows field in EmployeeV0 with the same name
 ...
 public void printState() {
 System.out.printf("Manager %s %s has an hourly rate of %.2f GBP%n",
 firstName, lastName, // No name conflict, field names are sufficient
 super.hourlyRate); // (2) Must use super to access superclass field
 }
}

```

One final comment on the use of the superclass `EmployeeV0` given in Program 12.1: it can be used independently of its subclasses. If a program only requires services from the `EmployeeV0` class, it need not be concerned with its subclasses. The services provided by the `EmployeeV0` class are in no way altered just because we have defined any subclasses for it.

### BEST PRACTICES

Avoid shadowing a field from the superclass, since such fields can be a potential source of program errors.



## 12.5 Final classes and methods

### Final classes

Sometimes we want to prevent a class being specialised, for example, to avoid change in properties and/or in behaviour. A class that cannot be extended, is called a *final class*. Final classes in Java are marked with the keyword `final`.

If we want to prevent the `HourlyWorker` class from being extended by a new subclass, we can declare the class as follows:

```
final class HourlyWorker extends EmployeeVO {
 ...
}
```

The compiler will not allow any class to extend and thereby inherit from the `HourlyWorker` class. Many classes in the Java standard library are final, for example, `String`, `Math` and `System` in the `java.lang` packages.

### Final methods

If we only want to prevent *part of the behaviour* of a class from being changed by its subclasses, we can define the methods that realise this behaviour as final. A *final method* cannot be overridden or redefined. Such methods are marked with the keyword `final` specified before the return type in the method header. With the following declaration, we can prevent subclasses from redefining the static method `calculateOvertime()` in the `EmployeeVO` class:

```
final static double calculateOvertime(double numHours) { // (1)
 return (numHours - NORMAL_WORKWEEK);
}
```

Similarly, we can prevent subclasses from overriding an inherited instance method:

```
final double calculateFixedSalary() {
 return NORMAL_WORKWEEK * hourlyRate;
}
```

As long as the superclass method is visible and not marked with the keyword `final`, a subclass can freely override or redefine it. Many methods in the `java.lang.Object` class are final, but the `toString()`, `equals()` and `hashCode()` methods are not; allowing subclasses to implement their own versions of these methods.

### BEST PRACTICES

Methods called in a constructor should be final, otherwise subclasses can override them, resulting in unforeseen and potentially erroneous situations when objects are created at runtime.



## Overloading methods

A method in a class is said to be *overloaded* if the class or its subclasses have another method declaration with the same method name, but with different parameter list.

The method declaration given below is allowed in a subclass of the `EmployeeV0` class even if the superclass has declared the `calculateOvertime()` method as final:

```
static double calculateOvertime(int numHours) { // (2)
 return (numHours - NORMAL_WORKWEEK);
}
```

The type of the parameter `numHours` in the method at (2) is different from the type of the parameter in the superclass method with the same name shown at (1). It only accepts an `int` as a parameter value, and not a `double`. Therefore, the method at (2) does *not* override the method at (1), but instead it is an overloaded method offering a new service.

### BEST PRACTICES

Documenting properly how a method is overloaded or overridden allows clients to determine an appropriate course of action.

## 12.6 Review questions

1. Which statements are true?
  - a A subclass is more general than its superclass.
  - b If `B` is a subclass of `A`, the relationship `B is-a A` must be satisfied.
  - c A subclass that extends the properties of the superclass, must also extend the behaviour.
  - d Clients that create subclass objects can use them as if they were objects of the superclass.

[The style ReviewQuestionContinuo is not justified. Must be fixed.]

(b), (d).

(a) False. A subclass is a specialisation of the superclass.

(b) True. The relationship `B is-a A` must be satisfied if `B` is a subclass of `A`.

(c) False. A subclass that extends the properties of the superclass, *need not* extend the behaviour. New fields added to the subclass can be made directly accessible by clients. On the other hand, subclasses often extend behaviour by either offering new methods or by extending inherited methods, taking new fields into account.

(d) True. A subclass object can be used as if it was an object of the superclass, because the *is-a* relationship is satisfied.



**2.** What does it mean that a class member is *visible*?

A *visible* member of class A can be referred to by its name by any client, including subclasses of class A. Subclasses can refer to any visible member just by its name. Clients (that are not subclasses of A) must, in addition to the member name, use a reference of class A in order to access any visible member. Such clients can refer to static members that are visible by using the class name to uniquely identify the static member.

**3.** How should a subclass object initialise fields in its superclass?

- a** By assigning the desired value directly to each field.
- b** By calling one of the superclass constructors, using the `super()` call (with appropriate actual parameter values).
- c** By leaving all initialisation to the compiler.

Justify your answer by stating the advantages and drawbacks for each alternative.

(b).

The subclass should call one of the superclass constructors, because this ensures that all fields are initialised to valid and consistent values. It also makes the subclass independent of the superclass representation of properties. Assigning values directly using the field name makes a tight coupling between the subclass and superclass, meaning that changes in the superclass will necessitate changes in the subclass. The compiler will insert an implicit call to the superclass constructor, `super()` (without any parameters), if no call to a superclass constructor is explicitly specified. It is not at all sure that the fields will be initialised appropriately if one relies on an implicit call to a superclass constructor.

**4.** Given the class declarations below, how can we at (1) call the inherited method `getName()` from class Person? How can we at (2) access the inherited field variable `idNumber` from class Person?

```
class Person {
 // field variable
 int idNumber;
 String firstName;
 String lastName;
 // other declarations...
 String getName() {
 return firstName + " " + lastName;
 }
}
class Student extends Person {
 int credits;
 void printState() {
 System.out.println("Student: " + _____ // (1)
 + " has id " + _____ // (2)
 + " and " + credits + " credits");
 }
}
```



At (1) the call `getName()` can be used. At (2) the inherited field `idNumber` can be referred to by its name. Since both members are visible in the superclass, the subclass can refer to them directly by their names.

5. How does a subclass override an instance method from the superclass?
    - a It defines a method with the same name.
    - b It defines a method with the same name and parameter list.
    - c It defines a method with the same name and parameter list, but where the return type can either be the same or a subtype of the return type in the superclass.
    - d It defines a method with the same name and parameter list, but with a different return type that is not a subtype of the return type in the superclass.
- (c).

A subclass overrides an instance method from the superclass by declaring a method with same name and parameter list, but where the return type can either be the same or a subtype of the return type in the superclass.

6. What is the difference between an overridden and an overloaded method? Identify which methods are overridden and which are overloaded in the code below. (Assume that the `EmployeeV0` class is defined as in Program 12.1.)

```
class PartTimeWorker extends EmployeeV0 {
 ...
 // Instance methods
 double calculateSalary(double numHours) { // (1)
 ...
 }
 double calculateSalary() { // (2)
 ...
 }
 void printState() { // (3)
 ...
 }
 void printStateOnScreen() { // (4)
 ...
 }
}
```

An instance method with the same signature and return type overrides the corresponding method in the superclass. An overloaded method has the same name, but must have a different parameter list. The return type can be the same or different from the overloaded method.

The method `calculateSalary()` at (1) and `printState()` at (3) are thus overridden methods, while the method `calculateSalary()` at (2) is an overloaded method. The method `printStateOnScreen()` at (4) is a new method in the subclass, and therefore it is neither overridden nor overloaded.



- 7.** Explain the term *upcasting*. Give an example of upcasting using the superclass Person and the subclass Student from Question 12.4.

Upcasting is conversion of a subclass reference to a superclass reference. The following code shows upcasting of the reference of a Student object to a reference of the superclass Person:

```
Person person = new Student();
```

- 8.** Explain the term *downcasting*. Under what condition is downcasting allowed? What is the name of the operator used to determine whether a reference refers to an object of a particular type?

Downcasting is conversion of a superclass reference to a subclass reference. This is only allowed when the object referred to by the superclass reference is in fact an object of the subclass. At runtime, we can use the `instanceof` operator to check whether an object referred to by a reference is of a particular type.

- 9.** Consider Program 12.6. What happens if you remove line (3) and the corresponding closing parenthesis `()` in this `if` statement? Will the program still compile? What will happen at runtime if you for instance create a ManagerVO object?

The program will compile without errors, but if we create an EmployeeVO object during execution, the program will throw an exception and abort. If we create a ManagerVO object on the other hand, the program will execute without errors.

- 10.** Which source code lines from (1) to (4) will be accepted by the compiler? Justify your answers.

```
class Super { ... }
class Sub1 extends Super { ... }
class Sub2 extends Super { ... }
Sub1 s1 = new Sub2(); // (1)
Super sup1 = new Sub1(); // (2)
Sub2 s2 = new Super(); // (3)
s1 = (Sub1) sup1; // (4)
```

Line (1) will not be accepted by the compiler because type conversion is only allowed between subclass and superclass references, and not between two subclasses with a common superclass. Line (2) is an example of upcasting. In (3) we try to convert a superclass reference to a subclass reference (downcasting). This is not allowed because the superclass reference refers to a superclass object, which cannot be referred to by a subclass reference. Line (4) performs a successful downcasting, where the superclass reference `sup1` refers to a subclass object, that was created in line (2).

- 11.** Which statements are true?

- a** A subclass object can contain inherited and static fields.
- b** A subclass object can always refer to inherited members as if they were declared in the subclass.



- c** A client can access all inherited members in a subclass object using a reference that refers to this object.
  - d** We need not create a subclass object to access the static members of the superclass.
  - e** A superclass reference can refer to a subclass object.
  - f** A subclass reference can refer to a superclass object.
- (b), (c), (d), (e).
- (a) False. A subclass object can contain inherited fields, but not static fields. Static fields belong only to the class in which they are declared.
- (b) True. A subclass object can always refer to inherited members as if they were declared in the subclass itself, i.e. by using the member name.
- (c) True. A client can access all inherited members in a subclass object using a reference that refers to this object.
- (d) True. We do not need to create a subclass object to access the static members of the superclass.
- (e) True. A superclass reference can refer to a subclass object.
- (f) False. A subclass reference can never refer to a superclass object.

- 12.** What is a final class? Can a final class be part of an inheritance hierarchy? Justify your answer.

A final class is a class that cannot be extended through inheritance. Such classes mark the lowest levels of inheritance hierarchies, as they cannot have any descendants.

- 13.** Find the errors in the class declarations given below. Try without a compiler first!

```

class Point2D {
 double x,y;
 Point2D(double x, double y) {
 x = x;
 y = y;
 }
 void printState() {
 System.out.println("Point2D: x= " + x + " y= " + super.y);
 }
 double calculateDistance(double x2, double y2) {
 double distance = Math.sqrt((x2-x)*(x2-x)+(y2-y)*(y2-y));
 return distance;
 }
}

class Point3D extends Point2D {
 double z;
 Point3D() {
 super();
 }
}

```



```

 z = 0;
 }
 Point3D(double x, double y, double z) {
 super(x, y);
 super.z = z;
 }
 void printState() {
 System.out.println("Point3D: x= " + super.x + " y= " + super.y +
 " z= " + super.z);
 }
 double calculateDistance(double x3, double y3, double z3) {
 double distance = Math.sqrt((x3-x)*(x3-x)+
 (y3-y)*(y3-y)+(z3-z)*(z3-z));
 return distance;
 }
}

```

Is the method `calculateDistance()` in `Point3D` overridden or overloaded? Justify your answer.

Assume that the errors in the source code above have been corrected. What will happen when the following `Point3D` objects are created at runtime?

```

Point3D p1 = new Point3D(12.0); // (1)
Point3D p2 = new Point3D(); // (2)
Point3D p3 = new Point3D(7.0, 6.5, 3.2); // (3)
Point3D p4 = new Point3D(0, 0); // (4)

```

In the class `Point2D` we cannot refer to the field `y` with `super.y`. The superclass (`Object`) has no field called `y`.

In the class `Point3D`, which extends the class `Point2D`, the default constructor attempts to call the default constructor of the `Point2D` class, using the call `super()`. The compiler will issue an error message because the `Point2D` class does not have a default constructor.

Furthermore, the two attempts to refer to the field `z` in the `Point3D` class with `super.z` will not be accepted, since this field is declared in the subclass (`Point3D`) and not in the superclass (`Point2D`).

The method `calculateDistance()` in the `Point3D` class is overloaded, because it has the same name as a method in the superclass `Point2D` and the method in the subclass has a different parameter list.

The constructor call in (1) will result in an error message because the `Point3D` class does not have a constructor that takes a parameter of type `double`. The constructor call in (2) will result in the default constructor of the `Point3D` class to be called. The fields `x`, `y` and `z` in the current 3D point object will be set to the value 0.0. The constructor call in (3) will set the coordinates in the current `Point3D` object as follows: `x=7.0`, `y=6.5` and `z=3.2`. The compiler will issue an error message in (4), as both parameter values are integers and the `Point3D` class does not provide a constructor with this signature.



## 12.7 Programming exercises

1. Given the following class declarations:

```
class Film {
 String title;
 double dailyRate;
 double calculatePrice(int numDays) { ... }
}

class Video ... {
 String videoType; // "VHS" or "BetaMax"
 double calculatePrice(int numDays, double discount) { ... }
}

class Dvd ... {
 int zone;
 double deposit;
 double calculatePrice(int numDays) { ... }
}
```

Draw a UML diagram for an object of each of the three classes `Film`, `Video` and `Dvd`, where the `Film` class is the superclass of the subclasses `Video` and `Dvd`. Draw a diagram similar to that in Figure 12.4 for a `Dvd` object that is referred to by a superclass reference and a subclass reference.

2. Use the UML diagram in Exercise 12.1 as a starting point. Complete the class declarations of `Film`, `Video` and `Dvd` with:
  - a Specification of the inheritance relationship between the classes.
  - b Constructors in superclass and subclasses.
  - c The method `calculatePrice()` in the superclass `Film` and subclass `Dvd`. This method calculates the price of the loan based on the number of days the customer keeps the film. For all DVDs, the customer must pay a deposit when renting it from the store. This amount is subtracted from the price when the item is returned.
  - d The method `calculatePrice()` for the class `Video`, which takes two parameters. The first one specifies the number of days the item is rented, and the second parameter specifies a discount in percentage that is set individually for each film rented on video-tape.
3. Write a test program that uses the classes from Exercise 12.2 to perform the following operations:
  - a Creating one `Video` and one `Dvd` object.
  - b Printing the title and daily rate for these objects.
  - c Allowing the user to choose one of these objects.
  - d Printing the deposit if the `Dvd` object is chosen.
  - e Registering a return, calculating and printing the price for renting the item.



- 4.** A drawing program needs some classes for geometrical objects. For all these classes, the user wants to be able to calculate the area and the circumference of the objects.

The program must support the following types of objects: point, rectangle, circle, polygon, ellipse, square, triangle and parallelogram. We can make the following assumptions:

- a** All objects are 2-dimensional.
- b** For points, the area and circumference, by definition, is equal to 0.
- c** Polygons will not be composed of more than a fixed number of points, say seven.

Write a set of classes for the different types of geometrical objects. Make use of inheritance relationships where appropriate.

Some hints to get started:

- a** All geometrical objects have at least one set of  $x$ - and  $y$ -coordinates.
- b** Some types of objects are specialised. For instance, a triangle is a polygon with three points and a square is a rectangle with the same height and width.
- c** Some objects will need to calculate the length of an edge (line segment). In other words, we need a method that calculates the distance between two points  $(x_1, y_1)$  and  $(x_2, y_2)$ .

Since this book does not cover graphics in Java, the drawing method can simply print a message on the screen describing what type of object it is drawing.

Make the drawing program menu-driven, and let the user randomly type up to ten geometrical objects. (The maximum number of objects that the program can handle can be defined as a constant.) If the user types more objects than the program can handle, the older geometrical objects can be replaced by the new ones.

The menu must also include a drawing command. When this command is chosen, the program must call the respective drawing methods to simulate the drawing.

- 5.** The game of Nim is played as follows: multiple rows of sticks are placed on a board, and two players alternate in removing one or more sticks from one of the rows. The player who removes the last stick is the winner.

The following class can be used for so-called basic players:

```
class BasicPlayer {
 /** Returns the name of the player. Note that a player of this class
 * does not have a name. Thus, a standard name is returned.
 */
 public String getName() {
 return "basic-player";
 }
 /** Returns a Move object containing information about
 which row the player wants to remove sticks from, and
 how many sticks the player wants to remove.
 An overview of how many sticks are left in the rows is specified
 as an array of integers.
 }
}
```



```

If row 0 has 5 sticks left, rows[0] will be equal to 5.
The players of this class are naive, and do not know
how to play the game. Therefore, the player always removes
one stick from row 0.

*/
public Move nextMove(int[] rows){
 /* Creates a Move object and returns it. Sends the requests for
 * a move as argument to the constructor of the Move class.
 */
 return new Move(1, // Wants to remove 1 stick
 0); // from row 0
}

}
}

```

As we can see from the code above, a player has two major responsibilities: (1) to know his/her name, and (2) to be able to determine the next move based on the current state of the game. The `BasicPlayer` is not very intelligent and does neither of these tasks well. The idea is that this class will serve as the basis for developing (by inheritance) more clever players.

In addition to classes for players, we need the following classes to be able to play a game of Nim:

**a** The `Move` class [The paras under this item should be indented.]

The `Move` class represents a move that a player wants to perform. Suggest a design for this class and implement it.

**b** The `Game` class [The paras under this item should be indented.]

Write the class `Game` that runs a round of the game. A `Game` object represents a round of Nim and keeps track of the rows of sticks on the board, as well as which players are participating. This class should be flexible, and allow change of players and number of rows and sticks in each row, without having to modify the class. The class must have a method that controls a round of the game by asking the players in turn for their next move, and updating the board accordingly. While controlling the game, the `Game` class should print how the game is proceeding.

Here is an example of how a round of the game can proceed. The `Game` class prints the following information:

```

Row 0 (3): |||
Row 1 (5):|||||
Row 2 (7):|||||||

```

Player 1 (John) removes two sticks from row 0. After his move, the distribution of sticks is as follows:

```

Row 0 (1): |
Row 1 (5):|||||

```



Row 2 (7): |||||||

The `Game` class continues to let the players remove sticks until there are no more sticks left on the board. The player that removed the last stick wins the game. When there are no more sticks left on the board, the class should print which player is the winner.

Declare members and implement the `Game` class that satisfies the criteria outlined above.

- c The `SimpleNim` client class [The paras under this item should be indented.]

Implement the class `SimpleNim` that allows two basic players to play a round of Nim. Before the game starts, the number of rows and the number of sticks per row in the initial setup of the game board should be specified.

6. Use the classes from Exercise 12.5 to develop a slightly more advanced Nim program with the following classes:

- a Class `RealPlayer` that allows the user of the program to choose the next move. Make this class a subclass of the `BasicPlayer`.
- b Class `ComputerPlayer` that is a computer player. Find a smart algorithm that this class can use to select the next move, based on the current number of sticks in each row of the board. `ComputerPlayer` is also a subclass of the `BasicPlayer`.
- c Class `Nim` that is a client that plays a round of the game between two players of any type. In other word, each player can be a `BasicPlayer`, a `RealPlayer` or a `ComputerPlayer`. You can use this client for instance to play a round of Nim against a computer player, or to investigate who will win if two computer players play against each other.

Document all classes, and choose suitable names of their members.

7. An educational institution wants to register students enrolled in one or more courses in the current semester, as well as the lecturers teaching these courses. Personal information needed includes first name, last name and a running registration number that is used to differentiate students with the same name. For each student, the program must in addition keep track of the courses taken by the student, and how many credits he/she has accumulated from earlier exams. Lecturers need a list of the courses they are teaching in the current semester.

Propose classes for persons, students, lecturers and courses, and draw a UML diagram to show how these classes are related to one another. Write class declarations and develop a menu-driven client that allows the user to type series of students, lecturers and courses.

To avoid having to type in the title of the courses multiple times, you can declare an array of courses, and use an index value to chose a course for students and lecturers. When all the students and lecturers are registered, the program should print appropriate lists for students and lecturers, with corresponding courses (and credits for students). The program should allow exam results to typed in as pass/fail (e.g. as "yes" and "no"), and update the number of credits for each student. Finally, the program should print a list of all students identified by their student number, name



and number of credits, as well as the total number of credits passed this current semester.

Suggest which changes you would make in the program you have just written to make it more robust and/or flexible, so that it can be made a part of a student administration system. Evaluate whether any of these changes will require changes in the inheritance relationships that are defined, based on the problem description above.

## Polymorphism and interfaces

### LEARNING OBJECTIVES

By the end of this chapter, you will understand the following:

- How multiple subclasses can extend the same superclass, refining properties and/or behaviour individually.
- Using a superclass reference to access inherited members in subclass objects.
- How Java determines at runtime which method to call when subclasses have overridden methods.
- Why fields in a class can only be shadowed, and not overridden.
- Defining a contract by specifying an interface.
- The difference between an interface and a class.
- Implementing interfaces.
- What an interface reference is, and why it is polymorphic.
- The difference between an abstract and a concrete class.
- Composing new classes by aggregation.

### INTRODUCTION

Chapter 12 introduced object-oriented programming (OOP), in particular inheritance and its consequences for inherited members in subclass objects. We discuss OOP further, in particular how polymorphic references promote a particular style of programming that aids maintenance and reuse of code. We distinguish between a contract and an implementation. The former is represented by an interface, and the latter is represented by a concrete class that implements the contract. We also look at how abstract classes and final classes can be used to achieve good program design.



## 13.1 Programming with inheritance

### A superclass with multiple subclasses

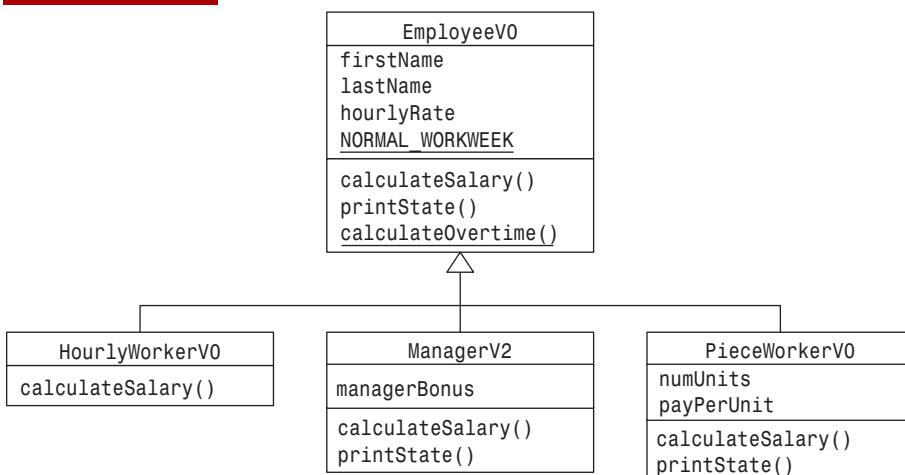
In the previous chapter we defined only one subclass from the superclass `EmployeeV0` in the program examples. We now need many types of employees in the DOT-COM company, and therefore define the following subclasses of the `EmployeeV0` class:

- `HourlyWorkerV0` — for employees earning their salary on a hourly basis.
- `ManagerV2` — for managers that get a bonus on top of their regular salary.
- `PieceWorkerV0` — for employees that work in the production department and are paid per unit produced.

We allow all subclasses to override the `calculateSalary()` method, in order to calculate the salary based on the different rules defined for each type of employee. The subclass for managers extends the state of the object with a field for the manager bonus, while the subclass for piece workers needs two new fields to store the number of units produced and the payment per unit, respectively. The subclasses `ManagerV2` and `PieceWorkerV0` also overrides the `printState()` method, so they can report on the new properties defined in these subclasses.

Figure 13.1 shows a UML diagram for the superclass and the three subclasses. All members of the superclass are visible and therefore inherited by the subclasses. The code for the three subclasses `HourlyWorkerV0`, `ManagerV2` and `PieceWorkerV0` in Figure 13.1 is shown in Program 13.1, Program 12.9 and Program 13.2, respectively.

**FIGURE 13.1 Inheritance hierarchy with multiple subclasses**



Note how assertions are used in Program 13.1 and Program 13.2 to control the values of the actual parameters in the `calculateSalary()` method of these subclasses. For simplicity the same assertion is used in all examples where an employee class is declared in this chapter. Strictly speaking the `ManagerV2` class in Program 12.9 could have left out the control of the parameter `numHours` in its `calculateSalary()` method, because the first thing this method does is to call the superclass method for salary calculation, which also



controls the value of the very same parameter. Likewise, we could have left out this assertion in the `calculateSalary()` method in the `PieceWorkerVO` class, since this method does not use the value of the `numHours` parameter. However, we have decided to retain the assertion, to uphold the contract for the `calculateSalary()` method as defined by the superclass.

### PROGRAM 13.1 The subclass `HourlyWorkerVO`

```
/**
 * An hourly worker is paid for each hour she/he has worked in
 * the current week, based on a fixed rate per hour.
 */
class HourlyWorkerVO extends EmployeeVO {
 // Constructors
 /**
 * Creates an hourly worker with default values from the superclass.
 */
 public HourlyWorkerVO() { // explicit default constructor
 super();
 }
 /**
 * Creates an hourly worker with the given name and hourly rate.
 * @param firstName First name (and any middle names) of the worker.
 * @param lastName Last name of the worker.
 * @param hourlyRate Payment per hour in GBP.
 */
 public HourlyWorkerVO(String firstName, String lastName,
 double hourlyRate) {
 super(firstName, lastName, hourlyRate);
 }

 // Instance method
 /**
 * Calculates the weekly salary for the hourly worker.
 * This method overrides the method in the EmployeeVO class.
 * @param numHours The number of hours worked in the current week.
 * @return Weekly salary in GBP for the current week.
 */
 public double calculateSalary(double numHours) {
 // Using assert to check the value of the actual parameter.
 assert numHours >= 0 : "Number of hours must be >= 0";
 return numHours * hourlyRate;
 }
}
```



## PROGRAM 13.2 The subclass **PieceWorkerVO**

```

/**
 * A piece worker is paid for each unit she/he produces.
 */
class PieceWorkerVO extends EmployeeVO {
 // Field variables
 double payPerUnit;
 int numUnits;

 // Constructors
 /**
 * Creates a piece worker with default values.
 */
 public PieceWorkerVO() { // explicit default constructor
 super();
 }
 /**
 * Creates a piece worker with the given name.
 * All other fields are set to default values implicitly.
 */
 public PieceWorkerVO(String firstName, String lastName) {
 super(firstName, lastName, 0.0);
 }
 /**
 * Creates a piece worker with the given name, payment per unit and
 * number of units produced in the current week.
 */
 public PieceWorkerVO(String firstName, String lastName,
 double pay, int count) {
 super(firstName, lastName, 0.0);
 payPerUnit = pay; // Stores extended state.
 numUnits = count;
 }

 // Instance methods
 /**
 * Calculates the weekly salary for the piece worker.
 * This method overrides the method in the EmployeeVO class.
 * @param numHours The number of hours worked in the current week.
 * This parameter is not used, but must be given for the class to
 * satisfy the same contract as the superclass.
 * @return Weekly salary in GBP for the current week.
 */
 public double calculateSalary(double numHours) {
 // Using assert to check the value of the actual parameter
 assert numHours >= 0 : "Number of hours must be >= 0";
 return calculateSalary();
 }
}

```



```

/**
 * Prints the name, payment per unit and number of units produced
 * for the piece worker.
 * This method overrides the method in the EmployeeV0 class.
 */
void printState() {
 super.printState();
 System.out.printf("Has produced %d unit at %.2f GBP/piece%n",
 numUnits, payPerUnit);
}
/**
 * Calculates the weekly salary for the piece worker.
 * This method extends the contract of the superclass.
 * @return Weekly salary in GBP for the current week.
 */
public double calculateSalary() {
 return numUnits * payPerUnit;
}
}

```

---

## Polymorphism and polymorphic references

A superclass reference can refer to objects of the superclass and its subclasses at runtime. If the subclass overrides an inherited method, and we call this method on an object of the subclass using a superclass reference, it is the method defined in the subclass that will be executed, and not the one in the superclass. This mechanism is called *polymorphism*, and is a consequence of inheritance and overriding. The word polymorphism stems from Greek, and can be translated as "the ability to take on many forms". The reference `employee` in Program 13.3 is called a *polymorphic reference*.

Program 13.3 illustrates polymorphic behaviour of superclass references. The program creates an `employee` object based on the user input. Regardless of the type of the `employee` object created, it is referenced by the superclass reference `employee`. This superclass reference is used to call methods on this object at (7) and at (8). The compiler has no way of knowing which object a superclass reference will refer to at runtime and therefore cannot resolve which method implementation should be called. The choice of the method is deferred till runtime, and determined based on the *type of the object* and the *method signature*. This mechanism is called *dynamic method lookup*.

The method call in line (7) has the signature `printState()`. If the reference `employee` refers to an object of the `EmployeeV0` class or the `HourlyWorkerV0` class, the method `printState()` declared in the `EmployeeV0` class will be executed. If the reference `employee` on the other hand refers to an object of the `ManagerV2` class, the overridden method in the `ManagerV2` class will be executed. Similarly, the `printState()` method in the `PieceWorkerV0` class will be executed if the reference `employee` refers to an object of this subclass. In other words, the object type determines which implementation of an inherited method will be executed, not the type of the reference.



At (8) in Program 13.3, the superclass reference `employee` is used to call the method `calculateSalary()`, that is overridden in all subclasses of the `EmployeeV0` class. Here, the overridden methods will be executed if the reference `employee` refers to a subclass object, and the `calculateSalary()` method in the superclass will be executed if the reference refers to an `EmployeeV0` object.

On the other hand, if a superclass reference is used to refer to a field that is shadowed in a subclass, it is the *type of the reference* that determines which field is accessed. This is determined by the compiler based on the class declaration. This means that it is not possible to use a superclass reference to refer to a shadowed field in the subclass, even if the reference refers to an object of the subclass at runtime. That is why it is not possible to override fields.

### PROGRAM 13.3    Superclass references handling objects of multiple subclasses

```
import java.util.Scanner;
/**
 * Uses polymorphism to calculate the weekly salary of any employee.
 */
public class SalaryManager {
 public static void main(String[] args) {

 Scanner keyboard = new Scanner(System.in);

 // Choose which type of employee to create.
 System.out.println("Types of employees:");
 System.out.println(" 1 - Employee");
 System.out.println(" 2 - Hourly worker");
 System.out.println(" 3 - Manager");
 System.out.println(" 4 - Piece worker");
 System.out.print("Choose a category: ");
 int employeeCode = keyboard.nextInt(); // (1)
 EmployeeV0 employee; // superclass reference
 // Creates an employee of the chosen type
 switch (employeeCode) {
 case 1:
 employee = new EmployeeV0("John", "Doe", 30.0); // (2)
 break;
 case 2:
 employee = new HourlyWorkerV0("Mary", "Smith", 35.0); // (3)
 break;
 case 3:
 employee = new ManagerV2("John D.", "Boss", 60.0, 105.0); // (4)
 break;
 case 4:
 employee = new PieceWorkerV0("Ken", "Jones", 12.5, 75); // (5)
 break;
 default:
 System.out.println("Invalid category - creates an EmployeeV0");
 }
 }
}
```



```

employee = new EmployeeVO("Mark", "Spencer", 25.0); // (6)
break;
}

// Prints the state and weekly salary for the employee
System.out.println("Employee information:");
employee.printState(); // (7)
System.out.printf("Weekly salary: %.2f GBP%n",
 employee.calculateSalary(42.5)); // (8)
}
}

```

Program output when a manager object is created:

Types of employees:

- 1 - Employee
- 2 - Hourly worker
- 3 - Manager
- 4 - Piece worker

Chose a category: 3

Employee information:

Manager John D. Boss has an hourly rate of 60.00 GBP  
and a bonus of 105.00 GBP for 37.5 hours work

Weekly salary: 2955.00 GBP

Program output when an hourly worker object is created:

Types of employees:

- 1 - Employee
- 2 - Hourly worker
- 3 - Manager
- 4 - Piece worker

Chose a category: 2

Employee information:

Employee Mary Smith has an hourly rate of 35.00 GBP

Weekly salary: 1487.50 GBP

## Arrays of polymorphic references

Program 13.3 showed that an `EmployeeVO` reference can refer to different types of employee objects. By creating an array of such references, we can refer to all employees in a company, and calculate the salary of every employee by traversing this array. Program 13.4 shows an example of using an array of polymorphic references.

To simplify the example, two arrays of employees and hours worked are created and explicitly initialised at (1) and (2), respectively. In a real-world program, data will typically be read from a file or be obtained from some other data source, such as a database.

At (3) and at (4), the expressions `empArray[i].firstName` and `empArray[i].lastName` refer to the two inherited fields `firstname` and `lastName`, respectively. The type of the refer-



ences will determine which fields are accessed, i.e. those in the `Employee0` superclass. At (5), the type of the object whose reference value is stored in the *i*-th array element will determine which `calculateSalary()` method is called. For the objects of the three subclasses, this means their overridden method for calculating the weekly salary is executed.

Figure 13.2 illustrates the array of polymorphic references after initialisation in Program 13.4. Each element of the array is a polymorphic reference that can refer to objects of different types of employees.

#### PROGRAM 13.4 Handling arrays of employees using polymorphic references

```
/*
 * Calculates the salary for different types of employees.
 */
class EmployeeArray {
 static EmployeeV0[] empArray = { // (1)
 new EmployeeV0("John", "Doe", 30.0),
 new HourlyWorkerV0("Mary", "Smith", 35.0),
 new PieceWorkerV0("Ken", "Jones", 12.5, 75),
 new ManagerV2("John D.", "Boss", 60.0, 105.0),
 };

 static double[] empHours = { 37.5, 25.0, 30.0, 45.0 }; // (2)

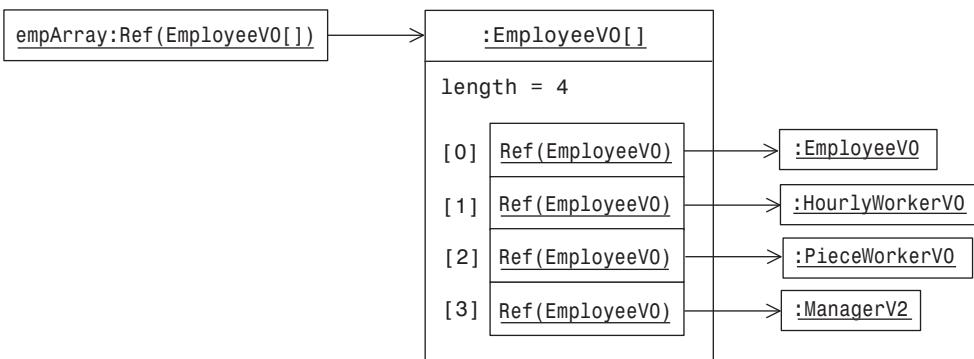
 public static void main(String[] args) {
 // Traverses the employee array and prints selected properties for each
 // employee in the array.
 for (int i=0; i<empArray.length; i++){
 System.out.printf(
 "Employee no. %d: %s %s has a weekly salary of %.2f GBP%n",
 i+1,
 empArray[i].firstName, // (3)
 empArray[i].lastName, // (4)
 empArray[i].calculateSalary(empHours[i])); // (5)
 }
 }
}
```

Program output:

```
Employee no. 1: John Doe has a weekly salary of 1125.00 GBP
Employee no. 2: Mary Smith has a weekly salary of 875.00 GBP
Employee no. 3: Ken Jones has a weekly salary of 937.50 GBP
Employee no. 4: John D. Boss has a weekly salary of 3255.00 GBP
```



**FIGURE 13.2** Array of polymorphic references



## Consequences of polymorphism

A superclass reference can be used to call methods that are inherited or overridden by the subclasses. With dynamic method lookup, such a method call can result in the execution of different method implementations in different subclasses.

From a client's perspective, subclass objects can be treated as if they were superclass objects because they fulfil the superclass contract, either by overriding or inheriting methods from the superclass. This means that the client does not need code for special treatment of each new subclass that is defined, as long as the client only uses properties and behaviour defined by the superclass contract.

### BEST PRACTICES

Use polymorphic references to access common behaviour in classes related through inheritance. This style of programming results in code that is concise, easy to review and maintain, since no special code is needed to handle each subclass individually.

## Inheritance is transitive

The inheritance relationship is *transitive*, i.e. if the `SalesManager` class is a subclass of the `ManagerV2` class, and the `ManagerV2` class is a subclass of the `EmployeeVO` class, then the `SalesManager` class is also a subclass of the `EmployeeVO` class. The `SalesManager` class inherits from all its superclasses: `ManagerV2`, `EmployeeVO` and `Object`. A superclass reference can refer to objects of any of its subclasses transitively.

In Program 13.5 an array of `Object` references is used to handle different types of objects. The array `objectArray` is initialised with different objects from (1) to (4), and contains references to `Integer`, `Employee`, `Double` and `String` objects, respectively. Since the element type is `Object`, we can call methods that are either inherited or overridden from the `Object` class, such as `getClass()` and `toString()` at (5). The method `toString()` is overridden by all the objects in the array, but not the `getClass()` method, which is declared as a final method in the `Object` class. In the loop at (5), the `toString()` method



executed is from each individual object, but the inherited `getClass()` method is always executed from the `Object` class. The program output shows the class of the respective objects in the array.

The `getClass()` method of the `Object` class returns the object that at runtime represents the class of the object on which the method is called. For example, calling the `getClass()` on a `String` object returns the object that represents the `String` class at runtime. Calling the `toString()` method on this object returns the name of the class it represents, in this case, the string "class `java.lang.String`". The call to the `toString()` method on this object is implicit at (5) in Program 13.5.

### PROGRAM 13.5 Polymorphic Object references

```
public class ObjectReport {
 public static void main(String[] args) {
 Object[] objectArray = {
 10, // (1) Integer
 new Employee("Tiny", "Tim", 100.0), // (2) Employee
 12.5, // (3) Double
 "no measurements" // (4) String
 };
 for (Object obj : objectArray) {
 System.out.println(obj.getClass() + ": " + obj.toString()); // (5)
 }
 }
}
```

Program output:

```
class java.lang.Integer: 10
class Employee: First name: Tiny Last name: Tim Hourly rate: 100.00 GBP
class java.lang.Double: 12.5
class java.lang.String: no measurements
```

## 13.2 Abstract classes

### Abstract and concrete classes

An *abstract class* is a class for which no objects can be created. Such a class is used as a superclass to enforce subclasses adhering to a common contract, forcing the subclasses to implement any missing behaviour required by the superclass. For example, an abstract superclass for employees may decide that it wants to force all its subclasses to implement their own method for calculating the weekly salary, but provides the implementation of other common behaviour, such as printing the state of an employee object.

An abstract class may choose to provide partial or complete implementation of its contract. The behaviour that an abstract class wants its subclasses to implement is specified by abstract methods. Program 13.6 shows the declaration of the abstract class `EmployeeVA`. An



abstract class is specified with the keyword `abstract`, and any abstract methods in the class are also explicitly designated with the same keyword. The implementation of the `calculateSalary()` method is now deferred to any subclass of the class `EmployeeVA`.

### PROGRAM 13.6 An abstract class with an abstract method

```
abstract class EmployeeVA {

 // Except for the abstract method below,
 // the rest of the specification is as in the EmployeeVO class.

 /**
 * Calculates the weekly salary for the employee.
 * This method is abstract. Subclasses must implement this method.
 * @param numHours Number of hours worked in current week.
 * @return Weekly salary in GBP for the current week.
 *
 * @pre numHours >= 0.0
 * @post salary > 0.0
 */
 abstract public double calculateSalary(double numHours);

}
```

A class need not declare any abstract methods to be considered abstract. As long as the class is specified with the keyword `abstract`, clients are prevented from creating objects of the class. In general, an abstract class will have at least one abstract method. The subclasses must provide an implementation for *all* the abstract methods, otherwise they are also considered to be abstract.

A class that is not abstract, is called a *concrete class*. Classes we have considered so far have been concrete classes. Abstract classes are only different from concrete classes in two respects: abstract classes can specify abstract methods, and we cannot create objects of abstract classes.

#### BEST PRACTICES

Define the contract of an abstract class using pre- and postconditions, in the same way as for an interface. Document also the contract for the methods whose implementation is provided by the abstract class.



## Implementing abstract methods in subclasses

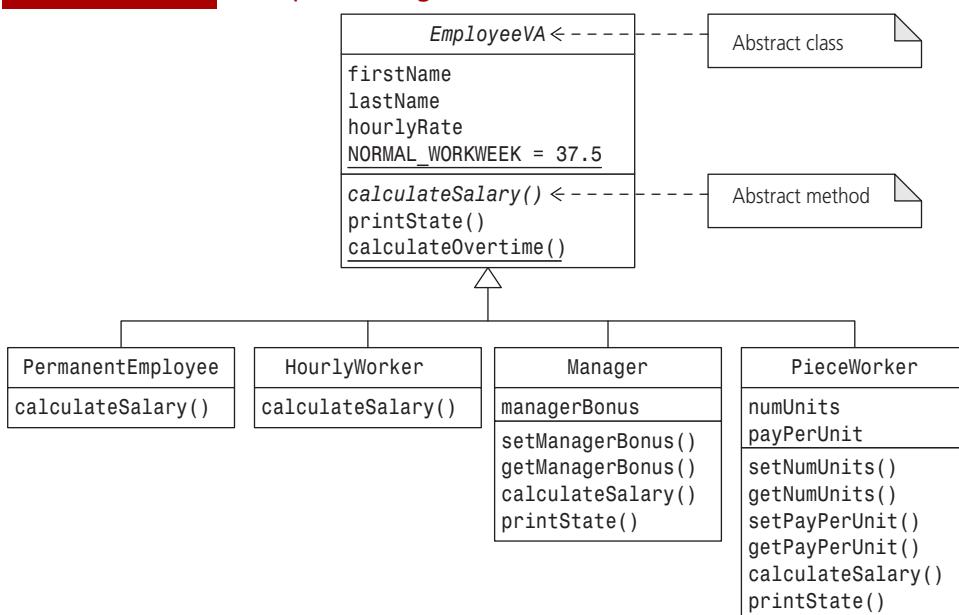
Figure 13.3 shows an inheritance hierarchy for different types of employees, where the superclass `EmployeeVA` is abstract. In the `EmployeeVA` class, the `calculateSalary()` method is abstract, and therefore all subclasses must implement this method.

The subclass `PermanentEmployee` provides an implementation of the `calculateSalary()` method, as shown in Program 13.7. Apart from the implementation of this method, the `PermanentEmployee` class does not declare any new members. The objects of this class inherit members from the superclass `EmployeeVA` in the usual way. Note that the implementation of the `calculateSalary()` method in the subclass is also a form of overriding the abstract method declaration in the superclass.

Clients cannot create objects of the superclass `EmployeeVA`, but they can still declare and use references of this type. These superclass references can be used to manipulate objects of subclasses. This is illustrated in Program 13.8, where a `PermanentEmployee` object is created and its reference value is assigned to a reference of type `EmployeeVA` at (1). This reference is used to retrieve the value of the field `hourlyRate` at (2) and to calculate the weekly salary at (3), where the polymorphic behaviour of the superclass reference ensures that the `calculateSalary()` method from the `PermanentEmployee` class is executed.

FIGURE 13.3

Implementing abstract methods in subclasses



PROGRAM 13.7

A subclass extending an abstract superclass

```

final class PermanentEmployee extends EmployeeVA {
 // Constructor
 public PermanentEmployee(String firstName, String lastName, double rate) {
 super(firstName, lastName, rate);
 }
}

```



```
// Instance method
/**
 * Calculates the weekly salary for the employee.
 * This method implements the abstract method in the EmployeeVA class.
 * @param numHours Number of hours worked in current week.
 * @return Weekly salary in GBP for the current week.
 */
public double calculateSalary(double numHours) {
 assert numHours >= 0 : "Number of hours must be >= 0"; // @pre
 double salary = NORMAL_WORKWEEK * hourlyRate;
 assert salary >= 0.0 : "Weekly salary must be >= 0.0"; // @post
 return salary;
}
}
```

---

### PROGRAM 13.8 Using a reference of an abstract class

```
// Using an abstract class.
public class EmployeeClient {
 public static void main(String[] args) {
 // EmployeeVA employee = new EmployeeVA(); // Compile-time error,
 // EmployeeVA is abstract
 EmployeeVA employee = new PermanentEmployee("Mel", "Ross", 40.0); // (1)
 System.out.println("Information about a permanent employee:");
 System.out.printf(
 "Hourly rate is %.2f GBP and weekly salary is %.2f GBP%n",
 employee.hourlyRate, // (2)
 employee.calculateSalary(42.0)); // (3)
 }
}
```

Program output:

```
Information about a permanent employee:
Hourly rate is 40.00 GBP and weekly salary is 1500.00 GBP
```

---

### Abstract classes in the Java standard library

The Java standard library provides numerous examples of abstract classes, but the most notable one in our context is the `java.lang.Number` class. This class defines abstract methods (`doubleValue()`, `floatValue()`, `intValue()`, `longValue()`) for converting the numerical value in a subclass object to any one of the numerical data types `double`, `float`, `int` and `long`. Subclasses `Double`, `Float`, `Integer` and `Long` of the superclass `Number` provide implementations for these methods. Thus, a call to any of these methods to convert the `int` value in an `Integer` object to one of the numerical types, will result in the method implementation in the `Integer` class to be executed.



An abstract class must be extended by inheritance in order to make practical use of its services, but a final class cannot be extended by inheritance. Therefore, a class cannot be declared both `abstract` and `final`.

### 13.3 Aggregation

What is the alternative if a class `B` fails the litmus test for design by inheritance and cannot inherit from class `A`? In that case we know that not all services from class `A` are appropriate for class `B`. The solution is to use design by *aggregation*. Class `B` declares a field of type `A`, and uses the appropriate services from class `A` to implement its own behaviour. Class `B` is thus able to filter out any unwanted behaviour from class `A`. An example where design by aggregation is chosen rather than design by inheritance can be found in Program 16.4 on page 529.

It is not unthinkable that class `B` might require services from several other classes, and declares appropriate fields for this purpose in its declaration. In this design approach, an object of class `B` can have several field references that refer to other objects at runtime. An object of class `B` is called a *composite object* and is thus associated with many *part objects*. The part objects can in turn be associated with other (part) objects.

Design by aggregation can also be used when our class requires services from an object of a final class. The class `String` is final and therefore cannot be extended. In many of the classes that we have written so far, we have applied aggregation to make use of `String` objects. Here is another example where the class `Square` uses aggregation to represent a square by an object of the final class `Rectangle`. The class `Square` declares a field of type `Rectangle` and uses a `Rectangle` object as a square:

```
final class Rectangle {
 // Sides of the rectangle:
 double length;
 double width;

 public Rectangle(double length, double width) {
 assert length >= 0.0 && width >= 0.0 : "Dimension cannot be negative.";
 this.length = length;
 this.width = width;
 }
 public double getLength() { return length; }
 public double getWidth() { return width; }
 public double area() { return length * width; }
}
class Square {
 Rectangle rectangle; // Using aggregation.

 public Square(double sideLength) { // All sides are equal.
 rectangle = new Rectangle(sideLength, sideLength);
 }
 // Uses appropriate methods from the Rectangle class to implement a square.
 public double getSideLength() { return rectangle.getLength(); }
}
```



```
public double area() { return rectangle.area(); }
}
```

## BEST PRACTICES

If the litmus test for design by inheritance fails, use design by aggregation. Design by aggregation allows part objects of a composite object to be changed without effecting the clients of the composite object.

## 13.4 Interfaces in Java

### The concept of contracts in programs

A *contract* specifies the behaviour of a class, i.e. what services the methods of the class provide in terms of the parameter values accepted and the values returned by these methods. It is also common to include special conditions in the contract of each method: *preconditions*, which are conditions that must be satisfied at the start of method execution, and *postconditions*, which are conditions that must be satisfied at the end of method execution.

We will use special Javadoc tags, `@pre` and `@post`, in the documentation comment of a method to specify the contract conditions. These conditions are tested using *assertions* at relevant places in the code. In the document comment of the method `calculateSalary()` below, relevant information about the contract is specified by the parameters accepted by the method (`@param` tag), the value returned by the method (`@return` tag), the pre- and post-conditions stated using the `@pre` and `@post` tags, and assertions used in the code to test the contract conditions:

```
/**
 * Calculates the weekly salary for the employee.
 * @param numHours Number of hours worked in current week.
 * @return Weekly salary in GBP for the current week.
 *
 * @pre numHours >= 0.0
 * @post salary >= 0.0
 */
double calculateSalary(double numHours) {
 assert numHours >= 0.0 : "Number of hours must be >= 0.0"; // @pre
 double salary = ...
 ...
 assert salary >= 0.0 : "Weekly salary must be >= 0.0"; // @post
}
```



## BEST PRACTICES

Specify the contract of each method in your program by defining the preconditions and postconditions that must be satisfied in order for the contract to be fulfilled. Use assertions to test the contract conditions at appropriate places in the code.

### Declaring an interface

An *interface* in Java specifies a group of related methods, but without their implementation. The interface only specifies the contract for each method: the parameter values accepted and the value returned by the method. There is no implementation of the method body. This form of method declaration is called an *abstract method*. It specifies the signature of the method and the return type.

Program 13.9 shows an example of a simple interface. The keyword `interface` is used to designate an interface specification. The interface `ISportsClubMember` only specifies one abstract method at (1). Note that the abstract method in Program 13.9 is *not* marked with the keyword `abstract`, even though it is abstract.

An interface allows us to formalise a contract, even though the contract conditions are only embedded as documentation, since no implementation can be provided. For that reason, we cannot create objects from interfaces. An interface is therefore in itself not very useful. Interfaces allow us to group services into contracts, but classes have to implement them in order make full use of their potential.

An interface can also declare constants, that are implicitly `static` and `final`. There is seldom any reason to use an interface for this purpose, as the enum types provide a better solution for defining constants.

#### PROGRAM 13.9 A simple interface

```
interface ISportsClubMember {
 /**
 * Calculates the membership fee.
 * @post fee >= 0.0
 */
 double calculateFee(); // (1) Abstract method
}
```

## BEST PRACTICES

It is a common practice to prefix the names of interfaces with the letter 'I' in order to distinguish them from classes.



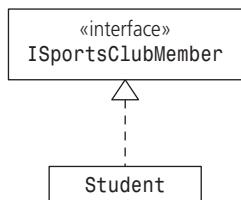
## Implementing an interface

If a class wants to implement the contract specified by an interface, the class must specify the name of the interface in its class header using the `implements` clause. We say that the class *implements the interface*. The class must also implement all the abstract methods of the interface. If it does not, then the class must be declared `abstract`, as some of the method implementations will be missing, and we will not be allowed to create objects of the class (see Section 13.2 on page 370).

Program 13.10 shows the class `Student` that implements the `ISportsClubMember` interface. The UML diagram in Figure 13.4 illustrates the relationship between the `ISportsClubMember` interface and the `Student` class. The class `Student` satisfies the contract represented by the interface, as it uses the `implements` clause to specify the interface name at (1), and implements the abstract method `calculateFee()` at (2). The class can provide additional services, but these are irrelevant with regard to the `ISportsClubMember` interface. The point to note is that other classes can also implement the `ISportsClubMember` interface and provide their own implementation of the abstract method `calculateFee()`.

**FIGURE 13.4**

Class implementing an interface



**PROGRAM 13.10**

Class implementing an interface

```

class Student implements ISportsClubMember { // (1)

 // Field variables
 String firstName;
 String lastName;
 int credits;

 // Constructors
 /**
 * Creates a student with the specified name and credits.
 */
 public Student(String firstName, String lastName, int credits) {
 this.firstName = firstName;
 this.lastName = lastName;
 this.credits = credits;
 }

 /**
 * Prints the name and number of credits for the student.
 */

```



```

void printState() {
 System.out.printf("Student %s %s has %d credits.%n",
 firstName, lastName, credits);
}

// Implements the interface
/**
 * Calculates the membership fee.
 * @post return >= 0.0
 */
public double calculateFee() { // (2)
 double fee = 10.0;
 assert fee >= 0.0 : "Membership fee for students must be >= 0.0"; // @post
 return fee;
}
// Other members...
}

```

## Implementing multiple interfaces

A class can implement *multiple* interfaces. In other words, Java supports *multiple inheritance from interfaces*, but only *single inheritance from classes*. Note that *implementation of methods* is only inherited from classes. In the DOT-COM company some types of employees can be board members. The contract needed for being a board member can be specified as an interface, `IBoardMember`. Then, if managers can be both union members and board members, we can declare the manager class as follows, with both the interfaces specified in a comma-separated list in the `implements` clause:

```

class ManagerV4 extends EmployeeV0 implements IUnionMember, IBoardMember {
 // Other members... (as in ManagerV2)

 // Implementation of the IUnionMember interface
 // ...

 // Implementation of the IBoardMember interface
 // ...
}

```

If hourly workers do not qualify to be board members, but can only be union members, the `HourlyWorkerV1` class need only implement the `IUnionMember` interface:

```

class HourlyWorkerV1 extends EmployeeV0 implements IUnionMember {
 // Other members... (as in HourlyWorkerV0)

 // Implementation of the IUnionMember interface
 // ...
}

```



Given the class declarations above, a manager is both a union member and a board member, whereas an hourly worker is a union member. Note that both managers and hourly workers are still employees, as they both extend the `EmployeeV0` class.

## Inheritance between interfaces

An interface `B` can extend another interface `A`. Then, `A` is a *superinterface* of `B` and `B` is a *subinterface* (or *derived interface*) of `A`. The subinterface inherits the member declarations from the superinterface, and can also declare new members, in a manner similar to a subclass extending the behaviour and properties specified by its superclass. An interface can only extend *one* interface. In other words, Java only supports single inheritance between interfaces.

Class `X` implementing subinterface `B`, must implement the abstract methods from both superinterface `A` and subinterface `B`:

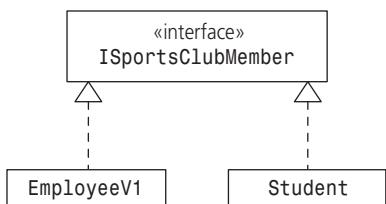
```
interface A { void methodA(); }
interface B extends A { void methodB(); }
class X implements B {
 void methodA() { ... }
 void methodB() { ... }
 // ...
}
```

## Polymorphic use of interface references

A Java interface is not a class, and we cannot create objects of an interface type. On the other hand, an interface defines a new reference type, and we can declare references of an interface type. These references can refer to objects of *all* classes that implement the interface. This also includes objects of subclasses that inherit the implementation from a superclass that implements the interface. A reference of an interface type is therefore *polymorphic* with respect to objects of classes implementing the interface.

Figure 13.5 shows that the `ISportsClubMember` interface (Program 13.9) is implemented by the two classes `Student` (Program 13.10) and `EmployeeV1` (Program 13.11). These two classes are otherwise not related, but still have some common behaviour, since both fulfil the same contract represented by the `ISportsClubMember` interface. A reference of the interface type `ISportsClubMember` can refer to objects of both the `EmployeeV1` and the `Student` class.

**FIGURE 13.5** Two classes implementing the same interface





Program 13.12 illustrates using interface references. It creates two employee and two student objects at (1). It then creates three arrays and initializes them with these objects at (2): one array stores just two employees, another one stores just two students and the third one stores both an employee and a student. At (4), three calls are made to the method `printFee()` declared at (5), passing each of the arrays in turn. The output from the program shows the class of each object that is in the array passed to the method `printFee()`, and the membership fee calculated for this object. We see that the correct implementation of the method `printFee()` is executed, depending on whether the object was an employee or a student.

The method `printFee()` takes a parameter of the array type `ISportsClubMember[]`. Since the array type `ISportsClubMember[]` is the *supertype* of both the `EmployeeV1[]` and `Student[]` array types, an array of these types can be passed to the method. Each element of the `members` array in the method `printFee()` acts as a polymorphic reference of type `ISportsClubMember`, and can manipulate any object that implements this interface. Note that the method `printFee()` is not exclusively for employees and students. It will accept *any* array whose objects implement the `ISportsClubMember` interface.

A reference of an interface type can *only* be used to call methods specified in the interface. The compiler will report an error if any attempt is made to access a member not declared in the interface.

### BEST PRACTICES

Program using references of interface types. That way *any* suitable objects that implement these interfaces can be used in the program.

### PROGRAM 13.11 Implementing the `ISportsClubMember` interface

```
/**
 * An employee is paid by the hour at a fixed hourly rate.
 * However, for each hour worked overtime during the week,
 * the hourly rate is doubled.
 * In addition, the employee is a member of a sports club.
 */
class EmployeeV1 implements ISportsClubMember {

 // Static variable, fields and constructors as in the EmployeeV0 class.

 // Static method calculateOvertime(), instance methods calculateSalary()
 // and printState() as in the EmployeeV0 class.

 // Implements the ISportsClubMember interface
 /**
 * Calculates the membership fee.
 */
 public double calculateFee() {
```



```

 double fee = 20.0;
 assert fee >= 0.0 : "Membership fee for employees must be >= 0.0"; // @post
 return fee;
 }
}

```

---

### PROGRAM 13.12 Using interface references

```

class UsingInterfaceReferences {
 public static void main(String[] args) {
 // (1) Create 2 employees and 2 students:
 EmployeeV1 employee1 = new EmployeeV1("Al", "Hansen", 325.0);
 EmployeeV1 employee2 = new EmployeeV1("Jackie", "Jones", 300.0);
 Student student1 = new Student("Peter", "Jablonski", 35);
 Student student2 = new Student("Lars", "Larsen", 30);

 // (2) Create 3 arrays:
 // one with employees only,
 // one with students only, and
 // one with both employees and students:
 EmployeeV1[] empArray = {employee1, employee2};
 Student[] studArray = {student1, student2};
 ISportsClubMember[] memArray = {employee1, student2};

 // (3) Print the fees for arrays created in (2):
 System.out.println("Employees only:");
 printFee(empArray);
 System.out.println("Students only:");
 printFee(studArray);
 System.out.println("Sports club members:");
 printFee(memArray);
 }

 // Takes an array of ISportsClubMembers and prints the fee for each member.
 static void printFee(ISportsClubMember[] members) { // (4)
 for (ISportsClubMember member : members) {
 System.out.printf(member.getClass().getSimpleName() + // (5)
 ": %.2f GBP%n",
 member.calculateFee());
 }
 }
}

```

Program output:

```

Employees only:
EmployeeV1: 20.00 GBP
EmployeeV1: 20.00 GBP
Students only:

```



Student: 10.00 GBP  
 Student: 10.00 GBP  
 Sports club members:  
 EmployeeV1: 20.00 GBP  
 Student: 10.00 GBP

---

## 13.5 Review questions

1. Every superclass reference is a \_\_\_\_\_ reference.  
 Every superclass reference is a *polymorphic* reference.
2. A superclass reference can refer to objects of both the \_\_\_\_\_ as well as its \_\_\_\_\_.  
 A superclass reference can refer to objects of both the *superclass* as well as its *subclasses*.
3. If we use a superclass reference sup to refer to a subclass object, which of the following members can we then refer to? What if we use a subclass reference sub?  
 a Members declared in the subclass that are visible.  
 b All members declared in the superclass that are visible.  
 c All members inherited from the superclass.  
 (a) is true for a subclass reference only, (b) and (c) are true for both superclass and subclass references, as explained below:  
 a The superclass sup cannot be used to refer to members declared in the subclass. However, we can use the subclass reference sub to refer to those members of the subclass that are visible.  
 b Since members in the superclass that are visible are inherited by the subclass, both the references sup and sub can be used to refer to such members.  
 c Same as b.
4. Which members can be referred to by a subclass reference? Assume that all members in the superclass and subclass are visible.  
 With a subclass reference we can refer to all members in both the subclass and the superclass. Visible members of the superclass are inherited by the subclass.

5. What are the main differences and similarities between an interface and an abstract class?

An interface specifies a contract guaranteeing a specific behaviour, but does not provide any implementation. An abstract class also specifies a particular behaviour, but may provide partial or complete implementation. An abstract class can contain fields and static variables, whose values can be changed during program execution, while an interface can only declare static variables that are implicitly final.



6. Given an interface `I`, an abstract class `A` that implements the interface `I`, and a subclass `B` of class `A`. Which of the following statements are true?
- a We can create objects of type `I`.
  - b A reference of type `A` can refer to an object of type `I`.
  - c A reference of type `I` can refer to an object of type `A`.
  - d A reference of type `I` can refer to an object of type `B`.
  - e A reference of type `I` can be used to call all methods that the subclass `B` inherits from its superclass `A`.
- (c), (d).
- a** False. We can *not* create objects of type `I` because it is not possible to create objects of an interface type.
- b** False. A reference of type `A` *cannot* refer to an object of type `I`. Even if we could create an object of the interface type, we cannot use a reference of a class that implements the interface to refer to such objects.
- c** True. A reference of type `I` can refer to an object of type `A`. An interface reference can refer to objects of *all* classes implementing the interface. It is polymorphic.
- d** True. Interface references are polymorphic, and thus a reference of type `I` can refer to a subclass object of type `B`.
- e** False. A reference of type `I` cannot be used to call all methods that the subclass `B` inherits from the superclass `A`. This is the case for all inherited methods that are not declared in the interface `I`.

7. How can we access at (1) the value of the constant `HOURS_TO_MINUTES` in the code below? How could we have referred to this constant if the `TimeCalculationClient` class had implemented the `ITimeCalculations` interface?

```
interface ITimeCalculations {
 double HOURS_TO_MINUTES = 60.0;
 double MINUTES_TO_HOURS = 1.0/60.0;
 double toMinutes(double hours);
 double toHours(double minutes);
}

public class TimeCalculationClient {
 public static void main(String[] args) {
 System.out.printf("There are %d minutes in an hour%n",
 _____); // (1)
 }
}
```

The constant can be accessed using the form `ITimeCalculations.HOURS_TO_MINUTES` since the `TimeCalculationClient` class does not implement the interface. If the class had implemented the interface, we could have just used the name of the constant to access its value.

8. Find the compile-time errors in the following code. (Try without a compiler first.)



```

// (1) An interface
interface IClubMember {
 double feePercentage = 0.012;
 double calculateFee(double sum) {
 feePercentage += 0.002;
 return feePercentage*sum;
 }
}

// (2) Class implementing the interface
class Worker implements IClubMember {
 public double calculateFee(double annualSalary) {
 feePercentage += 0.001;
 return feePercentage*annualSalary;
 }
 double getFeePercentage() { return feePercentage; }
}

// (3) Client using an interface reference
public class ClubClient {
 public static void main(String[] args) {
 IClubMember member = new IClubMember();
 IClubMember worker = new Worker();
 System.out.printf("Fee: %.2f GBP%n", worker.calculateFee(35000.0));
 System.out.printf("Percentage: %d%n", worker.getFeePercentage());
 }
}

```

**Errors in the `IClubMember` interface:** It is not allowed to implement any methods in an interface, nor is it allowed to change the value of a constant in an interface. Variable declarations in an interface are by definition `static` and `final`.

**Errors in the `Worker` class:** Changing the value of constants in an interface is not allowed, as they are final.

**Errors in the `ClubClient` class:** Creating objects of the interface type is not allowed. The method `getFeePercentage()` is not declared in the interface, thus we cannot use a interface reference to call this method. On the other hand, using an interface reference to refer to an object of type `Worker` that implements the interface is allowed.

9. Find the compile-time errors in the following code. (Try without a compiler first.)

```

// (1) An abstract class
abstract class ClubMember2 {
 double feePercentage = 0.012;
 double calculateFee(double sum) {
 feePercentage += 0.002;
 return feePercentage*sum;
 }
}

// (2) Extending an abstract class
class Worker2 extends ClubMember2 {

```



```

 double calculateFee(double annualSalary) {
 feePercentage += 0.001;
 return feePercentage * annualSalary;
 }
 double getFeePercentage() { return feePercentage; }
 }
 // (3) Client using superclass and subclass references
 public class AbstractClassClient {
 public static void main(String[] args) {
 ClubMember2 member = new ClubMember2();
 double percentage = member.feePercentage;
 Worker2 worker = new Worker2();
 System.out.printf("Fee: %.2f GBP%n", worker.calculateFee(35000.0));
 System.out.printf("Percentage: %d%n", worker.getFeePercentage());
 }
 }
}

```

In the abstract class `ClubMember2`, the variable `feePercentage` is not final, so its value can be changed. Implementing methods in an abstract class is allowed. Thus, there are no compile-time errors in this class.

In the `Worker2` class there are no compile-time errors either. The class overrides an inherited method, `calculateFee()`. The inherited variable `feePercentage` is not final, and is visible in the superclass and therefore inherited, so we can freely change its value. Declaring new methods in a subclass is not allowed (in this case `getFeePercentage()`).

In the `AbstractClassClient` class, the attempt to create an object of the abstract class `ClubMember2` is not allowed. In the last but one line we call a overridden method, and in the last line we call a new method declared by the subclass.

## 13.6 Programming exercises

1. Implement new versions of the classes for employees, hourly workers, managers and piece workers, that all override the `toString()` and `equals()` methods from the `Object` class. Base your new classes on the `EmployeeV0` class from Chapter 12, the `ManagerV2` class from Program 12.9, and the `HourlyWorkerV0` and `PieceWorkerV0` classes from Program 13.1 and Program 13.2, respectively.

Write a client, `SalaryManager2`, based on Program 13.3, that creates two employees, and then calculates and prints their weekly salary. Finally, it compares the two employees for equality.

Example of running the program, when two employees of different types are created:

```

Types of employees:
1 - Employee
2 - Hourly worker
3 - Manager
4 - Piece worker

```



Chose a category: 2  
 Employee information:  
 First name: Mary Last name: Smith Hourly rate: 35.00 GBP  
 Weekly salary: 1487.50 GBP

Types of employees:

- 1 - Employee
- 2 - Hourly worker
- 3 - Manager
- 4 - Piece worker

Chose a category: 3

Employee information:  
 First name: John D. Last name: Boss Hourly rate: 60.00 GBP  
 Manager bonus: 105.00 GBP  
 Weekly salary: 2955.00 GBP  
 Employees are different!

Example of running the program, when two employees with the same state are created:

Types of employees:

- 1 - Employee
- 2 - Hourly worker
- 3 - Manager
- 4 - Piece worker

Chose a category: 3

Employee information:  
 First name: John D. Last name: Boss Hourly rate: 60.00 GBP  
 Manager bonus: 105.00 GBP  
 Weekly salary: 2955.00 GBP

Types of employees:

- 1 - Employee
- 2 - Hourly worker
- 3 - Manager
- 4 - Piece worker

Chose a category: 3

Employee information:  
 First name: John D. Last name: Boss Hourly rate: 60.00 GBP  
 Manager bonus: 105.00 GBP  
 Weekly salary: 2955.00 GBP  
 Employees are equal!

2. Write a client, `SalaryClient`, that calculates and prints the weekly salaries for each employee stored in an array. It then calculates and prints the total salary costs for all the employees (see Problem 1 in the code below). The client should also determine how many managers there are in the array, and calculate the total amount paid out as manager bonuses (see Problem 2). Use the classes written in Exercise 13.1 for the different types of employees.

```
/*
 * Calculating employee salaries and simple statistics.
 */
public class SalaryClient {
```



```

public static void main(String[] args) {
 // Creates an array of different types of employees
 Employee[] empArray = {
 new Employee("John", "Doe", 30.0),
 new Manager("Kim", "Brown", 55.0, 235.0),
 new HourlyWorker("Mary", "Smith", 35.0),
 new Manager("John D.", "Boss", 60.0, 105.0),
 new PieceWorker("Ken", "Jones", 12.5, 75)
 };

 // Problem 1:
 // Calculates and prints the weekly salary for each employee.
 // Calculates and prints the total amount paid out in salaries.
 double empSalary;
 double sumSalaries = ____;
 // Implementation of Problem 1
 // ...

 // Problem 2:
 // Calculates the total number of managers and the total amount paid
 // out as manager bonuses.
 int numManagers = ____;
 double sumBonuses = ____;
 // Implementation of Problem 2
 // ...
}
}

```

Example of output from the program:

Overview of weekly salary:

The employee John Doe has a weekly salary of 1125.0 GBP  
 The employee Kim Brown has a weekly salary of 2297.5 GBP  
 The employee Mary Smith has a weekly salary of 1312.5 GBP  
 The employee John D. Boss has a weekly salary of 2355.0 GBP  
 The employee Ken Jones has a weekly salary of 937.5 GBP

Salary statistics:

Total salary costs for 5 employees is 8027.5 GBP  
 Total bonuses for 2 managers is 340.0 GBP

- 3.** Use the `SalaryClient` class from Exercise 13.2 as basis for a more flexible client, where the number of employees and their data is read from the keyboard.

Example of running the program:

- Number of employees: **5**

Type information about employee no.1:

Supported employee types are:

1 - Employee. 2 - Hourly worker. 3 - Manager. 4 - Piece worker

- Employee type: **1**

- First name: **John**

- Last name: **Doe**



```

- Hourly rate: 30.0
...
Type information about employee no.5:
Supported employee types are:
1 - Employee. 2 - Hourly worker. 3 - Manager. 4 - Piece worker
- Employee type: 4
- First name: Ken
- Last name: Jones
- Payment per unit: 12.5
- Number of units produced: 75
Overview of weekly salary:
The employee John Doe has a weekly salary of 1125.0 GBP
...
The employee Ken Jones has a weekly salary of 937.5 GBP
Salary statistics:
Total salary costs for 5 employees is 8027.5 GBP
Total bonuses for 2 managers is 340.0 GBP

```

To allow the user to type employee data at the keyboard, write a `TextUserInterface` class that reads data for different types of employees. This class is used by the client `SalaryClient2` given below. The `requestInteger()` method reads an integer from the keyboard that represents the number of employees to be created and stored in the array (Problem 1), while the `inputEmployee()` method reads data for an employee from the keyboard as shown above, and returns the reference value of an appropriate employee object (Problem 2). For example, if the user chooses the employee type 3, the `inputEmployee()` method returns the reference value of a new `Manager` object with the relevant data about this manager read from the keyboard.

The `TextUserInterface` class in Program 8.3 on page 213 can be used as basis for this exercise. Use the employee classes written in Exercise 13.1.

```

/**
 * Calculating employee salaries and other statistics.
 * Employee data is read from the keyboard.
 */
public class SalaryClient2 {
 public static void main(String[] args) {
 // Creates a text user interface for input of data about employees.
 TextUserInterface tui = new TextUserInterface();

 // Problem 1:
 // Creates an array that can hold different types of employees
 int numEmps = tui.requestInteger("Number of employees");
 Employee[] empArray = _____;

 // Problem 2:
 // Inserts employees into the employee array,
 // reading the data for each employee from the keyboard.
 for (int i=0; i<empArray.length; i++){
 empArray[i] = tui.inputEmployee("employee no. " + (i+1));
 }
 }
}

```



```

 // Calculates the weekly salary for each employee.
 // Calculates and prints the total amount paid out in salaries.
 // Use the same implementation as in the SalaryClient class.
 // ...

 // Calculates the total number of managers and the total amount paid
 // out as manager bonuses.
 // Use the same implementation as in the SalaryClient class.
 // ...
}

}

```

- 4.** Complete the declaration of the `LottoRow` class given below that represents a lotto row with numbers from 1 to 34, inclusive, where a number can only occur *once* in the row. The `nextInt()` method in the `Random` class returns a random number between 0 (inclusive) and  $n$  (exclusive), where  $n$  is given as a parameter in the method call.

```

import java.util.Random;
/**
 * Represents a row in a lotto draw, consisting of MAX_COUNT numbers.
 */
class LottoRow {
 // Constants
 _____ int SMALLEST_NUMBER = 1;
 _____ int LARGEST_NUMBER = 34;
 _____ int MAX_COUNT = 7;

 // Field variables
 boolean[] drawn; // true means number equal to index+1 has been drawn
 int[] lottoNumbers; // array (row) holding the numbers drawn
 int numDrawn; // how many numbers have been drawn so far

 // Constructors
 public LottoRow() { // Problem 1
 // Creates and initialises a lotto row. No numbers are drawn yet,
 // so the row does not contain any numbers.
 }

 public LottoRow(int[] row) { // Problem 2
 // Creates and initialises a lotto row, copying numbers from an array
 // of integers passed as parameter.
 }

 // Instance methods
 public void makeALottoDraw() { // Problem 3
 // Draw a random lotto row containing MAX_COUNT unique integers.
 // Tip: Use the array drawn to ensure the row does not contain
 // duplicates.
 }
}

```



```

public int getBall(int i) { // Problem 4
 // Gets the i-th "ball" of the current lotto row, i.e. the i-th number.
 // Return 0 if the given index is invalid.
}

// Overrides the method from the Object class
public boolean equals(Object obj) { // Problem 5
 // Determines whether two lotto rows are equal.
 // Return true if the two lotto rows have the same number in
 // corresponding positions; otherwise false.
}

// Overrides the method from the Object class
public String toString() { // Problem 6:
 // Returns a string representing the lotto row.
}
}
}

```

- 5.** Write a client, `LottoGame`, that uses the `LottoRow` class from Exercise 13.4 to do the following:
- Draw a lotto row.
  - Allow the user to input his or her own lotto row.
  - Compare the two lotto rows to determine whether the user has won.

Tip: The `Scanner` class can be used to read lotto numbers from the user. For part (b), use a constructor to create a lotto row based on an array of integers read from the keyboard. To simplify the comparison, you can assume that the numbers in the user's lotto row are read in the same sequence as the numbers in the lotto row drawn by the program.

Example of running the program where the user's lotto row is identical to the row generated by the program:

```

Type your 7 lotto numbers:
Number 1: 27
Number 2: 25
Number 3: 10
Number 4: 21
Number 5: 18
Number 6: 6
Number 7: 2

```

Congratulations! You won.

Example of running the program where the user's lotto row is not identical to the row generated by the program:

```

Type your 7 lotto numbers:
Number 1: 11
Number 2: 3
Number 3: 10

```



Number 4: 19

Number 5: 9

Number 6: 21

Number 7: 31

Sorry, you didn't win today...

The lotto draw was: 27, 25, 10, 21, 18, 6, 2

6. Use the game of Nim from Exercise 12.5 on page 357 as a starting point for this exercise. Define the behaviour of a player as a contract using pre- and postconditions, and decide whether the solution should be based on an interface or an abstract superclass. Implement the same types of players as in Exercise 12.5 based on your decision. Use assertions to verify that the conditions specified in the contract are satisfied at runtime.
7. Based on Exercise 6.8 on page 158, implement a solution for the game of *Craps* based on contracts defined by pre- and postconditions. Use assertions to verify that the conditions specified in the contract are satisfied at runtime.
8. Blackjack is a well-known card game where several players can play against a dealer. Each player gets two cards from the dealer to begin with. Both cards are placed face up on the table, for all players and the dealer to see. The dealer also gets two cards, but only one of them is placed face up. The other is placed face down. If players decide to get more cards later on, these will also be placed face up on the table.

The hand with highest value wins, as long as the total value of the cards does not exceed 21. The cards have values as follows: cards 2 through 10, regardless of the suit, are worth their face value, i.e. 2 of Spades has the value 2, 3 of Hearts has the value 3, and so on. For the jack, queen and king cards, each has the value 10, regardless of the suit. An ace has the value 1 or 11, depending on which value will bring the total value of the hand closer to 21, without exceeding this limit.

A hand with a total of 21 is called a “blackjack”, and is a winning hand. A hand with a total above 21 is called a “bust”, and is a losing hand. Each player plays individually against the dealer. The highest hand wins, unless it is a bust. If two hands have the same value, we have a “push” (or tie) and neither the player nor the dealer wins.

Before the first two cards are dealt, each player makes a bet, which is placed on the table. Each player then inspects his cards, as well as the face-up card of the dealer, and decides to:

- Hit - take another card.
- Stand (or stay) - take no more cards.
- Double down - double his bet, take *one* more card, and then stand.

A player is allowed to hit several times, until he has reached a total value he thinks will win the game (with the dealer) or the total exceeds 21. In the latter case, he has a bust and loses his bet to the dealer. When all players have finished taking cards, the dealer turns his face-down card, and plays his hand. The rule is he must continue to hit until the value of his hand is at least 17. When the dealer has finished taking cards, the total value of his hand is compared with the total value of each player's hand. If a player's hand is better than the dealer's, the player wins back his bet. If the



dealer has a bust, all players that don't have a bust themselves, win back their bet. If a player has a blackjack, he wins back 1.5 times his bet, unless the dealer also has a blackjack, making the game a tie. In tie games, the players get their bet back.

Implement a program that allows three users to play a game of Blackjack against a dealer. Define the behaviour of a player and a dealer as interfaces using contracts with pre- and postconditions. Use assertions to verify that the conditions specified in the contract are satisfied at runtime. Look at the contracts defined for the game of Nim in Exercise 13.6 on page 391 for classes that are needed to run a game.

Implement a basic player, optimistically hitting (i.e. taking more cards) until the total of his hand is 21 (a “blackjack”) or a bust occurs, a real player that determines the next operation from user input, and a computer player that hits until the total of his hand is at least 17. A dealer class must be implemented as well, also hitting until the value of his hand is at least 17. After that, the dealer should only continue hitting if one or more players have blackjack, and their win will exceed the bets of the other players that stand to loose.

The program will also need classes for a shoe, a device that deals random cards from a set of  $n$  deck of cards, typically four. Each deck has 52 cards, rank 2 to 10, Jack, Queen, King and Ace of suits Clubs, Diamonds, Hearts and Spades. The shoe must keep track of all cards dealt so far, and be able to collect and shuffle all cards to start the game all over again.



## P A R T   F O U R

### Applying OOP



## Test-driven program development

### LEARNING OBJECTIVES

By the end of this chapter, you will understand the following:

- How to develop large programs using test-driven programming techniques.
- How to write tests and run them using the JUnit framework.
- How to encapsulate and hide implementation details in classes and packages.
- How to create and use packages in order to write class libraries.

### INTRODUCTION

This chapter demonstrates techniques that can be used to develop large programs. These techniques help us analyse and understand a problem, implement and test a solution, and at the same time, avoid the source code becoming overtly complex and unstable during the development process.

We emphasise techniques for test-driven program development using the JUnit testing framework. As a running example, we undertake a case study: implementing the game four-in-a-row. In Section 20.9 we extend the game with a graphical user interface (GUI) after we have covered GUI building.

### 14.1 Developing large programs

The aim is to develop a program that can be used to play the strategy game *Four-in-a-row*. This game is played by two players, and the winner is the player who manages to place four of her pieces consecutively on a game board that comprises seven columns and six rows.



The game is played with pieces that come in two different colours, and each player has 21 pieces of the same colour. The game board is vertical, so that a player can drop a piece into a column. The piece will land in the bottommost vacant row.

**FIGURE 14.1** The strategy game four-in-a-row

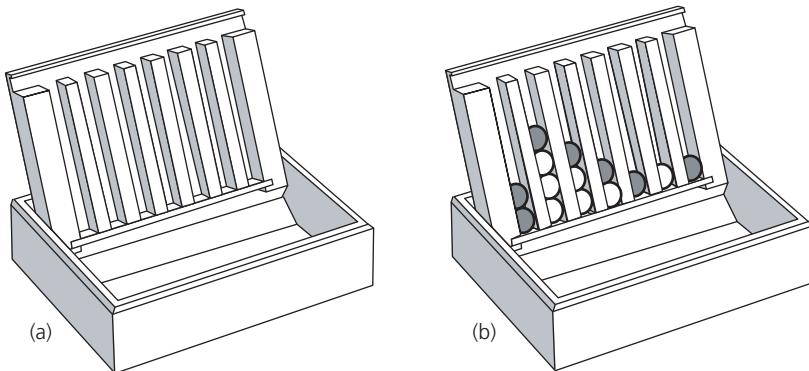


Figure 14.1a shows the game board when the game starts. The players take turns in dropping one of their pieces in a column that is not full. A player wins and ends the game when she drops a piece such that four of her pieces become consecutively placed, either vertically, horizontally or diagonally. Figure 14.1b shows an outcome where the player with the black pieces has won. The outcome is a tie if the board becomes full before any player has won.

As we progress with developing the program for the game, we will be forced to take a stand on how the interaction with the players takes place:

- How should the game board be represented?
- How should a player indicate which column to drop a piece in?
- Can the program take on the role of a player?

But first we focus on the most fundamental operations in the game. When we are trying to develop a solution, it is often useful to start with the core of the problem as early as possible.

## 14.2 Simple testing with assert

One fundamental operation is placing a piece on the game board when it is dropped in a column. In order to get started, let us look at the following concrete problem:

*A piece is dropped down the first column in an empty game board. Where does the piece land?*

Based on the description of the game, we expect the piece to land in the bottommost row of the first column. We can easily write code that we *assume* implements this behaviour, but we wish to *verify* that the code actually does what we expect. For small, simple programs, we can just run the program and manually verify that the behaviour is correct.



For large programs, it is not practical to perform an exhaustive manual testing each time a change is made to the program.

We have earlier seen how we can use `assert` statements to verify program behaviour. Let us write a test with an `assert` statement that verifies the behaviour of dropping a piece in the first column of an empty game board. Employing *automated testing* is highly recommended.

## The game board

We start with a test that verifies that the location given by the bottommost row and the leftmost column on a new game board is empty.

### PROGRAM 14.1 Testing an empty game board

```
// File: TestGameBoard.java
public class TestGameBoard {
 public static void main(String[] args) {
 testEmptyCell();
 }

 public static void testEmptyCell() {
 GameBoard board = new GameBoard();
 // Bottom row has index 0.
 int row = 0;
 // The leftmost column also has index 0
 int column = 0;

 // Period ('.') denotes an empty cell.
 assert board.pieceAt(column, row) == '.';
 }
}
```

Compilation errors:

```
TestGameBoard.java:8: cannot find symbol
symbol : class GameBoard
location: class TestGameBoard
 GameBoard board = new GameBoard();
 ^
TestGameBoard.java:8: cannot find symbol
symbol : class GameBoard
location: class TestGameBoard
 GameBoard board = new GameBoard();
 ^
```

2 errors

---

The test in Program 14.1 may seem trivial, but it has helped us understand a great deal:



- We need a `GameBoard` class.
- The `GameBoard` class needs a method `pieceAt()` that can be used to examine the state of each cell on the game board.
- Columns and rows are referred to via indices.
- The columns from left to right have indices 0, 1, 2, 3, 4, 5 and 6.
- The rows from top to bottom have indices 5, 4, 3, 2, 1 and 0.
- The state of a cell can be indicated by a value of the type `char`.
- The period sign ('.') denotes an empty cell. Other characters (for example, 'X' and 'O') can be used as game pieces of different colours.

These decisions are taken because this was the easiest way to formulate the first test for our problem. It is very possible that as we make progress with the problem, these choices may need to be re-evaluated.

From Program 14.1 we see that when we try to compile the program, the compiler reports 2 errors. This was expected, as we do not have a `GameBoard` class. Let us try to satisfy the compiler by writing a minimal implementation of the `GameBoard` class (Program 14.2).

### PROGRAM 14.2 Minimal `GameBoard` implementation

```
// File: GameBoard.java
public class GameBoard {}
```

Compilation errors:

```
TestGameBoard.java:15: cannot find symbol
symbol : method pieceAt(int,int)
location: class GameBoard
 assert board.pieceAt(column, row) == '.';
^
1 error
```

---

From Program 14.2 we see that the compiler reports that the `GameBoard` class is missing a `pieceAt()` method. The errors reported by the compiler can be a useful reminder of the work that remains to be done.

### PROGRAM 14.3 Minimal implementation of the `pieceAt()` method

```
// File: GameBoard.java
public class GameBoard {
 char pieceAt(int column, int row) {
 return '.';
 }
}
```

No compilation errors and no output when run.

```
> java -ea TestGameBoard
```



&gt;

Program 14.2 provides an implementation of the `pieceAt()` method in the `GameBoard` class. The program now compiles. No errors are reported when the program is run, either. We use the `-enableassertions` option to turn on the execution of assertions at runtime.

The method `pieceAt()` may seem naive, since it always returns the character '.', but it satisfies the test from Program 14.1. However, the game board ought to maintain the state of all the 42 cells on the board. We obviously need more tests. Gradually these tests will demand that we write a smarter `pieceAt()` method. Let us continue by writing tests for dropping pieces, and see if it forces us to write a smarter version of the method. Without testing it will not be possible to verify that the functionality of the smarter version of the method is correct.

## Game board state

Now that we know that the `testEmptyCell()` method from Program 14.1 works, we can continue writing additional tests in the `TestGameBoard` program. We modify the `main()` method from Program 14.1 and write a new test method, `testDropPiece()`, shown in Program 14.4, to verify that the program behaves as expected when dropping a piece in the first column of the game board.

### PROGRAM 14.4 Adding a new test method `testDropPiece()`

```
// File: TestGameBoard.java
public class TestGameBoard {
 public static void main(String[] args) {
 testEmptyCell();
 testDropPiece();
 }

 // ... other members are the same ...

 public static void testDropPiece() {
 GameBoard board = new GameBoard();
 int column = 0;
 board.dropPieceDownColumn('X', column);
 // The piece should now reside at the bottom row of the leftmost column:
 assert board.pieceAt(0, 0) == 'X';
 }
}
```

The compiler reports the following error:

```
TestGameBoard.java:22: cannot find symbol
symbol : method dropPieceDownColumn(char,int)
location: class GameBoard
 board.dropPieceDownColumn('X', column);
 ^

```



1 error

---

As we can see from Program 14.4, the method `testDropPiece()` places new demands on the `GameBoard` class, as this class does not have a method called `dropPieceDownColumn`. Again the compiler has helped us to uncover requirements that are not yet fulfilled. We write a naive implementation of the missing method, as shown in Program 14.5. Now the code compiles, but we get a runtime error in the `TestGameBoard` class.

#### PROGRAM 14.5 A naive version of the `dropPieceDownColumn()` method

```
// File: GameBoard.java
public class GameBoard {

 // ... other members are the same ...

 void dropPieceDownColumn(char piece, int column) {}
}
```

Compiles without errors, but throws an exception at runtime:

```
>java -ea TestGameBoard
Exception in thread "main" java.lang.AssertionError
 at TestGameBoard.testDropPiece(TestGameBoard.java:24)
 at TestGameBoard.main(TestGameBoard.java:5)
```

---

In Program 14.5, it is the `assert` expression (`board.pieceAt(0, 0) == 'X'`) that fails, because the `pieceAt()` method at the moment always returns the character `'.'`. We see the need to improve the `pieceAt()` method. The method `testDropPiece()` shows quite clearly that there is a connection between the piece that is passed as argument to the `dropPieceDownColumn()` method and the piece that is returned by the `pieceAt()` method. Let us rewrite the methods `dropPieceDownColumn()` and `pieceAt()` to take this into consideration.

With the new `GameBoard` class in Program 14.6, both the tests in the `TestGameBoard` class compile and run without errors. The class stores the state, but only for one cell, and ignores the column and row parameters in the method calls. This implementation is clearly not satisfactory for playing a full-fledged game of four-in-a-row.

#### PROGRAM 14.6 Game board with the state of only one cell

```
// File: GameBoard.java
public class GameBoard {
 char cellContents = '.';

 char pieceAt(int column, int row) {
 return cellContents;
 }
}
```



```
void dropPieceDownColumn(char piece, int column) {
 cellContents = piece;
}
}
```

---

## The state of several cells

We can emphasise the need for maintaining the state of several cells on the game board by adding several assertions in the `testDropPiece()` method of the `TestGameBoard` class, as shown in Program 14.7.

### PROGRAM 14.7 A naive version of the `testDropPiece()` method

```
// File: TestGameBoard.java
public class TestGameBoard {

 // ... other members are the same ...

 public static void testDropPiece() {
 GameBoard board = new GameBoard();
 board.dropPieceDownColumn('X', 0);
 board.dropPieceDownColumn('O', 2);
 /* The bottom rows should now look like this:

 1 | |
 0 | X . O . . . |

 / 0 1 2 3 4 5 6 \
 */
 assert board.pieceAt(0, 0) == 'X';
 assert board.pieceAt(1, 0) == '.';
 assert board.pieceAt(2, 0) == 'O';
 assert board.pieceAt(0, 1) == '.';
 }
}
```

Compiles without errors, but throws an exception at runtime:

```
>java -ea TestGameBoard
Exception in thread "main" java.lang.AssertionError
 at TestGameBoard.testDropPiece(TestGameBoard.java:30)
 at TestGameBoard.main(TestGameBoard.java:5)
```

---

The class in Program 14.7 compiles, but we get a runtime error in the `TestGameBoard` class. The assertion that fails is on line 30 of the `TestGameBoard.java` file (`assert(board.pieceAt(0, 0) == 'X')`). This assertion fails because the `pieceAt()` method returns the character '`'O'`', as this is the value that is stored in the `cellContents` field by the



previous call to the `dropPieceDownColumn()` method. Let us try to rewrite the `GameBoard` class to satisfy the requirements of the `testDropPiece()` method, by changing the declaration of the field `cellContents` to a two-dimensional array, as shown in Program 14.8.

### PROGRAM 14.8 Game board with two-dimensional array

```
// File: GameBoard.java
public class GameBoard {
 char[][] cellContents = new char[7][6];

 char pieceAt(int column, int row) {
 return cellContents[column][row];
 }

 void dropPieceDownColumn(char piece, int column) {
 int bottommostVacantRow = 0;
 cellContents[column][bottommostVacantRow] = piece;
 }
}
```

Compiles without errors, but throws an exception at runtime:

```
>java -ea TestGameBoard
Exception in thread "main" java.lang.AssertionError
 at TestGameBoard.testEmptyCell(TestGameBoard.java:16)
 at TestGameBoard.main(TestGameBoard.java:4)
```

In Program 14.8, the code compiles, but we get an assertion error in the `testEmptyCell()` of the `TestGameBoard.java` file. The source code of this method from Program 14.1, shows line 16 to be the following:

```
assert board.pieceAt(column, row) == '.';
```

We have made the assumption that a period sign indicates an empty cell, but the elements in the two-dimensional array `cellContents` are all assigned the default value '`\u0000`' for the type `char`. We have experienced a *regression*, where a test that earlier worked, does not work any more. There two obvious strategies for fixing this error:

- 1** Assign the period sign to all cells in the game board in the constructor of the `GameBoard` class.
- 2** Change the assertion and instead use the character '`\u0000`' to indicate an empty cell.

We have the possibility to change the assertion here, since the use of the period sign for marking empty cells was not a requirement in the original problem, but a convention we introduced in order to write some concrete tests. We choose to change the assertion, since this will result in smaller code, and because there is nothing special with the period sign that makes it a better choice than the character '`\u0000`'.

We now change all occurrences of the period sign in the class, and use the opportunity to introduce a constant that represents an empty cell. That way, the text code does not need



to know which value represents an empty cell. The new code is shown in Program 14.9. It both compiles and runs without errors. For the moment it assumes that the bottom-most row is always empty. This is something we ought to look into by writing test methods to check the stacking of the pieces as they are dropped into the columns.

### PROGRAM 14.9 Functional two-dimensional game board with tests

```
// File: GameBoard.java
public class GameBoard {
 public static final char EMPTY_CELL = '\u0000';

 char[][] cellContents = new char[7][6];

 GameBoard() {
 assert cellContents[0][0] == EMPTY_CELL;
 }

 char pieceAt(int column, int row) {
 return cellContents[column][row];
 }

 void dropPieceDownColumn(char piece, int column) {
 int bottommostVacantRow = 0;
 cellContents[column][bottommostVacantRow] = piece;
 }
}

// File: TestGameBoard.java
public class TestGameBoard {
 public static void main(String[] args) {
 testEmptyCell();
 testDropPiece();
 }

 public static void testEmptyCell() {
 GameBoard board = new GameBoard();
 // Bottom row has index 0.
 int row = 0;
 // The leftmost column also has index 0
 int column = 0;
 assert board.pieceAt(column, row) == GameBoard.EMPTY_CELL;
 }

 public static void testDropPiece() {
 GameBoard board = new GameBoard();
 board.dropPieceDownColumn('X', 0);
 board.dropPieceDownColumn('O', 2);
 /* The bottom rows should now look like this:

```



```

1 | |
0 | X . O . . . |
-----|
 / 0 1 2 3 4 5 6 \
 */
assert board.pieceAt(0, 0) == 'X';
assert board.pieceAt(1, 0) == GameBoard.EMPTY_CELL;
assert board.pieceAt(2, 0) == 'O';
assert board.pieceAt(0, 1) == GameBoard.EMPTY_CELL;
}
}

```

---

## Stacking pieces

The next test checks that the pieces are stacked as expected, when they are dropped into the same column. This test is implemented by the `testStacking()` method in Program 14.10.

The code compiles, but throws an assertion error in the `testStacking()` method at runtime. The stack trace tells us that the assertion error occurred in line 48 of the file `TestGameBoard.java`:

```
assert(board.pieceAt(3, 0) == 'X');
```

This tells us that we expected the bottom piece in column 3 to be 'X', but unfortunately it does not tell us which piece the call to the `pieceAt()` method actually returned. The returned value could be either `EMPTY_CELL` or 'O'. We would be better armed for dealing with the error if the error message had given us more information.

### PROGRAM 14.10 Checking stacking of pieces in a column

```

// File: TestGameBoard.java
public class TestGameBoard {
 public static void main(String[] args) {
 testEmptyCell();
 testDropPiece();
 testStacking();
 }

 // ... other members are the same ...

 public static void testStacking() {
 GameBoard board = new GameBoard();
 board.dropPieceDownColumn('X', 3);
 board.dropPieceDownColumn('O', 3);

 /* The bottom rows should now look like this:

```

```
2 | |
```



```

1 | . . . 0 . . . |
0 | . . . X . . . |

/ 0 1 2 3 4 5 6 \
*/
assert board.pieceAt(3, 0) == 'X';
assert board.pieceAt(3, 1) == '0';
assert board.pieceAt(3, 2) == GameBoard.EMPTY_CELL;
}
}

```

Compiles without errors, but throws an exception at runtime:

```

Exception in thread "main" java.lang.AssertionError
at TestGameBoard.testStacking(TestGameBoard.java:48)
at TestGameBoard.main(TestGameBoard.java:6)

```

## Better error messages when tests fail

The development process has so far consisted of two alternating phases:

- 1** Write an assert test that deals with a concrete aspect of the problem and reveals a deficiency in the current implementation.
- 2** Improve the implementation, so that the test no longer fails.

Each time we claim to have completed one of the phases, we run the program to see if the changes have had the desired effect. When we write tests, we wish to shed light on deficiencies by making the program fail. When we improve the implementation, we want to make the program run without errors. The tests make it possible for us to know exactly what functionality the code implements.

A successful execution of the tests does not result in any output, but when the program fails, we find out which assert expression failed. The error message is a little difficult to interpret, because it only informs about which line the error occurred in, but not its cause.

In Program 14.11, in order to improve error reporting, we have written a separate method (`assertEquals()`) to execute assertions that involve comparisons. The error message we now get is more informative, as we can see from the output in Program 14.11. It looks like the piece we dropped last ('0') has landed in the bottom row, where we expected to find the piece we dropped first ('X'). The reason is that the `dropPieceDownColumn()` method in the `GameBoard` class always sets the local variable `bottommostVacantRow` to 0, without first checking whether the row 0 is in fact vacant. We will correct this error after we have taken a closer look at using *helper methods* for testing.

### PROGRAM 14.11 Better error messages when testing

```

// File: TestGameBoard.java
public class TestGameBoard {

 // ... other members are the same ...

```



```

static void assertEquals(String message, char expected, char actual) {
 if (expected != actual) {
 System.err.printf("Not equal: %s: Expected <%s>, but got <%s>\n",
 message, expected, actual);
 }
 assert expected == actual;
}

public static void testStacking() {
 GameBoard board = new GameBoard();
 board.dropPieceDownColumn('X', 3);
 board.dropPieceDownColumn('O', 3);

 /* The bottom rows should now look like this:

 2 | |
 1 | . . . 0 . . . |
 0 | . . . X . . . |

 / 0 1 2 3 4 5 6 \

 */
 assertEquals("First dropped", 'X', board.pieceAt(3, 0));
 assertEquals("Second dropped", 'O', board.pieceAt(3, 1));
 assertEquals("Only two stacked", GameBoard.EMPTY_CELL, board.pieceAt(3, 2));
}
}

```

Compiles without errors, but throws an exception at runtime:

```

Not equal: First dropped: Expected <X>, but got <O>
Exception in thread "main" java.lang.AssertionError
 at TestGameBoard.assertEquals(TestGameBoard.java:40)
 at TestGameBoard.testStacking(TestGameBoard.java:56)
 at TestGameBoard.main(TestGameBoard.java:6)

```

---

## 14.3 Testing framework

The helper method `assertEquals()` that we wrote in Program 14.11 to improve the feedback when stacking of the pieces failed, can also be used in all the other tests we have written so far. As we write more tests, chances are that we will need several helper methods, for example, methods to test inequality, and to test values of other types than just chars.



## JUnit framework

JUnit is one of the most popular testing frameworks for Java, that provides methods for testing of programs. In this book we will be using JUnit version 4.4. It can be downloaded from the following URL, which is also provided on the book web site: <http://www.junit.org/>. The distribution has documentation that describes how to install and use JUnit. Installation consists of the following steps:

- 1 From the JUnit home page, navigate to the download page, and download the *archive file* `junit4.4.zip`.
- 2 Unzip the archive and copy the resulting directory `junit4.4` to where you want to install JUnit.
- 3 Include the `parentPath/junit4.4/junit-4.4.jar` entry in the environment variable `CLASSPATH`, where `parentPath` is the path of the parent directory where you copied the directory `junit4.4`.

Check that JUnit is installed properly:

```
> java org.junit.runner.JUnitCore
JUnit version 4.4
```

Time: 0

```
OK (0 tests)
>
```

If you get an error message that mentions the `NoClassDefFoundError`, check that your `CLASSPATH` environment variable is set correctly to indicate the `junit-4.4.jar` file. Read the installation documentation for JUnit for more information.

## Writing test methods

We now turn our attention to writing tests using JUnit. Program 14.12 illustrates the approach by modifying the declaration of the `TestGameBoard` class so that its test methods can be run with JUnit. The following steps can be used to write a *test class*, i.e. a class that has test methods:

- Declare the class header of the test class as any other class, as shown at (3). A common convention is to use the prefix "Test" for the class name so that readers can identify the class as a test case.
- Specify each test method with the annotation `@Test`, as shown for the test method `testEmptyCell()` at (4). An *annotation* is a special type in Java, and the annotation `@Test` tells JUnit that this method is to be executed as a test method. If a method in the class does not have this annotation, the method is not run by JUnit. This annotation is declared in the `org.junit` package of JUnit, and we need to include an `import` statement to that effect, as shown at (1). See *Packages and the import statement* on page 437. A test method is an instance method. It can have any name, but it must be `public`, `void` and without parameters. A common convention is to use the prefix "test" for the method name so that readers can identify the method as a test method.
- JUnit provides helper methods that are useful for writing test methods. A call to one such helper method, `assertEquals()`, is shown at (5). These helper methods in



JUnit are defined in a class called `org.junit.Assert`, and we need to include an `import static` statement to that effect, as shown at (2). See *Static import* on page 437. Table 14.1 shows a selection of the most common helper methods in this class. All the methods are `static` and `void`, and they all throw an exception of the type `AssertionError` if the test fails. This exception is handled by JUnit.

### PROGRAM 14.12 Writing tests using JUnit

```
// File: TestGameBoard.java

// (1) Import the annotation type org.junit.Test:
import org.junit.Test;

// (2) Import static method assertEquals from the class org.junit.Assert:
import static org.junit.Assert.assertEquals;

/** (3) Class with test methods. */
public class TestGameBoard {

 /** (4)
 * Any method prefixed by the annotation @Test is considered a test method,
 * and will be run by JUnit.
 * A test method is public, void, and has no parameters.
 */
 @Test
 public void testEmptyCell() {
 GameBoard board = new GameBoard();
 // Bottom row has index 0.
 int row = 0;
 // The leftmost column also has index 0
 int column = 0;

 // (5) Method that compares values:
 assertEquals("Empty", GameBoard.EMPTY_CELL, board.pieceAt(column, row));
 }
 // More declarations ...
}
```



**TABLE 14.1** Selected methods from the **org.junit.Assert** class

| <b>org.junit.Assert</b>                                                                                                 |                                                                                                                                                                                                                                                              |
|-------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>static void assertEquals(<br/>    datatype expected,<br/>    datatype actual<br/>)</code>                         | Asserts that two values of the type <i>datatype</i> are equal.<br>Overloaded methods are defined that can compare values of primitive types and also objects. In the second method, if the assertion fails, the message will be included in the test report. |
| <code>static void assertEquals(<br/>    String message,<br/>    datatype expected,<br/>    datatype actual<br/>)</code> |                                                                                                                                                                                                                                                              |
| <code>static void assertTrue(<br/>    String message,<br/>    boolean condition<br/>)</code>                            | Asserts that the given boolean expression condition is <code>true</code> . If the assertion fails, the <code>message</code> will be included in the test report.                                                                                             |
| <code>static void assertFalse(<br/>    String message,<br/>    boolean condition<br/>)</code>                           | Asserts that the given boolean expression condition is <code>false</code> . If the assertion fails, the <code>message</code> will be included in the test report.                                                                                            |
| <code>static void fail(String message)</code>                                                                           | Fails the test, and the <code>message</code> is included in the test report.                                                                                                                                                                                 |

Program 14.13 shows the class `TestGameBoard` where all the test methods have been modified so that they can be run by JUnit.

**PROGRAM 14.13** Writing tests for the game board

```
// File: TestGameBoard.java
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class TestGameBoard {

 @Test
 public void testEmptyCell() {
 GameBoard board = new GameBoard();
 // Bottom row has index 0.
 int row = 0;
 // The leftmost column also has index 0
 int column = 0;
 assertEquals("Empty", GameBoard.EMPTY_CELL,
 board.pieceAt(column, row));
 }
}
```



```

@Test
public void testDropPiece() {
 GameBoard board = new GameBoard();
 board.dropPieceDownColumn('X', 0);
 board.dropPieceDownColumn('O', 2);
 /* The bottom rows should now look like this:

 1 |
 0 | X . O . . .

 / 0 1 2 3 4 5 6 \
 */
 assertEquals("Cell 0,0", 'X', board.pieceAt(0, 0));
 assertEquals("Cell 1,0", GameBoard.EMPTY_CELL, board.pieceAt(1, 0));
 assertEquals("Cell 2,0", 'O', board.pieceAt(2, 0));
 assertEquals("Cell 0,1", GameBoard.EMPTY_CELL, board.pieceAt(0, 1));
}

@Test
public void testStacking() {
 GameBoard board = new GameBoard();
 board.dropPieceDownColumn('X', 3);
 board.dropPieceDownColumn('O', 3);

 /* The bottom rows should now look like this:

 2 |
 1 | . . . 0 . .
 0 | . . . X . .

 / 0 1 2 3 4 5 6 \
 */
 assertEquals("First dropped", 'X', board.pieceAt(3, 0));
 assertEquals("Second dropped", 'O', board.pieceAt(3, 1));
 assertEquals("Only two stacked", GameBoard.EMPTY_CELL,
 board.pieceAt(3, 2));
}
}

```

Running tests in the JUnit text-based user interface:

```

> java -ea org.junit.runner.JUnitCore TestGameBoard
JUnit version 4.4
...E
Time: 0.094
There was 1 failure:
1) testStacking(TestGameBoard)
java.lang.AssertionError: First dropped expected:<88> but was:<79>
 at org.junit.Assert.failAssert(Assert.java:74)
 at org.junit.Assert.failNotEquals(Assert.java:448)
 at org.junit.Assert.assertEquals(Assert.java:102)

```



```

at org.junit.Assert.assertEquals(Assert.java:323)
at TestGameBoard.testStacking(TestGameBoard.java:50)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
...
at org.junit.runner.JUnitCore.main(JUnitCore.java:44)

FAILURES!!!
Tests run: 3, Failures: 1

```

---

## Running tests with JUnit

When we run the test class with JUnit, JUnit will create an object of the test class and automatically execute all methods that have the annotation `@Test`. From Program 14.13, we see that test methods are no longer declared static, and a `main()` method is no longer necessary.

JUnit provides a text-based user interface for running the test. Program 14.13 shows how we obtain the test results from running the `TestGameBoard` class. Note the syntax of the command to start the JUnit text-based user interface. We have also used the option `-ea` to enable execution of any `assert` statements in the code. For each test method that fails, the test report prints shows a stack trace showing where the test failed. A summary of the tests run concludes the report. All we do is write the tests and run the test class. JUnit executes, tallies and reports statistics about the test methods in the test class.

From the test report in Program 14.13 we see that JUnit has run three methods from the class `TestGameBoard`, and one of these three tests failed. The test that failed was the test method `testStacking()`, which tests the state of the cell that should contain the piece that was dropped first. The test expected the piece '`X`', but found the piece '`0`'. The following line from the stack trace:

```
at TestGameBoard.testStacking(TestGameBoard.java:50)
```

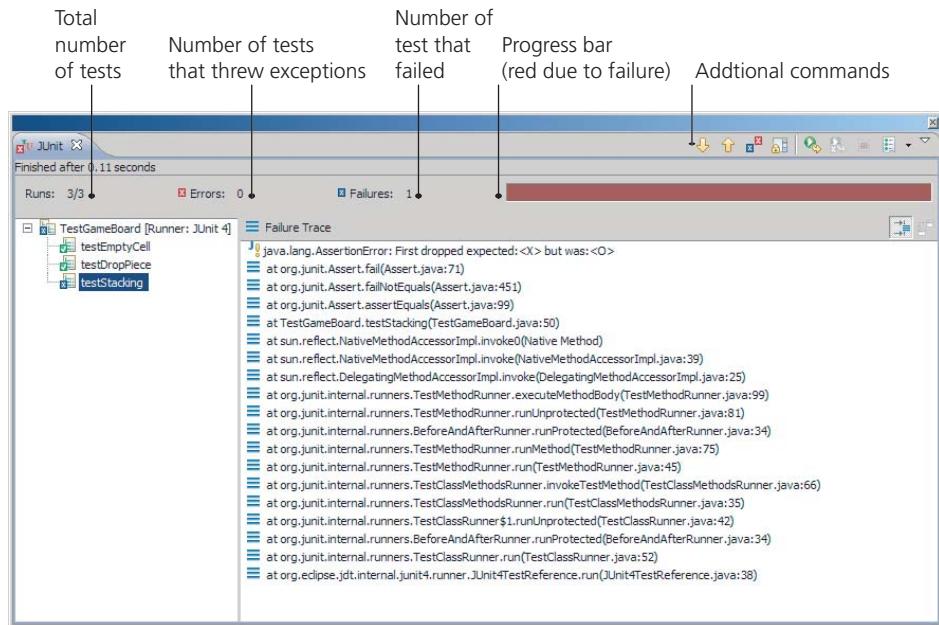
tells us that failure occurred in line 50 of the `TestGameBoard.java` file:

```
assertEquals("First dropped", 'X', board.pieceAt(3, 0));
```

In this book we will use the text-based user interface provided by JUnit. However, some *integrated development environments* (IDEs) like *Eclipse* (<http://www.eclipse.org/downloads/>) provide a graphical user interface for running tests with JUnit. An example of such a graphical user interface is shown in Figure 14.2. The graphical user interface shows essentially the same information as the text-based user interface, but also has some additional commands. The progress bar in the graphical user interface is green, but turns red if a test fails.



**FIGURE 14.2** Running tests in the JUnit graphical user interface



## Fixing test failures with JUnit

Let us now correct the error by writing a new method `findBottommostVacantRow()` in the `GameBoard` class (Program 14.14). This method finds the bottommost vacant cell in a column, instead of presuming that the cell in row 0 is vacant. After compiling the code, running the tests in the text-based user interface shows that all the tests passed (Program 14.14).

### PROGRAM 14.14 Find the bottommost vacant row

```
// File: GameBoard.java
public class GameBoard {
 public static final int NO_VACANT_ROW = -1;

 // ... other members are the same ...

 int findBottommostVacantRow(int column) {
 char[] columnContents = cellContents[column];
 for (int row = 0; row < columnContents.length; ++row) {
 if (columnContents[row] == EMPTY_CELL)
 return row;
 }
 return NO_VACANT_ROW;
 }

 void dropPieceDownColumn(char piece, int column) {
```



```
 cellContents[column][findBottommostVacantRow(column)] = piece;
}
}
```

Testing using the JUnit text-based user interface:

```
> java -ea org.junit.runner.JUnitCore TestGameBoard
JUnit version 4.4
...
Time: 0.016

OK (3 tests)
```

## Full columns

When we wrote the `findBottommostVacantRow()` method in the `GameBoard` class, we had to decide what the method should return if there were no vacant cells left in the column. We decided to return the value `NO_VACANT_ROW` (-1) to indicate that the column was full. If the `dropPieceDownColumn()` method is called with an index of a full column, the method will throw an exception of the type `IndexOutOfBoundsException`, since the value -1 is not a valid index for arrays. It is an error to call the `dropPieceDownColumn()` method when the column is full. Therefore, there ought to be a way to check whether the column is full, before this method is called.

The method `testFullColumn()` in the `TestGameBoard` class tests that a column, in this case column 3, becomes full after 6 pieces are dropped in the column. This method calls the `isColumnFull()` method in the `GameBoard` class. This method checks if the topmost cell in the column is empty. If it is not empty, the column is full. JUnit shows that the program code passes all four tests in the `TestGameBoard` class.

### PROGRAM 14.15 Testing whether a column is full

```
// File: TestGameBoard.java
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertFalse;
import org.junit.Test;

public class TestGameBoard {

 // ... other members are the same ...
 @Test
 public void testFullColumn() {
 GameBoard board = new GameBoard();
 int column = 3;
 char[] pieces = {'X', 'O', 'X', 'O', 'X', 'O'};
 for (char piece : pieces) {
 assertFalse(board.isColumnFull(column));
```



```

 board.dropPieceDownColumn(piece, column);
 }
 assertTrue(board.isColumnFull(column));
}
}

// File: GameBoard.java
public class GameBoard {

 // ... other members are the same ...

 boolean isColumnFull(int column) {
 char[] columnContents = cellContents[column];
 int indexOfTopmostCell = columnContents.length - 1;
 char topmostCell = columnContents[indexOfTopmostCell];
 return topmostCell != EMPTY_CELL;
 }
}
> java -ea org.junit.runner.JUnitCore TestGameBoard
JUnit version 4.4
...
Time: 0

OK (4 tests)

```

---

### BEST PRACTICES

When a test fails, always provide informative messages to determine the cause of the failure.

### BEST PRACTICES

Test only one aspect in each test method. If multiple aspects are tested in one method, the first aspect that fails testing will terminate the execution of the test method, and the remaining aspects will not be tested.

### BEST PRACTICES

Write tests as you analyse the problem and implement the solution.



## 14.4 Printing the game board

The `GameBoard` class has all the basic functionality needed to simulate moves where pieces are dropped down columns and become stacked on top each other in the columns. Until now, we have written tests that show that the class can simulate a game board, but have had no visual representation of the game board. The feedback we have had until now during the development of the game has been compile-time errors, error reports from testing and normal program execution with no output.

The next step in the development is writing a textual representation of the game board, as it will look at the start of the game.

We can divide the print problem into 2 steps:

- 1** Generate a text representation of the game board.
- 2** Write the text representation of the game board to the terminal window.

We require a class `GameTextUserInterface` with a method `gameBoardRowAsText()`, that takes a `GameBoard` object and returns a string that visualises the board as text. We require that a player of the game can choose the column to drop a piece in. It would be helpful if the printout showed column indices, so that a player can use a column index to indicate a column. Figure 14.3 shows the textual representation of an empty game board.

**FIGURE 14.3** Textual representation of the game board

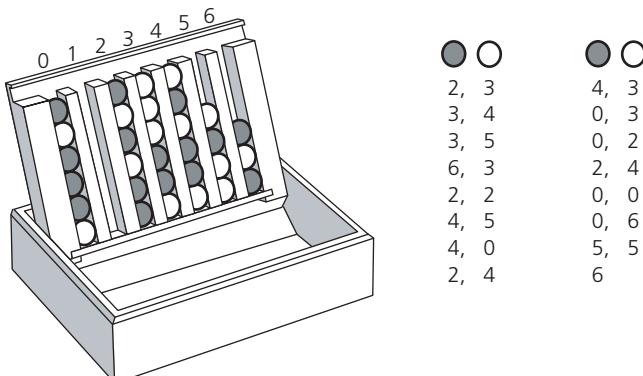
```
0 1 2 3 4 5 6
|
|
|
|
|
|
=====
=====
```

### Simulated game play

Each move in the four-in-a-row game can be made with a digit that represents the column in which the piece should be dropped. The moves in a game can therefore be represented by a sequence of digits. The sequence `2334356322454024430302240006556` is an example of realistic moves between two players of a four-in-a-row game. The outcome of these moves can be seen in Figure 14.4.



**FIGURE 14.4** Game board after several moves have been performed



One player drops 'X' pieces and the other drops '0' pieces in turn, illustrated by the black and white pieces in Figure 14.4, respectively. The game in Figure 14.4 begins with an 'X' piece in column 2. With a helper method, we can easily create GameBoard objects with the state of the board after a given sequence of moves. Program 14.16 shows this helper method, `buildGameBoard()`, in the class GameSimulator.

Using this method we can simplify the test methods `testDropPiece()` and `testStacking()`, as shown in Program 14.17. Here repeated calls to the `dropPieceDownColumn()` method are replaced by a single call to the `buildGameBoard()` method. To reassure us, our JUnit tests can be run to show that the new helper method `buildGameBoard()` works correctly.

#### PROGRAM 14.16 Simulating a sequence of moves

```
// File: GameSimulator.java
public class GameSimulator {
 public static GameBoard buildGameBoard(String moveSequence) {
 GameBoard board = new GameBoard();
 for (int i = 0; i < moveSequence.length(); ++i) {
 boolean isEvenNumberedMove = i % 2 == 0;
 char piece = 'X';
 if (!isEvenNumberedMove)
 piece = '0';
 String digit = moveSequence.substring(i, i + 1);
 int column = Integer.parseInt(digit);
 board.dropPieceDownColumn(piece, column);
 }
 return board;
 }
}
```



## PROGRAM 14.17 Testing of new helper method for simulating the game

```
// File: TestGameBoard.java
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertFalse;
import org.junit.Test;

public class TestGameBoard {
 @Test
 public void testEmptyCell() {
 GameBoard board = new GameBoard();
 // Bottom row has index 0.
 int row = 0;
 // The leftmost column also has index 0
 int column = 0;
 assertEquals("Empty", GameBoard.EMPTY_CELL, board.pieceAt(column, row));
 }

 @Test
 public void testDropPiece() {
 GameBoard board = GameSimulator.buildGameBoard("02");
 /* The bottom rows should now look like this:

 1 |
 0 | X . 0

 / 0 1 2 3 4 5 6 \
 */
 assertEquals("Cell 0,0", 'X', board.pieceAt(0, 0));
 assertEquals("Cell 1,0", GameBoard.EMPTY_CELL, board.pieceAt(1, 0));
 assertEquals("Cell 2,0", '0', board.pieceAt(2, 0));
 assertEquals("Cell 0,1", GameBoard.EMPTY_CELL, board.pieceAt(0, 1));
 }

 @Test
 public void testStacking() {
 GameBoard board = GameSimulator.buildGameBoard("33");
 /* The bottom rows should now look like this:

 2 |
 1 | . . . 0 . . .
 0 | . . . X . . .

 / 0 1 2 3 4 5 6 \
 */
 assertEquals("First dropped", 'X', board.pieceAt(3, 0));
 assertEquals("Second dropped", '0', board.pieceAt(3, 1));
 assertEquals("Only two stacked", GameBoard.EMPTY_CELL, board.pieceAt(3, 2));
 }
}
```



```

 }

 @Test
 public void testFullColumn() {
 GameBoard board = new GameBoard();
 int column = 3;
 char[] pieces = {'X', 'O', 'X', 'O', 'X', 'O'};
 for (char piece : pieces) {
 assertFalse(board.isColumnFull(column));
 board.dropPieceDownColumn(piece, column);
 }
 assertTrue(board.isColumnFull(column));
 }
}

```

---

## Method for printing game board

Program 14.18 is a first attempt at writing a `GameTextUserInterface` class with a `gameBoardRowAsText()` method that prints the game board, as shown in Figure 14.3. In order to test the program manually, we give the class `GameTextUserInterface` a `main()` method that creates and prints a game board. The state of the board is created based on the sequence of moves specified on the command line, and calling the `buildGameBoard()` method in the `GameSimulator` class with the sequence of moves as argument.

### PROGRAM 14.18 First attempt at printing the game board

```

// File: GameTextUserInterface.java
public class GameTextUserInterface {
 static String gameBoardRowAsText(GameBoard board, int row) {
 final int NUMBER_OF_COLUMNS = 7;
 String rowText = "|";
 for (int column = 0; column < NUMBER_OF_COLUMNS; ++column) {
 char state = board.pieceAt(column, row);
 if (state == GameBoard.EMPTY_CELL) {
 rowText += ".";
 } else {
 rowText += state;
 }
 rowText += " ";
 }
 return rowText + "|";
 }

 static String gameBoardAsText(GameBoard board) {
 String text = " 0 1 2 3 4 5 6\n";
 final int TOPMOST_ROW = 5;
 for (int row = TOPMOST_ROW; row >= 0; --row)

```



```

 text += gameBoardRowAsText(board, row) + "\n";
 return text + "=====\n";
}

public void showGameBoard(GameBoard board) {
 System.out.print(gameBoardAsText(board));
}

public static void main(String[] args) {
 GameTextUserInterface ui = new GameTextUserInterface();
 String move = "";
 if (args.length > 0)
 move = args[0];
 ui.showGameBoard(GameSimulator.buildGameBoard(move));
}
}

```

Program output:

```

>java GameTextUserInterface 2334356322454024430302240006556
0 1 2 3 4 5 6
|X . X 0 0 . . |
|0 . 0 0 X . . |
|X . X 0 0 0 . |
|X . 0 X X X X |
|X . X X X 0 0 |
|0 . X 0 0 0 X |
=====

```

As the output from Program 14.18 shows, the printout is not what we expected. Closer inspection of the printout shows that there is an extra space before the last occurrence of the character '|' on each row. The last column in each row must be handled specifically in the `gameBoardRowAsText()` method to rectify this problem.

Program 14.19 shows the revised version of the `gameBoardRowAsText()` method. The printout now looks right. This anomaly was not reported by JUnit because we had not written any test methods to check the printout. Some programmers might prefer to write test methods for the output as well, thereby avoiding all manual testing.

### PROGRAM 14.19 Printing the game board

```

// File: GameTextUserInterface.java
public class GameTextUserInterface {
 static String gameBoardRowAsText(GameBoard board, int row) {
 final int NUMBER_OF_COLUMNS = 7;
 final int LAST_COLUMN = NUMBER_OF_COLUMNS - 1;
 String rowText = "|";
 for (int column = 0; column < LAST_COLUMN; ++column) {
 char state = board.pieceAt(column, row);

```



```

 if (state == GameBoard.EMPTY_CELL) {
 rowText += ".";
 } else {
 rowText += state;
 }
 rowText += " ";
 }
 char state = board.pieceAt(LAST_COLUMN, row);
 if (state == GameBoard.EMPTY_CELL) {
 rowText += ".";
 } else {
 rowText += state;
 }
 return rowText + "|";
}

// ... other members are the same ...
}

```

Program output:

```

0 1 2 3 4 5 6
|X . X 0 0 . .
|0 . 0 0 X . .
|X . X 0 0 0 .
|X . 0 X X X X |
|X . X X X 0 0 |
|0 . X 0 0 0 X |
=====

```

## 14.5 Refactoring program code

The last change in the `gameBoardRowAsText()` method resulted in code duplication (the two `if` statements in Program 14.19), and this is a sign that the code should be re-written. With tests that verify that the code functions correctly, we can restructure the code, safe in the knowledge that the tests will catch any errors we make. Restructuring of the program code that does not change the behaviour of the program, is called *refactoring* of the code.

We reduce the code duplication in the `gameBoardRowAsText()` method by moving the code for determining the correct character in a cell to a separate method, `gameBoardCellAsText()`, as shown in Program 14.20.

By running a new manual test of the printout we can check that the changes we have made have not introduced any errors. The printout is still as shown in Program 14.19, and we can be sure that the program behaves as it did before refactoring.



The program is now at a stage where we can use it to play a primitive form of four-in-a-row. The command line can specify a sequence of moves, and the program prints the game board after the moves have been carried out. By running the program many times, as shown in Figure 14.5, we can play this simplified form of the game.

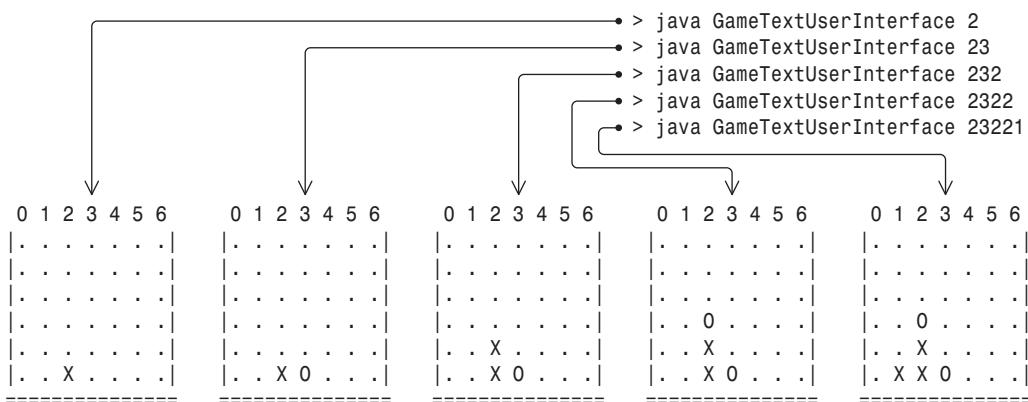
### PROGRAM 14.20 Refactoring the method `gameBoardRowAsText()`

```
// File: GameTextUserInterface.java
public class GameTextUserInterface {
 static String gameBoardRowAsText(GameBoard board, int row) {
 final int NUMBER_OF_COLUMNS = 7;
 final int LAST_COLUMN = NUMBER_OF_COLUMNS - 1;
 String rowText = "|";
 for (int column = 0; column < LAST_COLUMN; ++column)
 rowText += gameBoardCellAsText(board, column, row) + " ";
 return rowText + gameBoardCellAsText(board, LAST_COLUMN, row) + "|";
 }

 static char gameBoardCellAsText(GameBoard board, int column, int row) {
 char state = board.pieceAt(column, row);
 if (state == GameBoard.EMPTY_CELL)
 return '.';
 return state;
 }

 // ... other members are the same ...
}
```

**FIGURE 14.5** A primitive form of four-in-a-row



The program has about the same level of functionality as the physical four-in-a-row-game board, but with the following limitations:



- The players must themselves keep track of whose turn it is to play.

- The game board gives no indication when a player has won and the game is over.

In addition, it is very impractical to start the program each time for a new move, and errors can occur if a player types invalid input on the command line. All these limitations should be overcome before we can say that the program is complete. It will also be a good idea if the program can act as one of the players, i.e. a human player can play against the program.

### BEST PRACTICES

After refactoring, always run the tests to ensure that the program code is not broken.

## 14.6 Interactive four-in-a-row game

### Functional decomposition

We wish to create a four-in-a-row game program that does not have to be re-started every time a new move is made in an on-going game. We wish therefore to create a loop that repeats the following sequence of steps:

- 1 Show the game board
  - 2 Check whether the game is finished, and if that is the case, exit the game
  - 3 Play a move by: (Please indent the list below. Renumber from a-c.)
- a Informing which piece is to be dropped next
  - b Asking for a column number
  - c Dropping the piece in the specified column

The textual user interface is responsible for printing the game board and reading the input in step 1 and step 3b, respectively. The game board is responsible for checking whether the game is finished (in step 2), knowing what the next piece is (in step 3a), and executing the move (in step 3c). Step 1 is implemented by the `showGameBoard()` method in the `GameTextUserInterface` class (see Program 14.21). Step 3c is already implemented by the `dropPieceDownColumn()` method in the `GameBoard` class.

In Program 14.21, we show the modified `main()` method of the `GameTextUserInterface` class, that calls the `runGameLoop()` method, which implements the steps of the game loop outlined above.

The code for the game loop in the `runGameLoop()` method delegates the responsibility for the steps to other methods. Complex tasks can in this way be decomposed into manageable subtasks. The method `playOneMove()` in Program 14.21 does the decomposition of step 3 in the game loop. In the next two subsections we look at how the subtasks for step 3 and step 2 are implemented.



## PROGRAM 14.21 Text interface with the main loop and user interaction

```
// File: GameTextUserInterface.java
public class GameTextUserInterface {

 // ... other members are the same ...

 GameBoard board;

 GameTextUserInterface() {
 board = new GameBoard();
 }

 public void showGameBoard() {
 System.out.print(gameBoardAsText(board));
 }

 public static void main(String[] args) {
 GameTextUserInterface ui = new GameTextUserInterface();
 ui.runGameLoop();
 }

 /** The game loop */
 void runGameLoop() {
 while (true) {
 showGameBoard(); // Step 1
 if (board.isGameOver()) // Step 2
 return;
 playOneMove(); // Step 3
 }
 }

 /** Step 3: Play a move */
 void playOneMove() {
 char piece = board.nextPiece();
 showNextPiece(piece); // Step 3a
 int column = requestColumnSelection(); // Step 3b
 board.dropPieceDownColumn(piece, column); // Step 3c
 }

 /** Step 3a: Inform about next piece.
 void showNextPiece(char piece) {
 System.out.printf("Next piece: %s\n", piece);
 }

 /** Step 3b: Ask for the column number.
 int requestColumnSelection() {
 Scanner scanner = new Scanner(System.in);
 while (true) {

```



```

 System.out.print("Select a column [0-6]: ");

 if (scanner.hasNextInt()) {
 int column = scanner.nextInt();
 if (board.isValidColumnSelection(column)) {
 scanner.nextLine(); // Skip the rest of the line.
 return column;
 }
 }
 scanner.nextLine(); // Skip the rest of the line.
 System.out.println("Invalid column. Try again.");
 }
}
}

```

### Dropping the next piece (step 3a + 3b)

Step 3a of the game loop is straightforward and is implemented by the `showNextPiece()` method in the `GameTextUserInterface` class, as shown in Program 14.21.

Step 3b of the game loop is implemented by the method `requestColumnSelection()` in Program 14.21, that reads the column number from the player, and checks that a valid column number is typed, by calling the helper method `isValidColumnSelection()` in the `GameBoard` class. This last operation is seen as the responsibility of the game board.

The method `showGameBoard()` in Program 14.21 does not take the `GameBoard` object as parameter any longer, as this object is now accessible through the `board` field in the `GameTextUserInterface` class.

### A simplified end of game test (step 2)

The `GameBoard` class is now missing the methods `isValidColumnSelection()`, `nextPiece()` and `isGameOver()` that are called from the `GameTextUserInterface` class. Their implementation is shown in Program 14.22. The first two methods are easy to implement. A counter has been introduced to keep track of the number of moves. For the time being we use a simplified `isGameOver()` method that ignores winner configurations of the game and will allow 42 moves in every game.

#### PROGRAM 14.22 Playing the game

```

// File: GameBoard.java
public class GameBoard {

 // ... other members are the same ...

 int moveCounter;

 void dropPieceDownColumn(char piece, int column) {

```



```

 cellContents[column][findBottommostVacantRow(column)] = piece;
 ++moveCounter;
 }

 boolean isValidColumnSelection(int column) {
 if ((column < 0) || (column > 6))
 return false;
 return !isColumnFull(column);
 }

 char nextPiece() {
 boolean isEvenNumberedMove = moveCounter % 2 == 0;
 if (isEvenNumberedMove)
 return 'X';
 } else {
 return 'O';
 }
 }

 boolean isGameOver() {
 return moveCounter >= 42;
 }
}

```

```

// File: GameTextUserInterface.java
public class GameTextUserInterface {
 // ... other members are the same ...
 public static void main(String[] args) {
 GameTextUserInterface ui = new GameTextUserInterface();
 ui.runGameLoop();
 }
 // ... other members are the same ...
}

```

Playing the game:

```
> java GameTextUserInterface
```

```
0 1 2 3 4 5 6
```

```
| |
| |
| |
| |
| |
| |
| |
=====
```

Next piece: X

Select a column [0-6]: 2

```
0 1 2 3 4 5 6
```

```
| |
| |
| |
```



```

| |
| |
| . . X . . . |
=====
Next piece: 0
Select a column [0-6]: 3
 0 1 2 3 4 5 6
| |
| |
| |
| |
| |
| . . X 0 . . |
=====
Next piece: X
Select a column [0-6]: to
Invalid column. Try again.
Select a column [0-6]: 9
Invalid column. Try again.
Select a column [0-6]: ...

```

---

It is now possible to play several moves in a game, as shown in Program 14.22. If you do not want to play all the 42 moves in the game, you can type the `Ctrl+c` key combination to terminate the program.

The simplest way to test the `nextPiece()` method is to modify the `buildGameBoard()` method in the `GameSimulator` class, as shown in Program 14.23. Since the `buildGameBoard()` method has already been tested, the `nextPiece()` method will now be indirectly tested. This change allows us to simplify the `buildGameBoard()` method and remove duplicate code.

The method `isValidColumnSelection()` can be tested by writing a new test method `testIsValidColumnSelection()` in the `TestGameBoard` class, shown in Program 14.23. JUnit tells us that the code is still passing all the tests.

### PROGRAM 14.23 Simplified simulation and testing of the column selection

```

// File: GameSimulator.java
public class GameSimulator {
 public static GameBoard buildGameBoard(String moveSequence) {
 GameBoard board = new GameBoard();
 for (int i = 0; i < moveSequence.length(); ++i) {
 int column = Integer.parseInt(moveSequence.substring(i, i+1));
 board.dropPieceDownColumn(board.nextPiece(), column);
 }
 return board;
 }
}

```



```
// File: TestGameBoard.java
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertFalse;
import org.junit.Test;

public class TestGameBoard {

 // ... other members are the same ...
 @Test
 public void testIsValidColumnSelection() {
 GameBoard board = GameSimulator.buildGameBoard("111111");
 assertFalse(board.isValidColumnSelection(-1));
 assertTrue(board.isValidColumnSelection(0));
 assertFalse(board.isValidColumnSelection(1));
 assertTrue(board.isValidColumnSelection(6));
 assertFalse(board.isValidColumnSelection(7));
 }
}
> java -ea org.junit.runner.JUnitCore TestGameBoard
JUnit version 4.4
.....
Time: 0.031

OK (5 tests)
```

## 14.7 Ending a game

An important part of the game program still remains to be implemented, namely determining when a game is finished, either when no more moves can be made or when a winner has been declared. We have delegated this responsibility to the `isGameOver()` method. Implementing this method to handle all game configurations is a crucial part of the game. We therefore develop this method in steps, by defining new tests that gradually demand that increasingly more requirements are met by the method.

### No winners

If no cells in the game board are vacant, and no winner has been declared so far, the game is over. There are many game configurations where there is no winner. As a first test of the `isGameOver()` method, we require that it gives the right answer when such a configuration occurs, for example, as shown in Figure 14.6.

The test method `testSimpleGameStatus()` in Program 14.24 tells us that the current implementation of the `isGameOver()` method works fine in cases where there is no winner (for example, as the ones in Figure 14.6). For the moment, the program can be used to manually run through the sequence of moves tested above.



**FIGURE 14.6** Game without winners

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ○ | ○ | ○ | ✗ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| ✗ | ✗ | ✗ | ○ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| ○ | ○ | ○ | ✗ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| ✗ | ✗ | ✗ | ○ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| ✗ | ✗ | ✗ | ✗ | ○ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |

0000001111122222243333334444455555566666  
assertFalse(lastRound.isGameDone())

0000001111122222243333334444455555566666  
assertTrue(noWinner.isGameDone())

#### PROGRAM 14.24 No winner configurations

```
// File: TestGameBoard.java
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertFalse;
import org.junit.Test;

public class TestGameBoard {

 // ... other members are the same ...
 @Test
 public void testSimpleGameStatus() {
 GameBoard start = new GameBoard();
 assertFalse(start.isGameOver());
 GameBoard inProgress = GameSimulator.buildGameBoard("340222244");
 assertFalse(inProgress.isGameOver());
 GameBoard lastMove = GameSimulator.buildGameBoard(
 "00000011111222224333334444455555566666");
 assertFalse(lastMove.isGameOver());
 GameBoard noWinner = GameSimulator.buildGameBoard(
 "00000011111222224333334444455555566666");
 assertTrue(noWinner.isGameOver());
 }
}
```

---

#### Four pieces in a row

If one of the players in the meantime manages to place four of her pieces consecutively in a line, then that player is the winner. Figure 14.7 shows such a winning configuration. The program must be able to determine that one of the players has won. To test whether the `isGameOver()` method can determine such a winning configuration, we write a new test



method. Such a test method is shown in Program 14.25. This test fails on the configuration in Figure 14.7, and shows that the current implementation of the `isGameOver()` method does not accept the winning configuration in Figure 14.7.

**FIGURE 14.7**

Diagonal winning configuration

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| ○ |   |   | X |   |   |   |
| ○ | ○ |   | ○ | X | X | X |
| ○ | X |   | X | X | X | ○ |
| X | X |   | ○ | ○ | ○ | X |
| X | X |   | ○ | ○ | X | ○ |
| X | ○ | X | X | ○ | ○ | ○ |

332401133414455556110666004050633  
assertTrue(xWinnerDiagonal.isGameOver())

**PROGRAM 14.25**

Unsuccessful identification of winning configuration

```
// File: TestGameBoard.java
import static org.junit.Assert.assertTrue;
import org.junit.Test;

public class TestGameBoard {

 // ... other members are the same ...
 @Test
 public void testWinnerStatus() {
 GameBoard xWinnerVertical = GameSimulator.buildGameBoard("2324252");
 assertTrue(xWinnerVertical.isGameOver());
 GameBoard oWinnerHorizontal = GameSimulator.buildGameBoard("06142335");
 assertTrue(oWinnerHorizontal.isGameOver());
 GameBoard xWinnerDiagonal = GameSimulator.buildGameBoard(
 "332401133414455556110666004050633");
 assertTrue(xWinnerDiagonal.isGameOver());
 }
}
```

JUnit reports:

```
JUnit version 4.4
.....E
Time: 0.047
There was 1 failure:
1) testWinnerStatus(TestGameBoard)
java.lang.AssertionError:
 at org.junit.Assert.fail(Assert.java:74)
```



```

at org.junit.Assert.assertTrue(Assert.java:37)
at org.junit.Assert.assertTrue(Assert.java:46)
at TestGameBoard.testWinnerStatus(TestGameBoard.java:89)
...
at org.junit.runner.JUnitCore.main(JUnitCore.java:44)

```

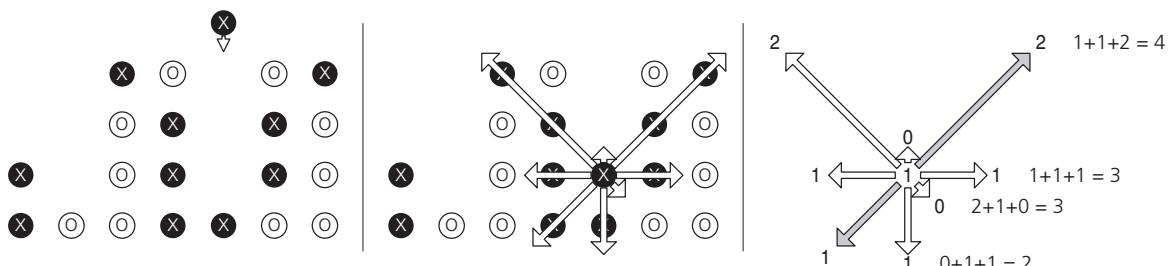
FAILURES!!!

Tests run: 7, Failures: 1

The program must test whether each piece when it is dropped can be part of a sequence of one's own pieces. As shown in Figure 14.8, the number of one's own pieces are counted in each of the vertical, horizontal, and the two diagonal directions. If the number of one's own pieces lying consecutively in any of the above directions is four or more, then that player is the winner, and the game is over. The code implementing these requirements is shown in Program 14.26.

After the `dropPieceDownColumn()` method has placed a piece, it will check if the placing of the current piece forms a winning configuration. This is determined by a call to the method `longestSequenceContainingCell()` (see Figure 14.8). The method `numOfConsecutivePieces()` is called to find the number of pieces of the same kind that occur consecutively in each direction. The `longestSequenceContainingCell()` method sums the number of pieces on both sides of the current piece in each direction, including the current piece itself. Refactoring resulted in a new enum type, `Direction`, that represents the directions used in determining a winning configuration.

**FIGURE 14.8** Counting consecutive pieces in all directions



**PROGRAM 14.26** Game board that finds a winning configuration

```

// File: GameBoard.java
enum Direction {
 UP(0, 1), LEFT(1, 0), DIAGONAL1(1, 1), DIAGONAL2(1, -1),
 DOWN(-UP.columnStep,-UP.rowStep),
 RIGHT(-LEFT.columnStep,-LEFT.rowStep),
 OPPOSITE_DIAGONAL1(-DIAGONAL1.columnStep,-DIAGONAL1.rowStep),
 OPPOSITE_DIAGONAL2(-DIAGONAL2.columnStep,-DIAGONAL2.rowStep);

 final static Direction[] DIRECTIONS
}

```



```
= {UP, LEFT, DIAGONAL1, DIAGONAL2};
final static Direction[] OPPOSITE_DIRECTIONS
= {DOWN, RIGHT, OPPOSITE_DIAGONAL1, OPPOSITE_DIAGONAL2};

int columnStep;
int rowStep;

Direction(int dx, int dy) {
 columnStep = dx;
 rowStep = dy;
}

Direction opposite() {
 return OPPOSITE_DIRECTIONS[this.ordinal()];
}
}

public class GameBoard {
 public static final int NO_VACANT_ROW = -1;
 public static final char EMPTY_CELL = '\u0000';
 public static final int ROW_COUNT = 6;
 public static final int COLUMN_COUNT = 7;
 public static final int CELL_COUNT = ROW_COUNT * COLUMN_COUNT;

 char[][] cellContents = new char[COLUMN_COUNT][ROW_COUNT];
 int moveCounter;
 char winner;

 GameBoard() {
 assert cellContents[0][0] == EMPTY_CELL;
 }

 char pieceAt(int column, int row) {
 return cellContents[column][row];
 }

 int findBottommostVacantRow(int column) {
 char[] columnContents = cellContents[column];
 for (int row = 0; row < columnContents.length; ++row) {
 if (columnContents[row] == EMPTY_CELL)
 return row;
 }
 return NO_VACANT_ROW;
 }

 boolean hasWinner() {
 return winner != '\u0000';
 }

 boolean isGameOver() {
```



```
 return (moveCounter >= CELL_COUNT) || hasWinner();
 }

void dropPieceDownColumn(char piece, int column) {
 assert !isGameOver();
 int row = findBottommostVacantRow(column);
 cellContents[column][row] = piece;
 if (longestSequenceContainingCell(column, row) >= 4)
 winner = piece;
 ++moveCounter;
}

int longestSequenceContainingCell(int column, int row) {
 int largestLength = 1;
 // Look for sequences of similar pieces in all directions.
 for (Direction direction : Direction.DIRECTIONS) {
 int piecesInALine = 1 + numOfConsecutivePieces(column, row, direction)
 + numOfConsecutivePieces(column, row, direction.opposite());
 largestLength = Math.max(largestLength, piecesInALine);
 }
 return largestLength;
}

int numOfConsecutivePieces(int column, int row, Direction direction) {
 char piece = pieceAt(column, row);
 int count = 0;
 while (true) {
 column += direction.columnStep;
 row += direction.rowStep;
 if (!positionContainsPiece(column, row, piece))
 break;
 ++count;
 }
 return count;
}

boolean validColumnIndex(int column) {
 return (column >= 0) && (column < COLUMN_COUNT);
}

boolean validRowIndex(int row) {
 return (row >= 0) && (row < ROW_COUNT);
}

boolean positionContainsPiece(int column, int row, char piece) {
 return validColumnIndex(column)
 && validRowIndex(row)
 && (pieceAt(column, row) == piece);
}
```



```

boolean isColumnFull(int column) {
 char[] columnContents = cellContents[column];
 int indexOfTopmostCell = columnContents.length - 1;
 char topmostCell = columnContents[indexOfTopmostCell];
 return topmostCell != EMPTY_CELL;
}

boolean isValidColumnSelection(int column) {
 return validColumnIndex(column)
 && !isColumnFull(column);
}

char nextPiece() {
 boolean isEvenNumberedMove = moveCounter % 2 == 0;
 if (isEvenNumberedMove) {
 return 'X';
 } else {
 return 'O';
 }
}
}

```

The collection of tests made it is easy to experiment with different strategies, as the tests give a clear message when something does not function as expected. Refactoring has helped to minimise the lines of code in each method. Combined with suitable method and variable names, the resulting source code of each method is easy to read and understand.

The game can now be played by two players, and it will end as soon as a winner is declared.

## 14.8 Machine-controlled player

Computer-based board games often provide a machine-controlled player that the user can play against. The functionality that reads the column selection from the player via the terminal window and plays a move is implemented in the methods `playOneMove()` and `requestColumnSelection()` in the class `GameTextUserInterface` from Program 14.21. In order to provide a machine-controlled player, we must change the `playOneMove()` method, so that the program only asks one of the two players for the column selection, and takes on the role of the second player and chooses the columns for this player itself.

### The interface `IPlayer` and implementations

It is a good idea to implement the new functionality by creating user-controlled and machine-controlled players. The interface `IPlayer` in Program 14.27 declares the method `performMove()` that all game players must implement. Such players are expected to choose a valid column, and drop a piece in that column. We can create player objects of classes that implements the `IPlayer` interface, and refer to them by references of the type `IPlayer`.



## PROGRAM 14.27 IPlayer interface and its implementations

```

// File: IPlayer
interface IPlayer {
 void performMove(GameBoard board); // (1)
}

// File: TerminalPlayer.java
public class TerminalPlayer implements IPlayer {
 GameTextUserInterface ui;

 TerminalPlayer(GameTextUserInterface ui) {
 this.ui = ui;
 } // (2)

 public void performMove(GameBoard board) {
 assert !board.isGameOver();
 assert board == ui.board;
 board.dropPieceDownColumn(board.nextPiece(),
 ui.requestColumnSelection()); // (3)
 }
}

// File: RandomPlayer.java
public class RandomPlayer implements IPlayer {
 public void performMove(GameBoard board) {
 assert !board.isGameOver();

 int choice;
 do {
 choice = (int) (Math.random() * GameBoard.COLUMN_COUNT); // (4)
 } while (!board.isValidColumnSelection(choice));

 board.dropPieceDownColumn(board.nextPiece(), choice); // (5)
 }
}

```

In Program 14.27, the `TerminalPlayer` class implements a player that plays via the textual user interface. The game player stores a reference to the user interface at (2), and uses this reference to ask the player about the column selection at (3).

In Program 14.27, the `RandomPlayer` class implements a machine-controlled player that chooses the column randomly. The selection is made by calling the static method `random()` in the class `java.lang.Math` to generate a pseudorandom number, and converts it to a column number at (4). We generate the column number in a loop whose condition checks that the column chosen is not full. Since calling the `performMove()` method when the state of the board represents a finished game is considered a programming error, we can be sure that there is a valid column that is not full. The piece is dropped down that column at (5).



## A game between arbitrary players

The method `playOneMove()` in the `GameTextUserInterface` class from Program 14.21 is now modified and shown in Program 14.28. This code fetches the player by calling the new `nextPlayer()` method in the `GameBoard` class shown in Program 14.28. The `nextPlayer()` method returns an `IPlayer` reference to a player object, for example, objects of the class `TerminalPlayer` and `RandomPlayer`, or any other class that implements the `IPlayer` interface. For the `playOneMove()` method, the implementation of the player is irrelevant as long as the object provides the services defined in the interface.

### PROGRAM 14.28 Incorporating players

```
// File: GameTextUserInterface.java
import java.util.Scanner;

public class GameTextUserInterface {

 void playOneMove() {
 char piece = board.nextPiece();
 showNextPiece(piece);
 IPlayer player = board.nextPlayer();
 player.performMove(board);
 }
 // ... other members are the same ...
}

// File: GameBoard.java
public class GameBoard {
 // ... other members are the same ...

 int moveCounter;
 IPlayer[] players;

 void setPlayers(IPlayer[] bothPlayers) {
 assert bothPlayers.length == 2;
 players = bothPlayers;
 }

 IPlayer nextPlayer() {
 return players[moveCounter % 2];
 }
 // ... other members are the same ...
}
```

---

In Program 14.28, the new method `nextPlayer()` in the `GameBoard` class is responsible for returning a game player that executes the next move. This behaviour resembles the behaviour of the `nextPiece()` method, that returns the next piece to be dropped. The new method uses the move counter and the array of players in the `GameBoard` object. The



`setPlayers()` method in the `GameBoard` class can be called to associate two arbitrary players with the board.

Program 14.29 shows how easy it is to create a game with different kinds of players. Creating game players is done at (1). In this case, an array with a text interface-based player and a player that chooses the column randomly is created at (1). The array of players is associated with the board via the text user interface. This is done by calling the `setPlayers()` method of the `GameBoard` class at (2) in Program 14.29.

We can easily change the array of players at (1) to create new variants of the game:

- Automated game where the program plays against itself:  
`{ new RandomPlayer(), new RandomPlayer() };` [Please indent.]
- Game with the same behaviour as in Program 14.21, where two players play against each other in the terminal window:  
`{ new TerminalPlayer(ui), new TerminalPlayer(ui) };` [Please indent.]

This is an example of object-oriented code that makes it easy to introduce fundamental changes in the behaviour by making small changes in the code.

#### PROGRAM 14.29 User playing against a naive machine-controlled player

```
/** An easy game where the user plays against a computer adversary that
 * chooses moves at random. */
public class EasyGame {
 public static void main(String args[]) {
 GameTextUserInterface ui = new GameTextUserInterface();
 IPlayer[] players = new IPlayer[]
 { new TerminalPlayer(ui), new RandomPlayer() }; // (1)
 ui.board.setPlayers(players); // (2)
 ui.runGameLoop();
 }
}
```

#### BEST PRACTICES

Program using interfaces. This results in a flexible implementation, allowing the program to work with any objects that implement the interfaces.

## 14.9 Class libraries

Source code can often be re-used in other programs, if one plans ahead. Code that is identified as being useful in many contexts is often rewritten as *class libraries* in order to make it available for other programmers and projects.



We have seen the use of class libraries such as JUnit and the Java standard library. Now we will look at how to develop our own class libraries. Class libraries must be developed with great care. They ought to satisfy the following requirements, at least:

- be easy to understand and use
- can be used without making changes to the code in the library
- should not, without good reason, restrain the use of the library

It is difficult to develop classes that can be used by other programs. Experience with development of classes and use of these in different contexts is often necessary before one can design suitable and re-usable abstractions.

## Packages and the import statement

In order to organise classes in libraries, we group classes that logically belong together into *packages*. Packages can be nested. For example, `java.lang` and `java.util` are two packages that are contained in the `java` package. To identify classes in different packages, we use a scheme similar to the one used in hierarchical file systems. In the `java` package, for example, there is a package called `util` that contains a class called `Scanner`. The path `java.util.Scanner` is called the *fully qualified name* of this `Scanner` class, and is different from any other `Scanner` class that might exist. We can use classes from other packages in our code as follows:

```
java.util.Scanner myScanner = new java.util.Scanner();
```

It is also possible to *import* classes from other packages using the `import` statement:

```
import java.util.Scanner;
```

This means that we do not have to specify the fully qualified name of the class when we use the class in our code:

```
Scanner myScanner = new Scanner();
```

Other members that can be contained in packages are interfaces and enum types, and the discussion above applies to them as well. It is also possible to import all package members in a given package by using the wild card '\*' as the member name:

```
import java.util.*; // imports all type names from the java.util package.
```

The wild card notation allows us to refer to any reference type that is declared in the package by its simple name. An `import` statement does not import the type names from nested packages. You have to declare an explicit `import` statement for nested packages.

An `import` statement applies to the *class* file, i.e. the one containing the byte code, not the source code. The byte code is executed at runtime, not the source code.

## Static import

In order to use a static field or a method from a class in another package, we have to use the fully qualified name of the class, unless the package is implicitly imported, as is the case with the `java.lang` package. For example, the following code shows the use of three static methods from the `java.lang.Math` class:

```
long fib = Math.round(Math.pow(0.5 + Math.sqrt(5)/2, n)/Math.sqrt(5));
```



In order to avoid using the fully qualified name of a class when referring to its static members, we first declare *static import* statements to designate the static members we want to use:

```
import static java.lang.Math.round;
import static java.lang.Math.pow;
import static java.lang.Math.sqrt;
```

Afterwards we can use the static members from the class as follows:

```
long fib = round(pow(0.5 + sqrt(5)/2, n)/sqrt(5));
```

It is also possible to import all static members of a class by using the wild card '\*' as the member name:

```
import static java.lang.Math.*;
```

Bear in mind that importing is a notational convenience that saves us from typing fully qualified names when using reference types.

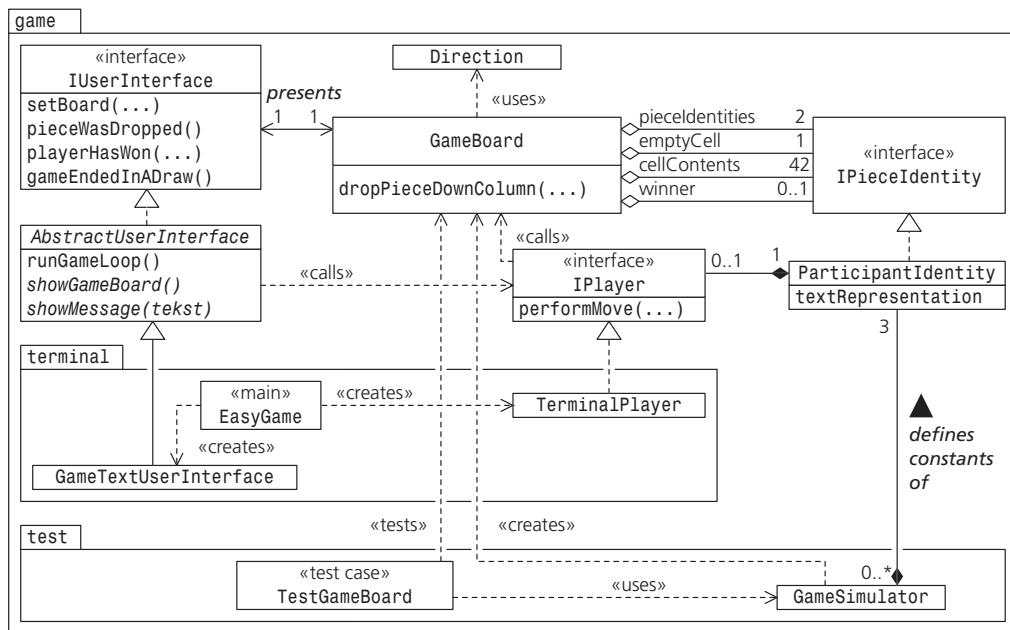
## Declaring packages

We will now write a class library for the four-in-a-row code that we have developed. The code must be reorganised so that it is possible to utilise it as a library. Figure 14.9 shows one way in which the functionality of the game can be distributed among classes in the packages `game`, `game.terminal` and `game.test`:

- The `game` package contains the main functionality that can be used to build different variants of the game.
- The `game.terminal` package contains classes related to running a game in the terminal window.
- The `game.test` package contains classes that are used to test the code in the `game` package.



**FIGURE 14.9** Packages with game functionality



A package declaration is declared at the beginning of the source code file, followed by the declaration of the types that are to be included this package:

```
package game.GameBoard; // File: game/GameBoard.java
public class GameBoard { /* ... */ };
```

The type declarations in a file are placed in the package that is specified in the package declaration. This also implies that several source code files can contain the type declarations to be included in the same package, as long as the package declaration in the different files specify the same package name.

## Designing frameworks

A *framework* is a class library that allows clients to call methods in the class library, but in addition, the code in the library can also call methods in the client code. A framework allows new classes to be defined externally, that provide methods that will be called when the framework executes certain operations. An example of a framework is JUnit, that allows a client to write test methods, and the framework calls these test methods and analyses their results. Frameworks provide flexibility in that they can be extended and used in new contexts that were not imagined when the framework was designed. Frameworks usually provide the ability to extend the functionality through implementation of interfaces or extension of *base classes*. Base classes are discussed in Section 14.11.

Program 14.30 shows the three interfaces `game.IPieceIdentity`, `game.IUserInterface` and `game.IPlayer`, that the four-in-a-row class library in the package `game` provides for extending the functionality.



## PROGRAM 14.30 Interfaces in the `game` package

```

package game; // File: game/IPieceIdentity.java
/**
 * Interface for an object that identifies a piece as belonging to
 * a particular set of pieces. E.g. one object can
 * represent yellow pieces, while another object can represent
 * blue pieces. Pieces r and s are assumed to belong to the same set if
 * (r == s) is true. This interface does not concern itself with how
 * pieces are presented, but classes that implement this
 * interface are free to provide such information.
 */
public interface IPieceIdentity { }

package game; // File: game/IUserInterface.java
public interface IUserInterface {
 void setBoard(GameBoard board);
 void pieceWasDropped();
 void playerHasWon(IPieceIdentity piece);
 void gameEndedInADraw();
}

package game; // File: game/IPlayer
/**
 * A player that can perform moves on a GameBoard. Implementations of
 * this interface can provide strategies for deciding which column to choose
 * for each move.
 */
public interface IPlayer {
 /**
 * Selects one of the eligible columns in the game board, and calls
 * the dropPieceDownColumn() method on the GameBoard object.
 * Player implementations can choose whether to perform the
 * dropPieceDownColumn() call immediately, or at a later time.
 */
 void performMove(IPieceIdentity piece, GameBoard board);
}

```

---

The interface `game.IPieceIdentity` now represents the way in which the pieces are represented, rather than the `game.GameBoard` class having this responsibility. The class `GameBoard` in Program 14.26 used the characters '`'X'`' and '`'0'`' to represent the two kinds of pieces used in the game. This representation was strongly motivated by the presentation in the terminal window, and is not suitable for presentation in a graphical user interface, where one might want to use different colours to identify the pieces. The class `game.GameBoard` does not need to know how to differentiate between the pieces. Neither does the class need to know how the pieces are represented by the user interface, nor which player owns which piece.



The interface `game.IUserInterface` now represents the functionality of a user interface that was previously in the class `game.GameBoard`: presenting the game board, keeping track of the progress in the game, interacting with the user and reporting the state of the game.

The interface `game.IPlayer` has the same role as the `IPlayer` interface from Program 14.27, making it possible to implement different types of players that can make a move. However, there are two differences from the previous version of the `IPlayer` interface:

- 1** The signature of the `performMove()` method is now different, as the piece to be dropped is passed as argument in the method call.
- 2** The implementation of the `IPlayer` interface can choose if it wishes to execute the move immediately or at a later time after the call to the `performMove()` method has returned. This flexibility allows implementations to utilize event-driven programming (see Section 20.5 on page 661 [Please fix Xref format.]). This aspect of the interface is not obvious from the signature of the `performMove()` method, and therefore noted in the documentation of the interface.

Program 14.31 shows the new implementation of the machine-controlled player that implements the `game.IPlayer` interface. The only difference from Program 14.27 is that now it is not necessary to call the `board.nextPiece()` method, since the piece is passed as an argument to the `performMove()` method.

### PROGRAM 14.31 Player with random moves in the `game` package

```
package game; // File: game/RandomPlayer.java

public class RandomPlayer implements IPlayer {
 public void performMove(IPieceIdentity piece, GameBoard board) {
 assert !board.isGameOver();

 int choice;
 do {
 choice = (int) (Math.random() * GameBoard.COLUMN_COUNT);
 } while (!board.isValidColumnSelection(choice));

 board.dropPieceDownColumn(piece, choice);
 }
}
```

#### BEST PRACTICES

Remember that if you have written the appropriate tests, you can make changes with confidence. Changes can be undone if the tests reveal any problems.



## 14.10 Encapsulating implementation details

*Encapsulation* is an abstraction technique for designing programs where the essential information is accessible, but the implementation details are hidden. Encapsulation helps reduce the complexity of the code, allowing only relevant information to be accessible from the outside. One does not have to be a watchmaker in order to use a watch. Similarly, we need not know the implementation details of a class library in order to use the library.

### Access modifiers for package members

A simple form of encapsulation is to hide information by using *access modifiers*. In Program 14.30, the interfaces `IPieceIdentity`, `IUserInterface` and `IPlayer` in the package `game` are declared with the access modifier `public`. As can be seen from Table 14.2, this is necessary in order to use these interfaces outside of the package `game`. For example, if the interface `game.IPlayer` is declared without the access modifier `public`, the compiler will report an error when we try to compile the class `game.terminal.TerminalPlayer` that implements this interface.

TABLE 14.2 Access to package members

| Access context      | <code>public</code> | No access modifier |
|---------------------|---------------------|--------------------|
| In the same package | Yes                 | Yes                |
| In other packages   | Yes                 | No                 |

The class `game.Direction` is a helper class that is only used by the `longestSequenceContainingCell()` and the `numOfConsecutivePieces()` methods in the class `game.GameBoard`. Client code need not know about this class, and it is therefore declared without the access modifier `public` in Program 14.32.

PROGRAM 14.32 Class with no access modifiers

```
package game; // File: game/Direction.java
enum Direction {
 UP(0, 1), LEFT(1, 0), DIAGONAL1(1, 1), DIAGONAL2(1, -1),
 DOWN(-UP.columnStep,-UP.rowStep),
 RIGHT(-LEFT.columnStep,-LEFT.rowStep),
 OPPOSITE_DIAGONAL1(-DIAGONAL1.columnStep,-DIAGONAL1.rowStep),
 OPPOSITE_DIAGONAL2(-DIAGONAL2.columnStep,-DIAGONAL2.rowStep);

 final static Direction[] DIRECTIONS
 = {UP, LEFT, DIAGONAL1, DIAGONAL2};
 final static Direction[] OPPOSITE_DIRECTIONS
 = {DOWN, RIGHT, OPPOSITE_DIAGONAL1, OPPOSITE_DIAGONAL2};
```



```

int columnStep;
int rowStep;

Direction(int dx, int dy) {
 columnStep = dx;
 rowStep = dy;
}

Direction opposite() {
 return OPPOSITE_DIRECTIONS[this.ordinal()];
}
}

```

---

## Access modifiers for class members

The access modifiers shown in Table 14.3 can be used to hide fields and methods of a class. The class `game.GameBoard` in Program 14.33 has been designed to encapsulate the implementation details of the game board. The access modifiers `public`, `protected` and `private` have been used to hide fields and methods in the class in order to provide the right level of accessibility.

**TABLE 14.3 Access to class members**

| Access context                    | public | protected | No access modifier | private |
|-----------------------------------|--------|-----------|--------------------|---------|
| In the same class                 | Yes    | Yes       | Yes                | Yes     |
| In the same package               | Yes    | Yes       | Yes                | No      |
| From subclasses in other packages | Yes    | Yes       | No                 | No      |
| From other code in other packages | Yes    | No        | No                 | No      |

The class `game.GameBoard` in Program 14.33 defines four constants that are used by the clients of the `game.GameBoard` class, and are therefore declared `public`. The constant `NO_VACANT_ROW` is an implementation detail that nobody outside of the class needs to know, and is therefore declared `private`.

The values stored in the fields at (2) can be read by the code in the `game` package, but cannot be changed, since the fields have been declared `final`. The keyword `final` is useful when the reference values in the fields should not be changed, in order to prevent other objects being substituted for the objects referenced by the fields.

The field `moveCounter` at (3) is declared without any access modifier. This field is therefore accessible to all code in the same package.



The fields at (4) are declared `private` and are therefore not directly accessible for the client code outside the `game.GameBoard` class. However, this class provides the helper methods `pieceAt()`, `isColumnFull()` and `hasWinner()`, and allows the client code to change the state with the methods `reset()` and `dropPieceDownColumn()`. These and other public methods and constructors are declared at (5).

The constructors receive information about the players in the game through the `IPieceIdentity` array that is passed as parameter. Earlier versions of the `GameBoard` class were only for two players, the newer version of the `game.GameBoard` class supports one or more players.

The method `reportToUserInterface()` at (6) reports the state of the game to the user interface each time a piece is dropped in a column. This method is declared `protected`, which means that subclasses of the `game.GameBoard` class can change the behaviour of this method. The subclasses need not be in the same package in order to call or override the `protected` method.

The remaining methods from (7) onwards are helper methods that are only used by other methods in the class. These helper methods are implementation details and are declared `private`, and thereby not accessible outside of the class.

### PROGRAM 14.33 Using access modifiers for class members

```
package game; // File: game/GameBoard.java

public class GameBoard {
 // Constants: // (1)
 public static final int ROW_COUNT = 6;
 public static final int COLUMN_COUNT = 7;
 public static final int CELL_COUNT = ROW_COUNT * COLUMN_COUNT;
 private static final int NO_VACANT_ROW = -1;

 // Immutable fields: // (2)
 final IUserInterface ui;
 final IPieceIdentity[] pieceIdentities;
 final IPieceIdentity emptyCell;

 int moveCounter; // (3)

 // private fields with state accessible through methods: // (4)
 private IPieceIdentity[][][] cellContents =
 new IPieceIdentity[COLUMN_COUNT][ROW_COUNT];
 private IPieceIdentity winner;

 // public constructors and methods: // (5)

 public GameBoard(IPieceIdentity[] identities, IPieceIdentity emptyCell) {
 this(identities, emptyCell, null);
 }
```



```

public GameBoard(IPieceIdentity[] identities,
 IPieceIdentity emptyCell, IUserInterface ui) {
 this.emptyCell = emptyCell;
 this.ui = ui;

 assert identities.length > 0;
 pieceIdentities = identities;
 reset();

 if (ui != null) {
 ui.setBoard(this);
 }
}

public void reset() {
 winner = null;
 moveCounter = 0;
 for (IPieceIdentity[] column : cellContents) {
 for (int rowIndex = 0; rowIndex < ROW_COUNT; ++rowIndex) {
 column[rowIndex] = emptyCell;
 }
 }
}

public IPieceIdentity pieceAt(int column, int row) {
 return cellContents[column][row];
}

public boolean hasWinner() {
 return winner != null;
}

public boolean isGameOver() {
 return (moveCounter >= CELL_COUNT) || hasWinner();
}

public void dropPieceDownColumn(IPieceIdentity piece, int column) {
 assert !isGameOver();
 int row = findBottommostVacantRow(column);
 cellContents[column][row] = piece;
 if (longestSequenceContainingCell(column, row) >= 4) {
 winner = piece;
 }
 ++moveCounter;
 reportToUserInterface();
}

public boolean isColumnFull(int column) {
 IPieceIdentity[] columnContents = cellContents[column];
 int indexOfTopmostCell = columnContents.length - 1;
}

```



```

IPieceIdentity topmostCell = columnContents[indexOfTopmostCell];
return topmostCell != emptyCell;
}

public boolean isValidColumnSelection(int column) {
 return validColumnIndex(column)
 && !isColumnFull(column);
}

public IPieceIdentity nextPiece() {
 return pieceIdentities[moveCounter % pieceIdentities.length];
}

protected void reportToUserInterface() { // (6)
 if (ui == null) {
 return;
 }

 ui.pieceWasDropped();
 if (!isGameOver()) {
 return;
 }

 if (hasWinner()) {
 ui.playerHasWon(winner);
 } else {
 ui.gameEndedInADraw();
 }
}

// private constructors and methods: // (7)

private int findBottommostVacantRow(int column) {
 IPieceIdentity[] columnContents = cellContents[column];
 for (int row = 0; row < columnContents.length; ++row) {
 if (columnContents[row] == emptyCell) {
 return row;
 }
 }
 return NO_VACANT_ROW;
}

private int longestSequenceContainingCell(int column, int row) {
 int largestLength = 1;
 // Look for sequences of similar pieces in all directions.
 for (Direction direction : Direction.DIRECTIONS) {
 int piecesInALine = 1 + numOfConsecutivePieces(column, row, direction)
 + numOfConsecutivePieces(column, row, direction.opposite());
 largestLength = Math.max(largestLength, piecesInALine);
 }
}

```



```

 return largestLength;
 }

private int num0fConsecutivePieces(int column, int row, Direction direction) {
 IPieceIdentity piece = pieceAt(column, row);
 int count = 0;
 while (true) {
 column += direction.columnStep;
 row += direction.rowStep;
 if (!positionContainsPiece(column, row, piece)) {
 break;
 }
 ++count;
 }
 return count;
}

private boolean validColumnIndex(int column) {
 return (column >= 0) && (column < COLUMN_COUNT);
}

private boolean validRowIndex(int row) {
 return (row >= 0) && (row < ROW_COUNT);
}

private boolean positionContainsPiece(int column, int row,
 IPieceIdentity piece) {
 return validColumnIndex(column)
 && validRowIndex(row)
 && (pieceAt(column, row) == piece);
}
}

```

## Application Programming Interface (API)

The externally accessible part of a class library is called its *Application Programming Interface* (API). When a class library is used by many clients, it is important that the API of the library is sufficiently *stable*. This means that the API is seldom changed in ways that require that clients change their code.

Here is a list of changes that can be done in a class library, without the danger of introducing compile-time errors for the clients of the library:

- add new packages, interfaces and classes.
- add new fields, methods and constructors to classes.
- delete fields and methods that are not accessible externally, i.e. all members that are not public or protected.
- delete or change classes that are not accessible, i.e. classes that are not public.



- make fields and methods more accessible by changing the access modifier, for example, change private to public.
- change implementation of methods.

These rules only prevent compile-time errors for clients that utilise the API of a package, and not for the code in the package. These rules only guarantee that the API will remain compatible, and do not guarantee that new releases of the class library will behave as the client code expects. The API tells which classes, fields and methods the library provides, but it does not tell explicitly which behaviour can be expected. It is therefore important to write good API documentation that describes all aspects of the class library that are not self-evident. The contract of each method accessible to clients should also be documented.

### BEST PRACTICES

Since a class library will be used by clients, it is crucial that any changes you make to the library should have no or minimal consequences for the clients.

## Limiting access to fields

When a field in an object is made accessible to clients, the object no longer has exclusive control of the field. A client can freely access and change the value in the field, and the object might not even be aware of it. If the field contains a reference value, the client code can not only replace the reference value, but also follow the reference and dig deep into the object hierarchy. This gives the client code a great deal of freedom, and makes changes to the class library in the future difficult, because many implementation details have been exposed in the API. It is therefore often a good idea to limit access to the fields. This can be done by hiding the fields and providing alternative ways of handling the state stored in the fields:

- If the client code needs to set the state of an object only once, consider doing this via the constructor of the class.
- If the client code needs to change the state of an object more than once, a `setState()` method should be considered.

By giving access to the object state through methods, as opposed to direct field access, the class implementation can also verify that the changes in the object state requested by the client code are valid.

### BEST PRACTICES

It is worth giving a careful thought to what should be accessible from a class or a package. Hide implementation details, and keep in mind that what the client does not know, is unlikely hurt your code.



## Tell — don't ask

Operations and the data they require ought to be tightly coupled. Frequent use of direct field access or `getState()` methods in order to dig out the state of other objects, indicates that the behaviour is perhaps implemented in the wrong place in relation to where the state is stored.

The program code should generally *tell* other objects what they should do, instead of *asking* them about the state. When an operation is executed on the state of an object, consider implementing the operation as a method in the class of the object, so that other clients can also call the method to execute the same operation.

### BEST PRACTICES

Write methods so that clients can tell objects in your program what to do, rather than asking them to expose the object state.

## 14.11 Base classes

*Base classes* are classes that are designed to be extended by subclasses. Classes that must be extended before they can be instantiated are called *abstract base classes*.

### Base classes in the game package

The package `game` provides the base classes `AbstractUserInterface` and `ParticipantIdentity`, shown in Program 14.34.

At (1), (2), (3) and (4) the class `game.AbstractUserInterface` provides the base implementation for all the methods declared in the interface `IUserInterface`. The method `runGameLoop()` at (5) and the private helper method `playOneMove()` at (6) implement a general game loop that is independent of any concrete user interface design. The methods `showNextPiece()` and `identityAsText()` only provide a default implementation for how to present the pieces, so the subclasses can readily override these methods.

Many of the methods in the base class delegate tasks to the two abstract methods `showGameBoard()` and `showMessage()`. These methods must be implemented by a subclass in order to be able to create user interface objects based on the interface `game.AbstractUserInterface`.

The base class `game.ParticipantIdentity` is not abstract, and we can therefore create objects of this class. This class will typically be extended by subclasses, if we want to create user interfaces that need to store more information about players in the game. A graphical user interface can, for example, store the colour of the pieces for each player.



### PROGRAM 14.34 Abstract user class in the `game` package

```

package game; // File: game/AbstractUserInterface.java

public abstract class AbstractUserInterface implements IUserInterface {
 protected GameBoard board;
 private boolean loopIsRunning;
 private boolean readyForNextMove;

 public void setBoard(GameBoard board) { // (1)
 this.board = board;
 }

 public void pieceWasDropped() { // (2)
 if (loopIsRunning) {
 readyForNextMove = true;
 showGameBoard();
 } else {
 runGameLoop();
 }
 }

 public void gameEndedInADraw() { // (3)
 showMessage("Game ended in a tie.");
 }

 public void playerHasWon(IPieceIdentity piece) { // (4)
 showMessage(identityAsText(piece) + " has won");
 }

 public void runGameLoop() { // (5)
 readyForNextMove = true;
 showGameBoard();
 loopIsRunning = true;
 while (readyForNextMove && !board.isGameOver())
 playOneMove();
 loopIsRunning = false;
 }

 private void playOneMove() { // (6)
 IPieceIdentity piece = board.nextPiece();
 showNextPiece(piece);
 ParticipantIdentity participant = (ParticipantIdentity) piece;
 readyForNextMove = false;
 participant.performMove(board);
 }

 protected void showNextPiece(IPieceIdentity piece) { // (7)
 showMessage("Next move: " + identityAsText(piece));
 }
}

```



```

}

protected String identityAsText(IPieceIdentity piece) { // (8)
 ParticipantIdentity participant = (ParticipantIdentity) piece;
 return participant.textRepresentation;
}

protected abstract void showGameBoard();
protected abstract void showMessage(String text);
}

package game; // File: game/ParticipantIdentity.java

/** The identity of a participant in the game. Objects of this class can be
 * considered to be a collection of similar game pieces,
 * and can also be considered to be the identity of the player
 * that owns the collection of pieces. */
public class ParticipantIdentity implements IPieceIdentity {
 private IPlayer player;
 public String textRepresentation;

 public ParticipantIdentity() {
 }

 public ParticipantIdentity(IPlayer player, String textRepresentation) {
 this.player = player;
 this.textRepresentation = textRepresentation;
 }

 public void performMove(GameBoard board) {
 player.performMove(this, board);
 }
}

```

---

## The package `game.terminal`

The package `game.terminal` builds on the functionality in the `game` package to implement a text-based four-in-a-row game, corresponding to the game from Program 14.29. The classes in Program 14.35 extend the interface `game.AbstractUserInterface` to create a text-based user interface that prints messages and the game board in the terminal window, and extends the `game.IPlayer` interface to create a player that asks about the column selection via the terminal window.

### PROGRAM 14.35 Terminal user interface in the package `game.terminal`

```

package game.terminal; // File: game/terminal/GameTextUserInterface.java
import game.AbstractUserInterface;

```



```
public class GameTextUserInterface extends AbstractUserInterface {
 protected void showGameBoard() {
 System.out.print(gameBoardAsText());
 }

 protected void showMessage(String text) {
 System.out.println(text);
 }

 private String gameBoardRowAsText(int row) {
 final int NUMBER_OF_COLUMNS = 7;
 final int LAST_COLUMN = NUMBER_OF_COLUMNS - 1;
 String rowText = "|";
 for (int column = 0; column < LAST_COLUMN; ++column)
 rowText += gameBoardCellAsText(column, row) + " ";
 return rowText + gameBoardCellAsText(LAST_COLUMN, row) + "|";
 }

 private String gameBoardCellAsText(int column, int row) {
 return identityAsText(board.pieceAt(column, row));
 }

 private String gameBoardAsText() {
 String text = " 0 1 2 3 4 5 6\n";
 final int TOPMOST_ROW = 5;
 for (int row = TOPMOST_ROW; row >= 0; --row)
 text += gameBoardRowAsText(row) + "\n";
 return text + "=====\\n";
 }
}

package game.terminal; // File: game/terminal/TerminalPlayer.java
import game.GameBoard;
import game.IPieceIdentity;
import game.IPlayer;

import java.util.Scanner;

public class TerminalPlayer implements IPlayer {
 public void performMove(IPieceIdentity piece, GameBoard board) {
 assert !board.isGameOver();
 board.dropPieceDownColumn(piece, requestColumnSelection(board));
 }

 private int requestColumnSelection(GameBoard board) {
 Scanner scanner = new Scanner(System.in);
 while (true) {
 System.out.print("Select a column [0-6]: ");
 }
 }
}
```



```

 if (scanner.hasNextInt()) {
 int column = scanner.nextInt();
 if (board.isValidColumnSelection(column)) {
 scanner.nextLine(); // Skip the rest of the line.
 return column;
 }
 }
 scanner.nextLine(); // Skip the rest of the line.
 System.out.println("Invalid column. Try again.");
 }
}
}

```

Program 14.36 sets up a game between a player that uses the terminal window to get the column selection and a machine-controlled player that chooses the column randomly. Unlike Program 14.29, the characters 'T' and 'R' are used to represent the game pieces for this particular game, instead of the characters 'X' and 'O'.

### PROGRAM 14.36 Running a game against an easy opponent

```

package game.terminal; // File: game/terminal/EasyGame.java
import game.GameBoard;
import game.IPieceIdentity;
import game.ParticipantIdentity;
import game.RandomPlayer;
/** An easy game where the user plays against a computer adversary that
 chooses moves at random. */
public class EasyGame {
 public static void main(String args[]) {
 GameTextUserInterface ui = new GameTextUserInterface();
 IPieceIdentity[] participants = new IPieceIdentity[] {
 new ParticipantIdentity(new TerminalPlayer(), "T"),
 new ParticipantIdentity(new RandomPlayer(), "R")
 };
 IPieceIdentity emptyCell = new ParticipantIdentity(null, ".");
 GameBoard board = new GameBoard(participants, emptyCell, ui);
 ui.runGameLoop();
 }
}

```

#### BEST PRACTICES

Extract common properties and behaviour into abstract base classes, allowing subclasses to provide different implementations.

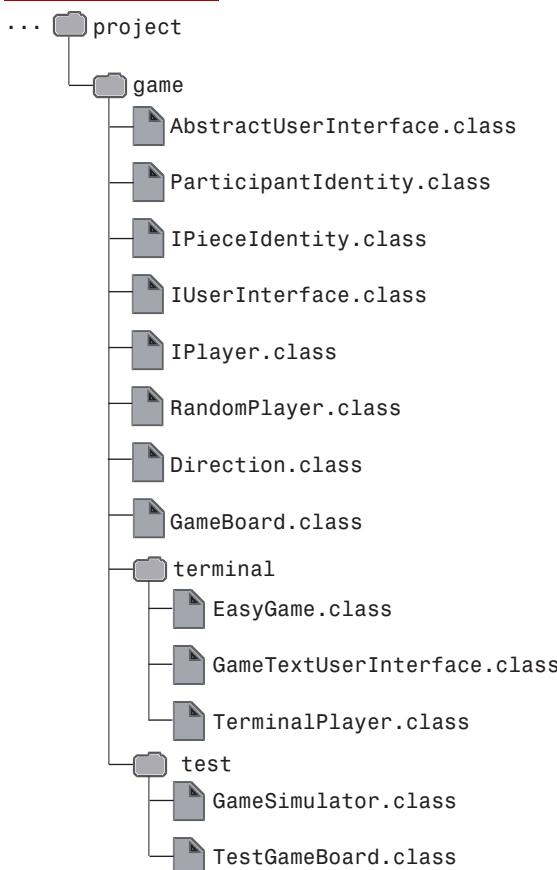


## 14.12 Compiling and running code in packages

Figure 14.9 showed the package hierarchy of our program in UML. In order to install and use packages, the package hierarchy must be mapped to a directory hierarchy on the file system. Figure 14.10 shows the package hierarchy of our program mapped under the directory `project`, which we assume is the work directory on the file system. The hierarchical file system represents the nesting of packages, where a directory corresponds to a package.

We need to compile the files for the source code in Program 14.36, so that the compiler generates the byte code files (i.e. `.class` files) in the hierarchy shown in Figure 14.10. There are several ways to organise the source code files for this purpose. The simplest way is to create the directories shown in Figure 14.10 and to place a source code file in the same directory where we want its byte code file to be generated. From the point of view of *using* a package, the location of the source code files is irrelevant. It is the location of the byte code files that matters.

**FIGURE 14.10** Package hierarchy mapped on the file system





With the setup outlined above, we can compile the game/terminal/EasyGame.java file and all classes that the class game.terminal.EasyGame depends on, by executing the following command in the project directory:

```
> javac game/terminal/EasyGame.java
```

The class files containing the byte code are generated and placed in the respective packages, as shown in Figure 14.10. Now we can run the program by executing the following command in the project directory:

```
> java -ea game.terminal.EasyGame
```

```
0 1 2 3 4 5 6
```

```
| |
| |
| |
| |
| |
| |
| |
=====
```

Next move: T

Select a column [0-6]: 4

```
0 1 2 3 4 5 6
```

```
| |
| |
| |
| |
| |
| |
| T . . |
=====
```

Next move: R

```
0 1 2 3 4 5 6
```

```
| |
| |
| |
| |
| |
| |
| R . . . T . . |
=====
```

Next move: T

Select a column [0-6]: ...

The machine-controlled player (R) is asked about the column selection, and a column is chosen randomly. The game can be terminated by typing the Ctrl+c key combination before it is finished.

## The package game.test

The best way to modify the code from Program 14.21, Program 14.26, Program 14.27 and Program 14.29 to a class library presented in Program 14.30, Program 14.31, Program 14.32, Program 14.33, Program 14.34, Program 14.35 and Program 14.36, is by small incremental steps. For each step, ensure that the code compiles, and that the JUnit tests from Program 14.17 and Program 14.23 report no errors.



Some of the changes introduced in the class library also require that the test code is changed. Program 14.37 shows all the test methods after all the changes have been incorporated.

In order to run the tests found in packages, the fully qualified name of the test case class must be specified. From the output of Program 14.37, we see that the program code still passes all the tests.

### PROGRAM 14.37 Tests for the game in the **test** package

```
package game.test; // File: game/test/GameSimulator.java
import game.GameBoard;
import game.IPieceIdentity;
import game.ParticipantIdentity;

public class GameSimulator {
 public static final IPieceIdentity X_PIECE = new ParticipantIdentity();
 public static final IPieceIdentity O_PIECE = new ParticipantIdentity();
 public static final IPieceIdentity EMPTY_CELL = new ParticipantIdentity();

 public static GameBoard buildGameBoard(String moveSequence) {
 IPieceIdentity[] pieces = new IPieceIdentity[]{ X_PIECE, O_PIECE };
 GameBoard board = new GameBoard(pieces, EMPTY_CELL);
 for (int i = 0; i < moveSequence.length(); ++i) {
 int column = Integer.parseInt(moveSequence.substring(i, i+1));
 board.dropPieceDownColumn(board.nextPiece(), column);
 }
 return board;
 }
}

package game.test; // File: game/test/TestGameBoard.java

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertFalse;
import org.junit.Test;

import game.GameBoard;

public class TestGameBoard {
 @Test
 public void testEmptyCell() {
 GameBoard board = GameSimulator.buildGameBoard("");
 // Bottom row has index 0.
 int row = 0;
 // The leftmost column also has index 0
 int column = 0;
 assertEquals("Empty", GameSimulator.EMPTY_CELL, board.pieceAt(column, row));
 }
}
```



```

}

@Test
public void testDropPiece() {
 GameBoard board = GameSimulator.buildGameBoard("02");
 /* The bottom rows should now look like this:

 1 |
 0 | X . 0

 / 0 1 2 3 4 5 6 \
 */
 assertEquals("Cell 0,0", GameSimulator.X_PIECE, board.pieceAt(0, 0));
 assertEquals("Cell 1,0", GameSimulator.EMPTY_CELL, board.pieceAt(1, 0));
 assertEquals("Cell 2,0", GameSimulator.O_PIECE, board.pieceAt(2, 0));
 assertEquals("Cell 0,1", GameSimulator.EMPTY_CELL, board.pieceAt(0, 1));
}

@Test
public void testStacking() {
 GameBoard board = GameSimulator.buildGameBoard("33");
 /* The bottom rows should now look like this:

 2 |
 1 | . . . 0 . . .
 0 | . . . X . . .

 / 0 1 2 3 4 5 6 \
 */
 assertEquals("First dropped", GameSimulator.X_PIECE, board.pieceAt(3, 0));
 assertEquals("Second dropped", GameSimulator.O_PIECE, board.pieceAt(3, 1));
 assertEquals("Only two stacked", GameSimulator.EMPTY_CELL,
 board.pieceAt(3, 2));
}

@Test
public void testFullColumn() {
 GameBoard board = GameSimulator.buildGameBoard("");
 int column = 3;
 for (int i = 0; i < GameBoard.ROW_COUNT; ++i) {
 assertFalse(board.isColumnFull(column));
 board.dropPieceDownColumn(board.nextPiece(), column);
 }
 assertTrue(board.isColumnFull(column));
}

@Test
public void testIsValidColumnSelection() {
 GameBoard board = GameSimulator.buildGameBoard("111111");
 assertFalse(board.isValidColumnSelection(-1));
}

```



```

 assertTrue(board.isValidColumnSelection(0));
 assertFalse(board.isValidColumnSelection(1));
 assertTrue(board.isValidColumnSelection(6));
 assertFalse(board.isValidColumnSelection(7));
 }

 @Test
 public void testSimpleGameStatus() {
 GameBoard start = GameSimulator.buildGameBoard("");
 assertFalse(start.isGameOver());
 GameBoard inProgress = GameSimulator.buildGameBoard("340222244");
 assertFalse(inProgress.isGameOver());
 GameBoard lastMove = GameSimulator.buildGameBoard(
 "0000001111122222433333444445555566666");
 assertFalse(lastMove.isGameOver());
 GameBoard noWinner = GameSimulator.buildGameBoard(
 "0000001111122222433333444445555566666");
 assertTrue(noWinner.isGameOver());
 }

 @Test
 public void testWinnerStatus() {
 GameBoard xWinnerVertical = GameSimulator.buildGameBoard("2324252");
 assertTrue(xWinnerVertical.isGameOver());
 GameBoard oWinnerHorizontal = GameSimulator.buildGameBoard("06142335");
 assertTrue(oWinnerHorizontal.isGameOver());
 GameBoard xWinnerDiagonal = GameSimulator.buildGameBoard(
 "332401133414455556110666004050633");
 assertTrue(xWinnerDiagonal.isGameOver());
 }
}

```

Compiling and running the tests:

```

> java -ea org.junit.runner.JUnitCore game.test.TestGameBoard
JUnit version 4.4
.....
Time: 0,016

OK (7 tests)

```

### BEST PRACTICES

Practice the test-driven development mantra "red (i.e. test fails), green (i.e. test passes), refactor".



## 14.13 Review questions

1. How can we verify that the program actually does what we expect?
    - a See that the compiler does not report any errors.
    - b Test the program manually and observe the behaviour.
    - c Document the behaviour.
    - d Write and run tests that verify the behaviour automatically.
    - e Create a class library of programs.

(b), (d)

(b) works fine for simple programs, but (d) is more practical for larger programs.
  2. What are the disadvantages of using assert statements for testing?
    - a assert statements do not automatically report the state of variables that are operands in the assertion expression.
    - b assert statements do not report the cases where the assertions fail.
    - c assert statements are not executed unless the option `-enableassertions` or `-ea` is used in the command line.

(a), (c)
  3. Which statements are true about JUnit?
    - a The fully qualified name of the class `Assert` in JUnit is `org.junit.Assert`.
    - b The static methods of the class `Assert` in JUnit can be imported by the following `import` statement:

```
import org.junit.Assert.*;
```

    - c The annotation `Test` in JUnit can be imported by the following `import` statement:

```
import org.junit.Test;
```

    - d The class `org.junit.runner.JUnitCore` has a `main()` method that is passed the names of the test classes from the command line.

(a), (c), (d)

The `import` statement in (b) is missing the keyword `static`, in order to import static members of a class.
  4. Complete the following statement:
- The minimal number of moves in a *completed* four-in-a-row game is \_\_\_\_\_, and the maximum number of moves is \_\_\_\_\_.
- The minimal number of moves in a *completed* four-in-a-row game is 7, and the maximum number of moves is 42.



5. Is it possible to use the `game.GameBoard` class to create the following variants of the four-in-a-row game without changing the code in the `game` package?
- A game where the pieces for both players are identical.
  - A game that has 9 rows and 9 columns.
  - A game where only 3 consecutive pieces in a row are needed to win.
  - A game where each player is allowed to make two moves at a time.
  - A game where two machine players use different strategies to play against each other.

(a), (d), (e)

A game where the pieces for both the players are identical can be created by player identities that use the same representation for both the players. The class `game.GameBoard` only supports boards specified by the constants `ROW_COUNT` and `COLUMN_COUNT`. The method `dropPieceDownColumn()` in the `game.GameBoard` class must be changed to allow a game where only 3 identical consecutive pieces in a row are needed to win. A game where each player is allowed to make two moves at a time can be created by repeating the player identity in the array that is passed as argument to the `game.GameBoard` constructor. Machine players that use new strategies can be introduced by declaring new classes that implement the `IPlayer` interface.

6. Which changes to the `game.GameBoard` class can be done while still ensuring that code using the `game.GameBoard` class will still compile without errors?
- Make the `longestSequenceContainingCell()` method `public`.
  - Change the name of the constant `NO_VACANT_ROW` to `INVALID_ROW_INDEX`.
  - Remove access modifier `public` from the `isGameOver()` method.
  - Add a new constructor that does not take any parameters.
  - Change the name of the method `hasWinner()` to `winnerHasBeenDeclared()`.

(a), (b), (d)

The methods `isGameOver()` and `hasWinner()` can be used by the program code outside the `game` package. Such code will not compile if the methods are made less visible or are given new names. The method `longestSequenceContainingCell()` and the constant `NO_VACANT_ROW` have earlier not been visible outside the package, and can therefore be changed without problem.

## 14.14 Programming exercises

1. We need a helper method `void assertInsideInterval(String message, int minimum, int maximum, int actual)` that verifies that the `actual` value is not less than the `minimum` and not greater than the `maximum`. If the assertion fails, an informative error message should be written to the terminal window.
- Write a helper method based on `assert` statements.
  - Write a helper method based on the JUnit framework.



2. Use the class library with the `game` and `game.terminal` packages to create a four-in-a-row game where 3 players can play against each other on the same game board. Let the text representation of the pieces be the characters '`X`', '`Y`' and '`z`'. It is not necessary to change the implementation of the class library.
3. Write an implementation of the `game.IPlayer` that will always choose the first leftmost vacant column.
4. Write a program that plays a tournament with 100 games between the player that was created in Exercise 14.3 and the player that implements the `game.RandomPlayer` interface. The program must not print anything during the games, but should at the end inform about how many games were won by each player, and how many games ended in a draw.
5. The aim of this exercise is to create the game of tic-tac-toe by refactoring the code to utilise the existing functionality. Tic-tac-toe is played by two players, '`X`' and '`O`', on a game board that is a  $3 \times 3$  grid. The players take turns in marking the vacant cells on the board. A player wins by placing 3 of her pieces consecutively, either vertically, horizontally or diagonally.

- a Write an interface `game.IBoard` that defines the following methods:

```
IPieceIdentity pieceAt(int column, int row);
int numColumns();
int numRows();
```

This interface should be accessible from outside the `game` package. [Please indent.]

- b Write a base class `game.BaseBoard` that implements the interface in (a). This class should not have any classes accessible from outside the class, and should have a method called `placePieceAt` that takes a column index, a row index and a `IPieceIdentity` reference, and can only be used by subclasses. Write a JUnit test that verifies that the pieces placed by the `placePieceAt()` method can be read by the `pieceAt()` method.
- c Change the `GameBoard` class to inherit from the base class in (b), instead of maintaining the board state itself. Do this without introducing incompatible changes in the `game` package. Use JUnit test to verify that the `GameBoard` class still behaves as expected after the changes have been made.
- d Move the method `int longestSequenceContainingCell()` into a new class `game.FindRows` that has the following abstract declaration:

```
public static longestSequenceContainingCell(IBoard board,
 int column, int row);
```

Move also other methods as needed, but make sure not to introduce any incompatibility, and see to that the code still passes the tests. [Please indent this para.]

- e Use the new classes `game.BaseBoard` and `game.FindRows` to create the tic-tac-toe game. Consider how the functionality of the game should be distributed, and how to best make use of the existing functionality in the packages `game` and `game.terminal`. Write tests that check the new functionality, and write the tests before the code that is to be tested.



## Using Dynamic Data Structures

### LEARNING OBJECTIVES

By the end of this chapter, you will understand the following:

- What an ADT (Abstract Data Type) is.
- Using a **StringBuilder** rather than a **String** to handle a character sequence.
- What generic types are, and how to use them.
- How objects can be organized into collections, and what interfaces and classes the Java standard library provides for maintaining collections.
- How to use a list and a set, and understand the difference between them.
- What collection to use in a given situation.
- How and when to use a map.
- Subtyping using the ? wildcard in parameterized types.
- Implementing and calling generic methods.
- Using generic methods from the Java standard library for sorting and searching in lists and arrays.

### INTRODUCTION

Programming can be summarized as follows:

| program = algorithms + data structures [should be indented]

I.e. both algorithms and data structures together constitute a program. In programming the development of algorithms that describe the *steps* necessary to solve a problem, is just as important as the development of data structures that describe how data is *organized* in order to solve a problem.



We have already seen the data structures strings and arrays. In this chapter we discuss data structures that expand and shrink as data is inserted and retrieved. Such data structures are *dynamic*. We present several examples on how to use such data structures to organize data in an effective way, depending on the problem at hand.

## 15.1 Overview

### Abstract Data Types

A *collection* is a structure for storing data. Collections have different properties, depending on how the data is organized by the collection, and a set of *operations* that allow data to be added to or retrieved from the collection. The operations comprise a contract that a client can utilize, without regard to how the collection is implemented. Such a collection, with the accompanying operations, is often called an *abstract data type* (ADT).

Previous chapters have showed examples of two data structures: String objects to handle strings of fixed length (Chapter 4), and arrays to handle a fixed number of values of a particular data type (Chapter 6). The length of the string in a String object cannot be changed and neither can the characters in the string. The length of the array cannot be changed after the array has been created. Both these data structures are called *static data structures*.

Java provides ready-made classes that implement many classical *dynamic data structures*. Table 15.1 gives an overview of the dynamic data structures that we will discuss and use in this chapter. In Chapter 16 ([insert Xref](#)) we will show how some of these dynamic data structures can be implemented.

**TABLE 15.1      Overview of dynamic data structures**

| Class name    | Dynamic data structure                    |
|---------------|-------------------------------------------|
| StringBuilder | <i>Character sequences</i> (Section 15.2) |
| ArrayList     | <i>Lists</i> (Section 15.5)               |
| HashSet       | <i>Sets</i> (Section 15.6)                |
| HashMap       | <i>Maps</i> (Section 15.7)                |

### Organising and manipulating data

Different data structures organise data in different ways. To use a data structure, it might be important to know whether the objects in the collection are *ordered*, i.e. whether each object has a particular position in the collection. In a normal list the objects will be ordered in the sequence they were inserted, called the *insertion order*. A *sorted* list will require that the objects implement a particular interface, for example the Comparable interface, so that the objects can be compared and maintained in natural order. In a normal set the order of the elements is irrelevant.



The most important operations on a dynamic data structure are to *put* data in it and *retrieve* data from it. These two operations are often called *insertion* and *lookup*, respectively. Each operation costs in terms of the time it takes to execute the operation, and choosing a data structure will quite often depend on how expensive these two operations are for a given data structure.

We will look at dynamic data structures that store *objects*. We handle objects with references, and it is the reference value of an object that is actually stored in the data structure. There are variants that can store values of primitive data types, but these are implemented for each primitive data type.

We will not implement a collection for each type of object. We will use *generic types* that allow us to customize collections with objects of particular types (Section 15.3, Section 15.8).

Data structures discussed in this chapter are created in computer memory and are no longer available when the program terminates, unless they are stored in secondary storage. How data can be stored on files in secondary storage is discussed in Section 11.2 and Chapter 19.

## 15.2 Character sequences: `StringBuilder`

The `String` class is not suitable if the characters in a string are modified frequently, as the contents of a `String` object are immutable. Each operation that modifies the contents of a `String` object actually returns a new `String` object with the modified content. The `StringBuilder` class can be used to represent a character sequence that can be modified, and that expands and shrinks dynamically, depending on the number of characters in it. We will use the name *string builder* for such a character sequence. By a string builder we mean an object of the class `StringBuilder`, and by a string we mean an object of the class `String`.

### Creating a character sequence using a `StringBuilder`

A string builder keeps track of its *size*, that indicates the number of characters in its character sequence, and its *capacity*, that indicates the total number of characters that can be inserted before it must be expanded. The default capacity is 16 characters, but other initial capacity can also be specified:

```
StringBuilder buffer1 = new StringBuilder(); // length 0, capacity 16
StringBuilder buffer2 = new StringBuilder(20); // length 0, capacity 20
```

A *buffer* is a storage place for keeping a limited amount of data in memory. Objects of the class `StringBuilder` can be regarded as buffers with characters that can be manipulated.

Since `String` objects contain characters, we can create a string builder from a `String` object:

```
// String builder of length 4, capacity 20 (4+16), character sequence "mama":
StringBuilder buffer3 = new StringBuilder("mama");
String str1 = new String("mia");
```

```
// String builder of length 3, capacity 19 (3+16), character sequence "mia":
StringBuilder buffer4 = new StringBuilder(str1);
```

A string builder's length and capacity can be determined by calling the methods `length()` and `capacity()`:

```
int capacity = buffer4.capacity(); // 19
int length = buffer4.length(); // 3
```

Earlier we saw an example of converting a string to a string builder. We can also convert a string builder to a string by calling the `toString()` method in the `StringBuilder` class:

```
String str2 = buffer3.toString(); // "mama"
```

In contrast to the `String` class, the `StringBuilder` class does not override the `equals()` method for comparing character sequences. In order to compare two string builders for equality we must first convert them to strings:

```
boolean flag = buffer3.toString().equals(buffer4.toString()); // false
```

## Modifying the contents of a `StringBuilder`

Inserting characters in a string builder automatically results in adjusting the capacity of the string builder, if necessary. The class provides positional access to the characters in a string builder, using an index. Legal index values are from 0 to  $n-1$ , where  $n$  is the length of the string builder.

A character, given a particular index, can be read, modified and deleted by calling the methods `charAt()`, `setCharAt()` and `deleteCharAt()`, respectively:

```
// Assume that buffer4 contains the character sequence "mia".
char ch = buffer4.charAt(1); // 'i'
buffer4.setCharAt(0, 'P'); // "Pia"
buffer4.deleteCharAt(2); // "Pi"
```

The overloaded method `append()` can be used to *append characters at the end* of a string builder, i.e. at index  $n$ , where  $n$  is the current length of the string builder. Values of primitive type, strings, character arrays and objects can be appended:

```
// Assume that buffer3 contains the character sequence "mama".
buffer3.append(" mia "); // "mama mia "
buffer3.append('R'); // "mama mia R"
buffer3.append(4); // "mama mia R4"
buffer3.append('U'); // "mama mia R4U"
buffer3.append(new Integer(2)); // "mama mia R4U2"
```

The overloaded method `insert()` can be used to *insert characters at a particular index* in a string builder. Any characters after the insertion index are shifted towards the end of the character sequence in the string builder. The method can be used to insert string representation of different types of values, as is the case with the `append()` method:

```
// Assume that buffer1 is empty.
buffer1.insert(0, "Saturday"); // "Saturday"
buffer1.insert(8, " March "); // "Saturday March "
buffer1.insert(9, 1); // "Saturday 1 March "
```



```

buffer1.insert(buffer1.length(),
 new Integer(2008)); // "Saturday 1 March 2008"
buffer1.insert(10, '.'); // "Saturday 1. March 2008"

```

## Other classes for handling character sequences

The close relationship between the class `String` and the class `StringBuilder` is illustrated by the string concatenation operator, `+`. The compiler actually uses a string builder to perform string concatenation. The code below:

```
String name1 = "seven" + 11; // "seven11"
```

is equivalent to the following code, where a string builder is used to store the characters, and afterwards the string builder is converted to a string:

```
String name2 = new StringBuilder().append("seven").append(11).toString();
```

The Java standard library provides a class called `StringBuffer` that has the same methods as the class `StringBuilder`. However, it is recommended to use the class `StringBuilder`, unless one needs thread-safe character sequences, which is something we will not need in this book. The class `StringBuilder` is more than adequate for our purpose.

Table 15.2 shows a selection of methods from the `StringBuilder` class. Note that a call to a method requiring an index will result in a `StringIndexOutOfBoundsException` if the index value is invalid.



### BEST PRACTICES

Choose the `StringBuilder` class any time you need to work with a character sequence whose characters are modified frequently. This approach will provide a better performance than using the `String` class.

**TABLE 15.2** Selected methods from the `StringBuilder` class

| <code>java.lang.StringBuilder</code> |                                                                                                                                                                |
|--------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>int length()</code>            | Returns the number of characters in the character sequence, i.e. the <i>length</i> of the character sequence.                                                  |
| <code>int capacity()</code>          | Returns the current capacity of the string builder, i.e. the total number of characters that can be inserted currently, before the string builder is expanded. |
| <code>char charAt(int index)</code>  | Returns the character at the <code>index</code> in the character sequence. The first character is at <code>index 0</code> .                                    |



| <b>java.lang.StringBuilder</b>                                                                                                                                                                                                                                                                                                                                                                                                                                            |                                                                                                                                                                                                                                                                                                            |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void setCharAt(int index, char ch)</code>                                                                                                                                                                                                                                                                                                                                                                                                                           | Replace character at the specified <code>index</code> with the character <code>ch</code> .                                                                                                                                                                                                                 |
| <code>void deleteCharAt(int index)</code><br><code>void delete(int startIndex, int endIndex)</code>                                                                                                                                                                                                                                                                                                                                                                       | The first method deletes the character at the specified <code>index</code> . The second method deletes the characters from the specified <code>startIndex</code> , inclusive, to the specified <code>endIndex</code> , exclusive.                                                                          |
| <code>int lastIndexOf(String subString)</code><br><code>int lastIndexOf(String subString, int startIndex)</code>                                                                                                                                                                                                                                                                                                                                                          | Returns the index of the start of the <i>last</i> occurrence of the specified <code>substring</code> in the character sequence, otherwise returns -1. Argument <code>startIndex</code> can be used to start the search from a particular index, otherwise the search starts at index 0.                    |
| <code>String substring(int startIndex)</code><br><code>String substring(int startIndex, int endIndex)</code>                                                                                                                                                                                                                                                                                                                                                              | Returns a string containing the subsequence from <code>startIndex</code> to ( <code>endIndex-1</code> ). The returned string has length ( <code>endIndex-startIndex</code> ). If no <code>endIndex</code> is specified, the subsequence extends to the end of the character sequence.                      |
| <code>StringBuilder append(Object obj)</code><br><code>StringBuilder append(String str)</code><br><code>StringBuilder append(char t)</code><br><code>StringBuilder append(boolean b)</code><br><code>StringBuilder append(int i)</code><br><code>StringBuilder append(long l)</code><br><code>StringBuilder append(float f)</code><br><code>StringBuilder append(double d)</code>                                                                                         | This overloaded method adds characters to the <i>end</i> of the character sequence. The number of characters appended depends on the parameter value which is first converted to a string representation, if necessary. The methods return the reference value of the updated <code>StringBuilder</code> . |
| <code>StringBuilder insert(int index, Object obj)</code><br><code>StringBuilder insert(int index, String str)</code><br><code>StringBuilder insert(int index, char t)</code><br><code>StringBuilder insert(int index, boolean b)</code><br><code>StringBuilder insert(int index, int i)</code><br><code>StringBuilder insert(int index, long l)</code><br><code>StringBuilder insert(int index, float f)</code><br><code>StringBuilder insert(int index, double d)</code> | This overloaded method inserts characters at the specified <code>index</code> . The number of characters inserted depends on the parameter value which is first converted to a string representation, if necessary. The methods return the reference value of the updated <code>StringBuilder</code> .     |
| <code>StringBuilder reverse()</code>                                                                                                                                                                                                                                                                                                                                                                                                                                      | Replaces the character sequence with the result from reversing its characters and returns the reference of the object.                                                                                                                                                                                     |



### java.lang.StringBuilder

|                                            |                                                                                                                           |
|--------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| <code>String toString()</code>             | Returns a <code>String</code> object that contains the current character sequence.                                        |
| <code>void setLength(int newLength)</code> | Sets the length of the current string builder to the value of the parameter. To clear a string builder, pass the value 0. |

## 15.3 Introduction to generic types

ADTs where we can replace the *reference types*, are called *generic types*. This means that we can use the same definition of an ADT on different types of objects. We illustrate such ADTs by developing a generic type that can be used to create pairs of values.

We can define the following class to represent a pair with two int values:

```
class PairInt {
 private int first;
 private int second;
 PairInt (int first, int second) {
 this.first = first;
 this.second = second;
 }
 // other methods
}
```

If we need pairs with other types of primitive values, for example the primitive type `double`, we can write a new class, for example a class named `PairDouble`, where the fields have the primitive type `double`. This is not a good solution, as it involves primarily duplicating code and changing the type of the values that can form a pair in the new class.

If we want to represent a pair with two objects we can utilize the fact that a reference of the `Object` type can refer to objects of all types, and define the class `PairObj` shown in Program 15.1 for this purpose.

### PROGRAM 15.1 Legacy version of a pair with values

```
// Legacy class
public class PairObj {
 private Object first;
 private Object second;
 PairObj () { }
 PairObj (Object first, Object second) {
 this.first = first;
 this.second = second;
 }
 public Object getFirst() { return first; }
```

```
public Object getSecond() { return second; }
public void setFirst(Object firstOne) { first = firstOne; }
public void setSecond(Object secondOne) { second = secondOne; }

public static void main(String[] args) {
 PairObj firstPair = new PairObj("Adam", "Eve");
 PairObj anotherPair = new PairObj("17. May", 1905);
 Object obj = firstPair.getFirst();
 if (obj instanceof String) {// Is the object of the right type?
 String str = (String) obj;// Type conversion to the subclass String.
 System.out.println(str.toLowerCase()); // Specific method in String.
 }
}
```

Program output:

```
adam
```

15



The class `PairObj` can be used to create a pair with arbitrary objects, also with primitive values as these will be encapsulated in corresponding wrapper objects:

```
PairObj firstPair = new PairObj("Adam", "Eve"); // (String, String)
PairObj anotherPair = new PairObj("17. May", 1905); // (String, Integer)
```

If we want to fetch an object from a pair, we get this object via an `Object` reference:

```
Object obj = firstPair.getFirst();
```

If we want to use type-specific properties or behaviour of the fetched object, we must type convert the `Object` reference. In addition we must make sure that the object referred to by the `Object` reference is of the right type, otherwise we risk a `ClassCastException` at runtime.

```
if (obj instanceof String) {// Is the object of the right type?
 String str = (String) obj;// Type conversion to the subclass String.
 System.out.println(str.toLowerCase()); // Specific method in String.
}
```

As we can see from the above example, this approach places certain demands on how to use the class `PairObj` to create and maintain pairs of objects. For instance, it is the responsibility of the programmer to ensure that the objects being paired are of the same type. Implementing classes for pairing specific types of objects is not a good solution. For one thing, it is not certain we know in advance what types of objects will be paired. *Generic types* offer a better solution for such problems, where we can define only one class and specify specific reference types we wish the class to use each time we create an object of this class.

## Declaring generic classes

Program 15.2 declares a *generic class* that can be used to create pairs of objects where *both* objects have the same type. The compiler will enforce proper types of the objects in a pair, without the program having to take any extra actions to ensure their type.

### PROGRAM 15.2 Generic class `Pair<T>`

```
class Pair<T> implements PairRelationship<T> { // (1)
 private T first;
 private T second;
 Pair () { }
 Pair (T first, T second) {
 this.first = first;
 this.second = second;
 }
 public T getFirst() { return first; }
 public T getSecond() { return second; }
 public void setFirst(T firstOne) { first = firstOne; }
 public void setSecond(T secondOne) { second = secondOne; }
 public String toString() {
 return "(" + first.toString() + "," + second.toString() + ")";
 }
}
```

15



A generic class specifies one or more *formal type parameters*, enclosed by the characters < and > right after the class name, as shown in (1). The generic class in Program 15.2 has only *one* formal type parameter, namely `T`. If a generic class has several formal type parameters, these are specified as a comma-separated list, `<T1, T2, ..., Tn>`.

In defining a generic class, we can take as starting point a class where the `Object` type is utilized to generalize the use of the class. In the generic class `Pair<T>` we have used `T` in all the places where the type `Object` was used in the definition of the class `PairObj`. We see from the definition of the class `Pair<T>` that the reference type `T` is used like a reference type in the class body: as a field type, as a return type and a parameter type in the methods. Which reference type `T` actually represents is not known in the generic class `Pair<T>`. The references `first` and `second` have the type `T`, and can therefore only be used to call methods inherited from the `Object` class, as these methods are inherited by all objects, regardless of their object type. One such example is the call to the `toString()` method in (2).

Note that the formal type parameters are *not* specified after the class name in the constructor declaration.

It is quite common to use one-letter names for formal type parameters; a convention that we will follow in this book.

## Using generic classes

We can declare references and create objects of generic classes, and call methods on these objects, in much the same way as we have done before for non-generic classes.

When we use a generic class, we specify the *actual type parameters* that replace the formal type parameters in the class definition. For example, `Pair<String>` introduces a new reference type during compilation, that is to say, pairs that only allow `String` objects, where the formal type parameter `T` is replaced by the actual type parameter `String`. Similarly we can specify a reference type for pairs of `Integer` objects, namely `Pair<Integer>`, where the formal type parameter `T` is replaced by the actual type parameter `Integer`. `Pair<String>` and `Pair<Integer>` are called *parameterized types*. The compiler verifies that parameterized types are used correctly in the source code, i.e. their usage does not cause a problem at runtime. Actual type parameters in a parameterized type are specified after the class name, in the same way as formal type parameters in a generic class definition: enclosed by the characters `<` and `>`, and separated by a comma if necessary.

15



Since there is *only one* implementation of a generic class in Java and values of primitive types have different sizes, it would require that different code is generated for each primitive type that is used as an actual type parameter. For this reason, primitive data types are *not* allowed as actual type parameters.

In the source code parameterized types are used pretty much like other non-parameterized types. Program 15.3 shows how parameterized types are used in reference declarations and in constructor calls. The compiler uses the declaration of the reference `strPair` in (1) to check that the reference `strPair` is only used to handle pairs with `String` objects, as in (4) and (5). Specification of the parameterized type in the constructor call in (1) results in a pair that is guaranteed to hold objects of the class `String` at runtime. Illegal actual type parameter values are reported by the compiler, as in (2). Analogously, the compiler will check the actual type parameters for creation of a pair of type `Pair<Integer>` in (3), where each element has the type `Integer`.

In (4) we see that the types `Pair<Integer>` and `Pair<String>` are different, as we would expect.

### PROGRAM 15.3 Parameterized types

```
public class ParametrizedTypes {

 public static void main(String[] args) {
 Pair<String> strPair = new Pair<String>("Adam", "Eve"); // (1)
 // Pair<String> mixPair = new Pair<String>("17. May", 1905); // (2) Error!
 Pair<Integer> intPair = new Pair<Integer>(2005, 2010); // (3)
 // strPair = intPair; // (4) Compile-time error!
 Pair<String> tempPair = strPair; // (5) OK

 strPair.setFirst("Ole"); // (6) OK. Only String accepted.
 // intPair.setSecond("Maria"); // (7) Compile-time error!
 String name = strPair.getSecond().toLowerCase(); // (8) "eve"
 System.out.println(name);
 }
}
```

```
}
```

Program output:

```
eve
```

We can call instance methods of a generic class in the usual way, as in (6), (7) and (8). Note that we cannot put an object of the wrong type in a pair, as shown in (7). If we wish to call specific methods on an object fetched from a given pair, we do not need to check the type of the object and type convert its reference value, as in (8). The compiler does the job if we use parameterized types. The result is less code (compare with the `main()` method in Program 15.1) and fewer runtime errors.



## Generic interfaces

We can also declare *generic interfaces*. These can then be customized by supplying specific types as actual type parameters. Specification of formal type parameters in a generic interface is the same as in a generic class. The code below shows a generic interface that all pairs must implement:

```
interface PairRelationship<T> {
 T getFirst();
 T getSecond();
 void setFirst(T firstOne);
 void setSecond(T secondOne);
}
```

A generic interface can be implemented by a generic (or a non-generic) class:

```
class Pair<T> implements PairRelationship<T> {
 // same as before
}
```

We can parameterize a generic interface as we did with a generic class. The code in (9) below declares a reference, `oneStrPair`, that has the type `PairRelationship<String>`, and assigns it the reference value of a pair that has the type `Pair<String>`. The assignment is legal, since the parameterized type `Pair<String>` is a subtype of the parameterized type `PairRelationship<String>`:

```
PairRelationship<String> oneStrPair = new Pair<String>("Eve", "Adam"); // (9)
```

The Java standard library contains many generic interfaces. For example, the interface `java.lang.Comparable`, that is used to compare objects, has the following declaration:

```
public interface Comparable<T> {
 int compareTo(T obj);
}
```

A class, whose objects have a natural order, can implement the `Comparable<T>` interface:

```
class Gizmo implements Comparable<Gizmo> {
 public int compareTo(Gizmo obj) { /* Implementation */ }
 // ...
```

}

Note that we have parameterized the generic interface Comparable<T> with Gizmo, since gizmos are what the method compareTo() will compare in the class Gizmo.

## Handling of generic types at compile-time

Generic types comprise generic classes and generic interfaces.

The generic class Pair<T> is compiled and represented by the class Pair, in other words, only one class (Pair) exists that has the *class name* of the generic class (Pair<T>), and the compiler generates only one *class file* (Pair.class) with the Java byte code.

It is important to understand that a parameterized type, for example Pair<String>, is *not* a *class*. Parameterized types are used by the compiler to check that objects created are used correctly in the program. The JVM is oblivious about the use of generic types. It does not differentiate between Pair<String> and Pair<Integer>, and only knows about the class Pair.

The way generic types are implemented in Java has some consequences, since all usage of parameterized types for a generic type are checked against one and the same class. This limits both what properties and what operations a generic class can have. For example, a generic class with a formal type parameter T cannot use T as a type for its static fields or in its static methods. Since only one class represents all parameterizations of a generic class, and only one occurrence of a static member can exist in a class, it is impossible to say which concrete type T should be. Neither can we create objects of a formal type parameter T in a generic class, since it is impossible to say which concrete type such an object should have.

It is possible to just use a generic class by its name only, like a non-generic class, without specifying actual type parameters for its usage. The compiler will issue an *unchecked warning* if such a use can be a potential problem at runtime. Such usage is permitted for compatibility with legacy code, but is strongly advised against when writing new code.

Section 15.8 provides a discussion of some other aspects concerning the usage of generic types in Java.

### BEST PRACTICES

If a class will act as a “container” of different types of objects, requiring the same operations, consider making the class generic.



## BEST PRACTICES

Don't use the `Object` type indiscriminately as an actual type parameter for generic classes just to pacify the compiler. Used properly, generic types provide many benefits, such as early detection of errors at compile time rather than at runtime.

## 15.4 Collections

15



Collections are a part of the `java.util` package in Java. A collection provides a *contract* in the form of an *interface* that a client can use to interact with the collection. After a collection has been created, the client uses the methods of the interface and references of the interface type to handle the collection. The clients need not know which *concrete* implementation is being used for the collection. The implementation of the collection can change without consequences for the client.

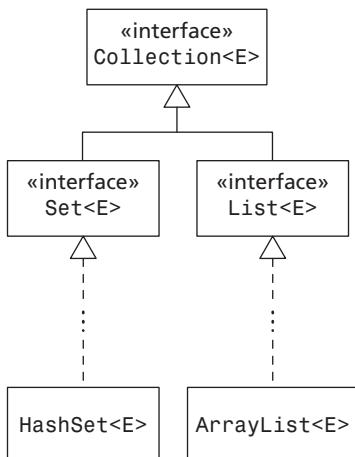
Collections in the `java.util` package are implemented as generic types. This means that we must specify what type of objects a collection will contain when we create it. This also implies that the collections we create are customised for particular types of objects. The compiler will make sure that each collection is handled correctly, depending on the type of objects we put in the collection.

### Superinterface `Collection<E>`

A partial inheritance hierarchy for the main collection interfaces and classes is shown in Figure 15.1. The type parameter `E` stands for the element type, i.e. the type of the objects in the collection. *Basic operations* that can be executed on all collections are defined by the `Collection<E>` interface. The subinterfaces `Set<E>` and `List<E>` extend the `Collection<E>` superinterface to incorporate operations for *sets* and *lists*, respectively. The generic classes `HashSet<E>` and `ArrayList<E>` are concrete implementations of *sets* (Section 15.6) and *lists* (Section 15.5), respectively.

Table 15.3 shows the basic operations that the interface `Collection<E>` provides for all collections. The operations are the usual ones that are executed on collections, for example adding an element, removing an element and checking whether the collection contains a certain element. Note that some of the formal parameters for the basic operations in Table 15.3 are of the type `Object`, for example the parameter of the method `contains()`, but others have the same parameter type as the type parameter `E` of the interface, for example the method `add()`. This is how Sun MicroSystems decided to define the basic operations for collections in the Java standard library.

**FIGURE 15.1** Selected collections and interfaces from the `java.util` package



15



**TABLE 15.3** Basic operations from the `Collection<E>` interface

| java.util.Collection                          |                                                                                                                                        |
|-----------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| <code>int size()</code>                       | Returns the number of elements in the collection.                                                                                      |
| <code>boolean isEmpty()</code>                | Determines whether the collection is empty.                                                                                            |
| <code>boolean contains(Object element)</code> | Determines whether the specified <code>element</code> is in the collection.                                                            |
| <code>boolean add(E element)</code>           | <i>Insertion:</i> tries to insert the specified <code>element</code> into the collection, and returns <code>true</code> if successful. |
| <code>boolean remove(Object element)</code>   | <i>Deletion:</i> tries to delete the specified <code>element</code> from the collection, and returns <code>true</code> if successful.  |
| <code>Iterator&lt;E&gt; iterator()</code>     | Returns an iterator that can be used to traverse the collection.                                                                       |

Table 15.4 shows the *bulk operations* that the interface `Collection<E>` specifies. A bulk operation is executed on whole collections. Mention of “the current collection” in Table 15.4 refers to the collection on which a method is invoked. Note that a bulk operation can change the current collection. Bulk operations are discussed in detail in connection with sets in Section 15.6.

Several bulk methods in Table 15.4 have the type `Collection<?>` as the type of the formal parameter. The type `Collection<?>` denotes a collection of unknown type, meaning that the method call can pass a `Collection` of any type. The formal parameter `c` of the method `addAll()` has the type `Collection<? extends E>`. This type denotes all `Collections` where the element type is a subtype of the type parameter `E`. Such type specifications make it possible to perform bulk operations between different collections, as long as the elements in these collections are compatible with regard to the bulk operation. Such parameterized



types are further discussed in Section 15.8.

**TABLE 15.4 Bulk operations from the `Collection<E>` interface**

| <code>java.util.Collection</code>                            |                                                                                                                                                                                                                                                |
|--------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>boolean containsAll(Collection&lt;?&gt; c)</code>      | <i>Subset</i> : returns <code>true</code> if all the elements in the specified collection <code>c</code> are also in the current collection.                                                                                                   |
| <code>boolean addAll(Collection&lt;? extends E&gt; c)</code> | <i>Union</i> : adds all the elements from the specified collection <code>c</code> to the current collection, and returns <code>true</code> if the current collection was modified.                                                             |
| <code>boolean retainAll(Collection&lt;?&gt; c)</code>        | <i>Intersection</i> : only elements from the specified collection <code>c</code> are retained in the current collection, the rest being removed from the current collection. Returns <code>true</code> if the current collection was modified. |
| <code>boolean removeAll(Collection&lt;?&gt; c)</code>        | <i>Difference</i> : elements from the specified collection <code>c</code> are removed from the current collection. Returns <code>true</code> if the current collection was modified.                                                           |
| <code>void clear()</code>                                    | Removes all elements from the current collection.                                                                                                                                                                                              |

## Traversing a collection using an iterator

An important operation on a collection is to *traverse* the collection, that is to say access each element of the collection one by one in a systematic way. Collections from the Java standard library provide an *iterator* for this purpose. The iterator implements a Java interface that defines the methods for traversing a collection.

Table 15.5 shows the interface `Iterator<E>` that can be used to traverse a collection. The iterator for a collection is obtained by calling the method `iterator()` on the collection, defined in the `Collection<E>` interface. In order to access the elements one by one we use a loop in which we use the operations from Table 15.5. As the loop condition we call the method `hasNext()`, in order to find out whether there are still more elements left to access in the collection. The current element is obtained by calling the method `next()`. The current element can also be deleted from the collection by calling the method `remove()`. Note that these methods are called on the *iterator*, and not on the collection:

```
// Create a list of strings.
Collection<String> collection = new ArrayList<String>();
collection.add("9"); // Add elements.
collection.add("1");
collection.add("1");
```

```

Iterator<String> iter = collection.iterator(); // Get an iterator.
while (iter.hasNext()) { // More elements in the collection?
 System.out.print(iter.next()); // Print the current element.
}

```

If the method `hasNext()` returns true, we can call the method `next()` to return the next element in the collection, and to increment the iterator. In other words, it is not necessary to increment the iterator explicitly, this is taken care of by the `next()` method.

### BEST PRACTICES

The two methods `hasNext()` and `next()` of an `Iterator` should be used *in sync*, otherwise we risk a runtime error during the traversal of the collection supported by the iterator.

15



**TABLE 15.5 Operations in the `Iterator<E>` interface**

| java.util.Iterator             |                                                                                                                                                                       |
|--------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>boolean hasNext()</code> | Returns true if the underlying collection has elements left to iterate over.                                                                                          |
| <code>E next()</code>          | Returns the next element in the underlying collection and increments the iterator. If there are no more elements left, throws a <code>NoSuchElementException</code> . |
| <code>void remove()</code>     | Deletes the current element from the underlying collection. This method can be called only once after each call of the <code>next()</code> method.                    |

### Traversing a collection using the `for(:)` loop

Earlier we have seen how to traverse an array using a `for(:)` loop (see Section 6.6). A collection can also be traversed with a `for(:)` loop, if it implements the `java.lang.Iterable<E>` interface:

```

interface Iterable<E> {
 Iterator<E> iterator();
}

```

The method `iterator()` returns an iterator that implement the interface `Iterator<E>` (see Table 15.5). The interface `Iterable<E>` actually says that if a collection has an iterator, we can traverse the collection with a `for(:)` loop. The interface `Collection<E>` extends the interface `Iterable<E>`, so that all collections that implement the `Collection<E>` interface can be traversed using the `for(:)` loop. The collection with strings, that was created above, can be traversed with the following code:

```

for (String str : collection)
 System.out.print(str);

```

The element variable `str` has the type `String`, which is the element type of the collection. For each element in the collection the loop body is executed. The current element can thus be accessed by the element variable during the execution of the loop body. Behind the scene, an iterator is actually used to traverse the collection, but we do not use it explicitly in the `for(:)` loop.

### BEST PRACTICES

Make sure that the type of the local loop variable in a `for(:)` loop is assignment compatible with the element type of the collection that is being traversed.

## Default string representation of a collection

We can use the `System.out.println()` method to print all elements of a collection:

```
System.out.println(collection); // Output: [9, 1, 1]
```

A collection defines its own `toString()` method that creates a default string representation of the collection. For each element, the element's `toString()` method is called to create a string representation of the element. The default string representation is as follows, where `elementi` is the string representation of an element in the collection:

```
[element0, element1, ..., elementn-1]
```

## 15.5 Lists: `ArrayList<E>`

### Subinterface `List<E>`

A *list* (a.k.a. *sequence*) is a collection where each element has a *position*. A list also allows duplicates. The `List<E>` interface extends the `Collection<E>` superinterface for lists. The most important extension concerns operations that allow positional access to the elements in the list using an index, analogous to accessing the elements in an array. A selection of list operations is shown in Table 15.6.

The class `ArrayList<E>` in the `java.util` package implements lists. It satisfies the `List<E>` interface. We can execute all operations in the `List<E>` interface, including those that are inherited from the `Collection<E>` interface. Here we will confine ourselves to the list operations in Table 15.6. The index of the first element in a list is 0, the same as for an array. An array has the field `length` that indicates the number of elements in it, while in a list the method `size()` (from the `Collection<E>` interface) returns this information. By specifying an index we can *get*, *set* or *remove* the element at the position given by the index in the list. We can add an element at the position given by the index, shifting the elements towards the end of the list, if necessary (the method `add()` in Table 15.6). We can find the index of the first position where an object is stored in a list (the method `indexOf()` in Table 15.6). Remember that a list can have duplicates. Section 15.6 shows a simple way of removing duplicates from a list.





There are many ways to implement a list. The implementation in the class `ArrayList<E>` is known as *resizable arrays*. In Section 16.1 (fix Xref) we present an implementation that is based on *linked lists*.

**TABLE 15.6** Selected list operations from the `List<E>` interface

| <code>java.util.List</code>                 |                                                                                                                                                                                                           |
|---------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>E get(int index)</code>               | Returns the element at the specified <code>index</code> .                                                                                                                                                 |
| <code>E set(int index, E element)</code>    | Replaces the element at the specified <code>index</code> with the specified <code>element</code> . Returns the element that was replaced.                                                                 |
| <code>E remove(int index)</code>            | Removes and returns the element at the specified <code>index</code> . Elements in the list are shifted if necessary.                                                                                      |
| <code>void add(int index, E element)</code> | Inserts the specified element at the specified <code>index</code> . Elements in the list are shifted if necessary.                                                                                        |
| <code>int indexOf(Object element)</code>    | Returns the index of the first occurrence of an element in the list that is equal to the specified <code>element</code> according to the <code>equals()</code> method, or -1 if there is no such element. |

## Using lists

Program 15.4 illustrates the use of some of the list operations. The program reads the arguments from the command line into a list (`wordList`), and *censors* certain words from this list. Words that should be censored are stored in a separate list (`censoredWords`).

An empty list of strings is created in (1):

```
List<String> wordList = new ArrayList<String>();
```

Arguments from the command line are inserted in the `wordList` using the `add()` method. In (2) another list (`censoredWords`) is constructed with words that should be censored.

In (3), for each word in the `wordList`, we try to find its index in the `censoredWords` list. If we find the index, the word must be censored in the `wordList`. But this requires knowing its index in the `wordList`. We find that by calling the method `indexOf()` on the `wordList`. Given this index we can replace the word in the `wordList` by calling the `set()` method.

Remember to include an `import` statement at the beginning of the source code file if classes from the `java.util` package are used without their fully qualified class name.

## BEST PRACTICES

In many cases, choosing a list is just as effective as using an array. If the length is not fixed, then a list is certainly a better choice than an array.

### PROGRAM 15.4 Lists

```
import java.util.ArrayList;
import java.util.List;

public class ListClient {

 static final String CENSORED = "CENSORED";

 public static void main(String args[]) {
 // (1) Read words from the command line into a word list:
 List<String> wordList = new ArrayList<String>();
 for (int i = 0; i < args.length; i++)
 wordList.add(args[i]);
 System.out.println("Original word list: " + wordList);

 // (2) Create a list with words that are censored:
 List<String> censoredWords = new ArrayList<String>();
 censoredWords.add("fun");
 censoredWords.add("cool");
 censoredWords.add("easy");

 // (3) Censor the word list:
 for (String element : wordList) {
 if (censoredWords.indexOf(element) != -1) {
 int indexInWordList = wordList.indexOf(element);
 wordList.set(indexInWordList, CENSORED);
 }
 }

 // (4) Print the censored list:
 System.out.println("Censored word list: " + wordList);
 }
}
```

Running the program:

```
>java ListClient Java is cool fun and easy
Original word list: [Java, is, cool, fun, and, easy]
Censored word list: [Java, is, CENSORED, CENSORED, and, CENSORED]
```



## 15.6 Sets: HashSet<E>

### Subinterface Set<E>

The interface `Set<E>` models the mathematical concept *set* that allows such operations as *union*, *intersection* and *difference* in set theory. Unlike to a list, a set does not permit duplicates. Even if we insert the same value several times, only one occurrence of the value is stored in the set. There is no order on the elements in a set.

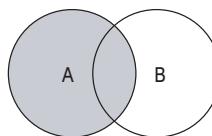
The `Set<E>` interface introduces no new methods beyond those already found in the `Collection<E>` interface, but specializes the existing ones for sets. The `HashSet<E>` class implements the `Set<E>` interface, and thereby also the `Collection<E>` interface (see Figure 15.1 on page 476). We will use the class `HashSet<E>` to work with sets.

15



The bulk operations from Table 15.4 are particularly interesting for sets, because these define the basic operations in set theory. Figure 15.2b, Figure 15.2c and Figure 15.2d illustrates application of set operations on sets A and B from Figure 15.2a. It is easy to forget that these operations are not exclusively for sets, but are defined for *all* collections (for example `ArrayList<E>`) that implement the `Collection<E>` interface.

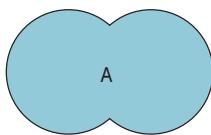
FIGURE 15.2 Set Theory



A = [kiss, Sivle, madonna, aha, abba]

B = [TLC, wham, madonna, abba]

(a)



A.addAll(B)

After execution:

A = [kiss, TLC, Sivle, wham, aha, madonna, abba]

(b) Union

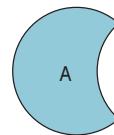


A.retainAll(B)

After execution:

A = [madonna, abba]

(c) Intersection



A.removeAll(B)

After execution:

A = [kiss, Sivle, aha]

(d) Difference

### Using sets

Program 15.5 illustrates the use of set operations from Figure 15.2. At (1) we create two empty sets with the element type `String` for two concerts, A and B. We create an empty set for concert A in the usual way by instantiating the class `HashSet<E>` with the type argument `String`:



```
Set<String> concertA = new HashSet<String>();
```

At (2) in Program 15.5 we add the different artists to `concertA` by calling the `add()` method, for example:

```
concertA.add("aha");
```

If a call to the `add()` method results in an element being added to the set, the method returns the boolean value `true`. The return value `false` will indicate that the element already exists in the set. This is illustrated at (3) in Program 15.5, where we try to register the artist "TLC" one more time.

Given that the sets `concertA` and `concertB` represent artists for two different concerts, we can print some statistics with the help of the bulk operations. We can for example find out whether all artists who performed in concert B, also performed in concert A. This is equivalent to determining whether concert B is a *subset* of concert A, that is to say if all elements in concert B are also present in concert A (compare with Table 15.4). The subset operation is illustrated at (4) in Program 15.5. The output from the program shows that concert B is not a subset of concert A.

The union operation illustrated in Figure 15.2b is executed at (5) in Program 15.5. By creating a *union* of the two sets, we can find all the artists who performed in the two concerts. The method `addAll()` corresponds to the union operation in set theory. Union, intersection and difference operations are all *destructive operations*, that is to say they can modify the set on which they are called. We perform these operations in Program 15.5 on a copy, so that the original set remains unchanged. The code below shows how we can create a copy of a set by calling the right constructor in the class `HashSet<E>` with the type argument `String`:

```
Set<String> allArtists = new HashSet<String>(concertA);
```

This does *not* imply that copies are made of the elements in the `concertA`, only that both sets have *references* to the *same* objects.

To find all artists that performed in both concerts corresponds to *intersection* between the two sets (Figure 15.2c). The method `retainAll()` corresponds to the intersection operation, and it is applied at (6) in Program 15.5.

If we are interested in finding artists that performed only in one concert and not in the other, we can use the *difference* operation (Figure 15.2d). The `removeAll()` method corresponds to the difference operation, and is applied at (7) and (8) in Program 15.5.

We showed above how we can make a copy of a set by calling the right constructor. We can also create a set from a list (and vice versa) by calling a suitable constructor. The code below creates a set (`wordSet`) from a list of strings (`wordList`), and then a new list from this set, thereby removing duplicates from the original list:

```
Set<String> wordSet = new HashSet<String>(wordList);
wordList = new ArrayList<String>(wordSet);
```



## BEST PRACTICES

If the collection will not have duplicates and positional access is not important, then using a set is the right choice.

### PROGRAM 15.5 Set Theory

```
import java.util.HashSet;
import java.util.Set;

public class SetClient {
 public static void main(String args[]) {
 // (1) Create two sets for concerts:
 Set<String> concertA = new HashSet<String>();
 Set<String> concertB = new HashSet<String>();
 System.out.println("(1) Created an empty concert A: " + concertA);
 System.out.println("(1) Created an empty concert B: " + concertB);

 // (2) Add artists to the concerts:
 concertA.add("aha"); concertA.add("madonna");
 concertA.add("abba"); concertA.add("kiss");
 concertA.add("Sivle");
 System.out.println("(2) Artists in concert A: " + concertA);

 concertB.add("TLC"); concertB.add("madonna");
 concertB.add("abba"); concertB.add("wham");
 System.out.println("(2) Artists in concert B: " + concertB);

 // (3) Duplicates are not allowed in sets:
 String artist = "TLC";
 if (concertB.add(artist))
 System.out.println("(3) Artist " + artist +
 " registered for concert B.");
 else
 System.out.println("(3) Artist " + artist +
 " already registered for concert B.");

 // (4) Subset: determine whether all artists that performed
 // in concert B, also performed in concert A:
 if (concertA.containsAll(concertB))
 System.out.println("(4) All artists in concert B performed" +
 " in concert A.");
 else
 System.out.println("(4) All artists in concert B did not perform" +
 " in concert A.");
 }
}
```



```

// (5) Union: find all artists:
Set<String> allArtists = new HashSet<String>(concertA);
allArtists.addAll(concertB);
System.out.println("(5) All artists: " + allArtists);

// (6) Intersection: find artists that performed in both concerts:
Set<String> bothConcerts = new HashSet<String>(concertA);
bothConcerts.retainAll(concertB);
System.out.println("(6) Artists that performed in both concerts: " +
 bothConcerts);

// (7) Difference: find artists that performed in concert A only:
Set<String> onlyConcertA = new HashSet<String>(concertA);
onlyConcertA.removeAll(concertB);
System.out.println("(7) Artists that performed in concert A only: " +
 onlyConcertA);

// (8) Difference: find artists that performed in concert B only:
Set<String> onlyConcertB = new HashSet<String>(concertB);
onlyConcertB.removeAll(concertA);
System.out.println("(8) Artists that performed in concert B only: " +
 onlyConcertB);
}
}

```

Program output:

```

(1) Created an empty concert A: []
(1) Created an empty concert B: []
(2) Artists in concert A: [kiss, abba, madonna, Sivle, aha]
(2) Artists in concert B: [abba, madonna, wham, TLC]
(3) Artist TLC already registered for concert B.
(4) All artists in concert B did not perform in concert A.
(5) All artists: [kiss, abba, madonna, TLC, wham, aha, Sivle]
(6) Artists that performed in both concerts: [abba, madonna]
(7) Artists that performed in concert A only: [kiss, aha, Sivle]
(8) Artists that performed in concert B only: [TLC, wham]

```

## 15.7 Maps: `HashMap<K, V>`

A *map* (a.k.a. *hash table*) is used to store *entries* (a.k.a. *bindings*, *mappings*). Each entry is a pair of objects, where the first object (called the *key*) is associated with the second object (called the *value*). A telephone list is an example of a map, where each entry associates a telephone number (key) with a name (value). Another example of a map is entries that associate an employee with the number of hours the employee has worked in a week.

The relation between keys and values in a map is a *many-to-one* relation, that is to say, many different keys can have the same value, but different values cannot have the same



key. This implies that the keys in a map are *unique*. Table 15.7 shows a schematic representation of a map. How it is implemented, is not important for our purpose. Each row in the table corresponds to an entry in the map, associating a word (i.e. a key) to its frequency (i.e. a value) in a given text. We see from Table 15.7 that a key occurs only once in an entry, while a value can be associated with several keys.

**TABLE 15.7 Schematic illustration of a map**

| Word (key) | Frequency (value) |
|------------|-------------------|
| to         | 2                 |
| be         | 4                 |
| or         | 1                 |
| not        | 1                 |
| what       | 1                 |
| will       | 2                 |

## Hashing

Storage and retrieval of entries in a map requires that it is possible to identify a key in an entry with the help of an integer value. This integer value, called the *hash value*, is employed both in order to store the key (and its value) and also to retrieve the entry of a key in the map. This technique is called *hashing*. In Java the method `hashCode()` is used to compute the hash value of an object. This method is defined in the `Object` class, but must be overridden if the objects of a class are to be used in a map. The hash value of the key must not be confused with the value associated with the key in an entry that is stored in the map.

The hash value of an object must satisfy the following conditions:

- The hash code must always be the same for an object as long as its state has not changed.
- Two objects that are equal according to the `equals()` method must have the same hash value.

The method `hashCode()` in the `Object` class does not satisfy the above conditions. It returns the address in the memory where the object is stored, even if the object state has changed. Two `String` objects with the same state will get different hash values, since they will be stored at different addresses. The class `String` therefore overrides both the `hashCode()` and the `equals()` method from the `Object` class, so that the conditions above are fulfilled. The wrapper classes do the same for all primitive data types. For example, the `hashCode()` method for integer wrapper classes will return the integer value encapsulated in the wrapper object as hash value.

Hashing is a large topic. Here we will make do with some simple rules that can be used to implement the `hashCode()` method, so that the above mentioned conditions are fulfilled.

## PROGRAM 15.6 Hashing

```
/**
 * The Point3D_V1 class represents a point in space,
 * but it overrides neither the equals() method nor the hashCode() method.
 */
class Point3D_V1 { // (1)
 int x;
 int y;
 int z;

 Point3D_V1(int x, int y, int z) {
 this.x = x;
 this.y = y;
 this.z = z;
 }

 public String toString() { return "[" +x+ "," +y+ "," +z+"]"; }
}

/**
 * The Point3D class represents a point in space,
 * and it overrides both the equals() method and the hashCode() method.
 */
class Point3D { // (2)
 int x;
 int y;
 int z;

 Point3D(int x, int y, int z) {
 this.x = x;
 this.y = y;
 this.z = z;
 }

 public String toString() { return "[" +x+ "," +y+ "," +z+"]"; }

 /** Two points are equal if they have the same x, y and z coordinates. */
 public boolean equals(Object obj) { // (3)
 if (this == obj) return true;
 if (!(obj instanceof Point3D)) return false;
 Point3D p2 = (Point3D) obj;
 return this.x == p2.x && this.y == p2.y && this.z == p2.z;
 }

 /** The hash value is computed based on the coordinate values. */
 public int hashCode() { // (4)
 int hashValue = 11;
 hashValue = 31 * hashValue + x;
```





```

 hashValue = 31 * hashValue + y;
 hashValue = 31 * hashValue + z;
 return hashValue;
 }
}

public class Hashing { // (5)
 public static void main(String[] args) {

 System.out.println("When equals() and hashCode() methods are" +
 " not overridden:");
 Point3D_V1 p1 = new Point3D_V1(1, 2, 3);
 Point3D_V1 p2 = new Point3D_V1(1, 2, 3);
 System.out.println("Point3D_V1 p1" + p1 + ":" + p1.hashCode());
 System.out.println("Point3D_V1 p2" + p2 + ":" + p2.hashCode());
 System.out.println("p1.hashCode() == p2.hashCode(): " +
 (p1.hashCode() == p2.hashCode()));
 System.out.println("p1.equals(p2): " + p1.equals(p2));

 System.out.println();
 System.out.println("When equals() and hashCode() methods are" +
 " overridden:");
 Point3D pp1 = new Point3D(1, 2, 3);
 Point3D pp2 = new Point3D(1, 2, 3);
 System.out.println("Point3D pp1" + pp1 + ":" + pp1.hashCode());
 System.out.println("Point3D pp1" + pp2 + ":" + pp2.hashCode());
 System.out.println("pp1.hashCode() == pp2.hashCode(): " +
 (pp1.hashCode() == pp2.hashCode()));
 System.out.println("pp1.equals(pp2): " + pp1.equals(pp2));
 }
}

```

Program output:

When equals() and hashCode() methods are not overridden:

```

Point3D_V1 p1[1,2,3]: 29115481
Point3D_V1 p2[1,2,3]: 19621457
p1.hashCode() == p2.hashCode(): false
p1.equals(p2): false

```

When equals() and hashCode() methods are overridden:

```

Point3D pp1[1,2,3]: 328727
Point3D pp1[1,2,3]: 328727
pp1.hashCode() == pp2.hashCode(): true
pp1.equals(pp2): true

```

---

Program 15.6 consist of three classes. The class `Point3D_V1` in (1) overrides neither the `equals()` nor the `hashCode()` method. Output from the program shows that objects of the class `Point3D_V1` have different hash values and are not equal, even if they have the same state (see references `p1` and `p2` in the class `Hashing`).

The class `Point3D` in (2) overrides the `equals()` method at (3). Two `Point3D` objects are equal if they have the same state, i.e. corresponding fields in the two objects have the same values. The class `Point3D` also overrides the `hashCode()` method at (4). The hash value is calculated according to the following formula:

```
hashValue = 11 * 313 + x * 312 + y * 311 + z
```

where the values of the fields `x`, `y` and `z` are included. This formula can be generalised with regard to the number of fields in the object. We only include the fields that are used in the `equals()` method. This helps to ensure that the second condition above is satisfied. If a field refers to an object, then the hash value of this object is used in the calculation. Output from the program shows that objects of the class `Point3D` have the same hash value and are equal when they have the same state (see the references `pp1` and `pp2` in the class `Hashing`).

### BEST PRACTICES

Be sure to override *both* the `equals()` and the `hashCode()` method, if the objects of your class will be used in sorting and searching. Consider also implementing the `Comparable<E>` interface for your class.

15



## The `Map<K, V>` interface

The functionality of maps is specified by the interface `Map<K,V>` in the `java.util` package. The class `HashMap<K,V>` is a concrete implementation of the `Map<K,V>` interface (Figure 15.3). The interface `Map<K,V>` and the class `HashMap<K,V>` are examples of generic types with two type parameters. The type parameter `K` denotes the type of the keys and the type parameter `V` denotes the type of the values in the map.

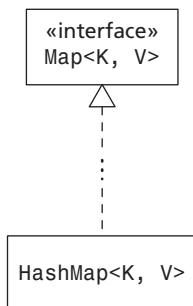
We can create an empty map with strings as keys and integers as values:

```
Map<String, Integer> wordMap = new HashMap<String, Integer>();
```

Note that the reference `wordMap` has the reference type `Map<String, Integer>`, and the reference refers to an object of the parameterized type `HashMap<String, Integer>` (a map). This is completely legal, as the class `HashMap<K,V>` implements the interface `Map<K,V>`.

FIGURE 15.3

Map interface and its implementation



A selection of basic operations in the interface `Map<K,V>` are given in Table 15.7. The method `put()` creates an entry and inserts it into the map:

15



```
wordMap.put("be", 3);
```

The integer 3 is automatically encapsulated in an `Integer` object. It is important to note that the `put()` method will *overwrite* any previous entry with the same key, and return the value from the previous entry. If we execute the following statement:

```
Integer previousFrequency = wordMap.put("be", 4); // returns 3
```

the `put()` method will insert the new entry `<"be", new Integer(4)>` and return a reference to an `Integer` object with the integer value 3, that is the value from the previous entry for the key "be".

We can retrieve the value of a key by calling the method `get()`:

```
Integer frequency = wordMap.get("be"); // returns 4
```

The entry of a key can be deleted by calling the `remove()` method that returns the value in the entry. We can also call *membership methods* to determine whether a key is registered, and whether a value occurs in one or more entries:

```
boolean keyFound = wordMap.containsKey("be"); // true
boolean valueFound = wordMap.containsValue(2008); // false
```

TABLE 15.8

Basic operations from the `Map<K,V>` interface

| java.util.Map                      |                                                                                                                                                                                                                     |
|------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>int size()</code>            | Returns the number of entries in the map.                                                                                                                                                                           |
| <code>boolean isEmpty()</code>     | Determines whether the map is empty, i.e. has any entries.                                                                                                                                                          |
| <code>V put(K key, V value)</code> | Associates the key to the value and stores the entry in the map. If the key already has a entry from before, it returns the old value from this entry, otherwise it returns the reference value <code>null</code> . |



| java.util.Map                       |                                                                                                                                                                                                                                        |
|-------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| V get(Object key)                   | Returns the value from the entry which this key has, if it was registered in the map, otherwise the method returns the reference value <code>null</code> .                                                                             |
| V remove(Object key)                | Tries to remove the entry of the <code>key</code> if the <code>key</code> was registered in the map. If successful the method returns the value of the <code>key</code> , otherwise it returns the reference value <code>null</code> . |
| boolean containsKey(Object key)     | Determines whether the specified <code>key</code> has an entry in the map.                                                                                                                                                             |
| boolean containsValue(Object value) | Determines whether the specified <code>value</code> is associated with one or more keys in the map.                                                                                                                                    |

## Map views

The interface `Map<K,V>` does not offer an iterator. If we want to access all the entries in a systematic way, we must first create a *view* of the map. A *map view* is a collection that is associated with the underlying map. With the help of this view we can, for example, traverse the underlying map. *Changes made via a view are reflected in the underlying map*. Table 15.9 shows two methods we can use to create different views from a map.

The method `keySet()` creates a view that consists of a *set with all the keys* in the map. If the reference `wordMap` refers to the map in Table 15.7, the following code will create a *key view*:

```
Set<String> keyView = wordMap.keySet(); // [not, to, what, or, will, be]
```

A key view represents the key column in Table 15.7. For the client it is not necessary to know how this set is implemented, only that it satisfies the interface `Set<K,V>`. Since keys are unique, it is appropriate to create a set because a set cannot have duplicates. We can use a `for(:)` loop to traverse this set. As we traverse the set with the keys, we can use the method `get()` on the map to retrieve the value corresponding to each key.

TABLE 15.9 View operations from the `Map<K,V>` interface

| java.util.Map                 |                                                                                                                   |
|-------------------------------|-------------------------------------------------------------------------------------------------------------------|
| public Set<K> keySet()        | Returns the <i>Set view</i> of all the keys in the map.                                                           |
| public Collection<V> values() | Returns the <i>Collection view</i> of all the values in all the entries in the map, which may contain duplicates. |

The method `values()` creates a view that consists of a *collection with all the values* in the map. If the reference `wordMap` refers to the map in Table 15.7, the following code will create a *value view*:

```
Collection<Integer> valueView = wordMap.values(); // [1, 2, 1, 1, 2, 4]
```



A value view represents the value column in Table 15.7. The method `values()` returns a collection that satisfies the interface `Collection<V>`. Since the values are not unique, it makes sense to create a collection that allows duplicates. We can use a `for(:)` loop to traverse this collection with values from the map in the usual way.

## Using maps

Program 15.7 shows a client that uses a map and several views. The client reads text from the command line, creates a map with word frequencies as shown in Table 15.7, and prints some statistics.

### PROGRAM 15.7 Using maps and views

```
import java.util.Collection;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

public class MapClient {

 public static void main(String args[]) {
 // (1) Create an empty map for <String, Integer> entries,
 // representing a word and its frequency.
 Map<String, Integer> wordMap = new HashMap<String, Integer>();

 // (2) Read words from the command line.
 for (int i = 0; i < args.length; i++) {
 // Lookup if the word is already registered.
 Integer numOfTimes = wordMap.get(args[i]);
 if (numOfTimes == null)
 numOfTimes = 1; // Not registered before, i.e. first time.
 else
 // Registered. Frequency is incremented.
 numOfTimes++;
 wordMap.put(args[i], numOfTimes);
 }

 // (3) Print the word map.
 System.out.println("Whole word map: " + wordMap);

 // (4) Print total number of words read.
 Collection<Integer> freqCollection = wordMap.values();
 int totalNumOfWords = 0;
 for (int frequency : freqCollection)
 totalNumOfWords += frequency;
 System.out.println("Total number of words read: " + totalNumOfWords);

 // (5) Print all distinct words.
 Set<String> setOfAllWords = wordMap.keySet();
```

```

System.out.println("All distinct words: " + setOfAllWords);
System.out.println("Number of distinct words: " + wordMap.size());

// (6) Print all duplicated words.
Collection<String> setOfDuplicatedWords = new HashSet<String>();
for (String key : setOfAllWords) {
 int numOfTimes = wordMap.get(key);
 if (numOfTimes != 1)
 setOfDuplicatedWords.add(key);
}
System.out.println("All duplicated words: " + setOfDuplicatedWords);
System.out.println("Number of duplicated words: " +
 setOfDuplicatedWords.size());
}
}

```

Running the program:

```

>java MapClient to be or not to be what will be will be
Whole word map: {not=1, to=2, what=1, or=1, will=2, be=4}
Total number of words read: 11
All distinct words: [not, to, what, or, will, be]
Number of distinct words: 6
All duplicated words: [to, will, be]
Number of duplicated words: 3

```

---

In Program 15.7 we create an empty map for `<String, Integer>` entries at (1). Thereafter, in (2), we read words from the command line and insert them in the map according to the following algorithm [fix pseudocode formatting]:

Repeat while several arguments on the command line:

Do a lookup in the map with the current argument.

If the current argument has frequency equal to 0, i.e. it is not registered:

Let the frequency of the current argument be 1.

otherwise:

Increment the frequency of the current argument.

Insert the current argument with the right frequency into the map.

A lookup in the map is done by calling the `get()` method, that returns the value associated with the key if the key exists, otherwise it returns `null`. Insertion in the map is done by the `put()` method, that overwrites any earlier entry.

At (3) we print the complete map, that is to say all the words registered, together with the corresponding frequency. The string concatenation operator (+) converts the map to its default string representation by calling the overridden `toString()` method for maps. The string representation of keys and values will depend on the `toString()` method of





these objects. The map `wordMap` has the following string representation, as the output from the program shows:

```
{not=1, to=2, what=1, or=1, will=2, be=4}
```

At (4) we are interested in calculating the total number of words that were read. This is obviously equal to the value of the expression `args.length`, but we will calculate it from the map. The total number of words is equal to the sum of all the frequencies, since each frequency represents how many times a word was read. For this purpose we use the method `values()` on the map to create a value view that is a collection with all the frequencies (i.e. values) from the map. We traverse this collection with a `for(:)` loop to sum up the frequencies:

```
Collection<Integer> freqCollection = wordMap.values();
int totalNumOfWords = 0;
for (int frequency : freqCollection)
 totalNumOfWords += frequency;
```

In the loop header above we have used implicit conversions from `Integer` objects (representing frequencies) to `int` values.

At (5) we print all the distinct words in the map, that is to say all the unique words that were read. All distinct words are comprised of the keys that are registered. For this purpose we use the method `keySet()` on the map to create a key view that is a set with all the words (i.e. keys) from the map, and print the set.

How do we find all the words that have duplicates, i.e. all words with frequency greater than 1? At (6) we create a key view that we traverse. In each iteration we do a lookup in the map to find the corresponding frequency and maintain duplicated words in a set [fix pseudocode formatting]:

Create an empty set for duplicated words.

Create a key view as a set of all keys.

Repeat while more elements in the set of all keys:

    Do a lookup in the map with the current key.

    If the frequency is not equal to 1, i.e. a duplicated word:

        Insert word into the set for duplicated words.

### BEST PRACTICES

If you plan to use a map, make sure that the keys provide an appropriate implementation of the `hashCode()` method.



## 15.8 More on generic types

The following method adds the two values in a numerical pair:

```
static double sumPair(Pair<Number> pair) {
 return pair.getFirst().doubleValue() + pair.getSecond().doubleValue();
}
```

We expect to be able to call this method with numerical pairs whose type is a subtype of `Pair<Number>`, for example `Pair<Integer>`. However, the code below results in a compile-time error:

```
double sum = sumPair(new Pair<Integer>(100, 200)); // (1) Compile-time error!
```

The compiler will not accept this method call because the parameterized type `Pair<Integer>` is *not* a subtype of the parameterized type `Pair<Number>`, even though `Integer` is a subtype of `Number`. The array type `Integer[]` is a subtype of the array type `Number[]`, but this type relationship does not hold for parameterized types. If such a relationship was allowed at compile-time, it could lead to problems at runtime, as shown by the following code:

```
Pair<Integer> iPair = new Pair<Integer>(2008, 6); // Integer pair
Pair<Number> nPair = iPair; // Assume this was allowed at compile-time.
nPair.setFirst(25.5); // Set a Double in Integer pair.
Integer i = iPair.getFirst(); // Results in a ClassCastException at runtime!
```

### Specifying subtypes: `<? extends T>`

We must rewrite the method `sumPair()`, so that it is possible to pass pairs with *element type* that is a *subtype* of `Number`. The *wildcard* `?` can be used for this purpose:

```
static double sumPair(Pair<? extends Number> pair) {
 return pair.getFirst().doubleValue() + pair.getSecond().doubleValue();
}
```

The parameterized type `Pair<? extends Number>` stands for all pairs with the element type that is a *subtype of* `Number`, including `Number` itself. The parameterized types `Pair<Number>`, `Pair<Double>` and `Pair<Integer>` are *subtypes* of the parameterized type `Pair<? extends Number>`, because `Number`, `Double` and `Integer` are *subtypes* of `Number`. With the new implementation of the method `sumPair()`, the code in (1) above will compile. Here are some further examples of how subtyping between parameterized types works:

```
Pair<Double> doublePair = new Pair<Double>(100.50, 200.50);
double newSum = sumPair(doublePair); // (2) Ok
Pair<Number> numPair = doublePair; // (3) Compile-time error!
Pair<? extends Number> newPair = doublePair; // (4) Ok
```

In (2) we are just passing a pair with the type `Pair<Double>` to a formal parameter of type `Pair<? extends Number>` in the method `sumPair()`. The type `Pair<Double>` is a subtype of the type `Pair<? extends Number>`, and therefore the parameter passing is valid. In (3) the type of the reference `doublePair` is `Pair<Double>` and the type of the reference `numPair` is `Pair<Number>`. The type `Pair<Double>` is *not* a subtype of the type `Pair<Number>`, and there-



fore the assignment is *not* valid. In (4) the type of the reference `doublePair` is `Pair<Double>` and the type of the reference `newPair` is `Pair<? extends Number>`. The type `Pair<Double>` is a subtype of the type `Pair<? extends Number>`, and therefore the assignment is valid.

## Specifying supertypes: `<? super T>`

The parameterized type `Pair<? super Integer>` in (5) below stands for all pairs with an element type that is a *supertype of* `Integer`, including `Integer` itself. The parameterized type `Pair<? super Number>` is the *supertype* of the parameterized types `Pair<Number>`, `Pair<Comparable>` and `Pair<Integer>` because `Number`, `Comparable` and `Integer` are *super-types* of `Integer`. Here are some further examples:

```
Pair<Number> numPair = new Pair<Number>(100.0, 200);
Pair<Integer> iPair = new Pair<Integer>(100, 200);
Pair<? super Integer> supPair = numPair; // (5) Ok
supPair = iPair; // (6) Ok
supPair = doublePair; // (7) Ok
```

The assignments (5), (6) and (7) are valid because the type of the left-hand side reference is a supertype of the type of the reference on the right-hand side.

## Specifying any type: `<?>`

The wildcard `?` can also be used on its own to represent any arbitrary type. For example the parameterized type `Pair<?>` represents the type of *all* pairs, regardless of their specific element type. In other words, the parameterized type `Pair<?>` is the *supertype for all parameterized types* of the generic class `Pair<T>`. The parameterized type `Pair<?>` is thus the *supertype* of the parameterized types `Pair<? extends Number>`, `Pair<? super Integer>`, `Pair<Number>` and `Pair<Integer>`. Again, here are some examples:

```
Pair<?> pairB = numPair; // (8) Ok
pairB = newPair; // (9) Ok
pairB = supPair; // (10) Ok
pairB = new Pair<?>(100.0, 200); // (11) Compile-time error!
pairB = new Pair<? extends Number>(100.0, 200); // (12) Compile-time error!
```

The assignments (8), (9) and (10) are valid because the type of the right-hand side reference is a subtype of the parameterized type `Pair<?>` of the reference on the left-hand side. The code in (11) and (12) results in a compile-time error because the exact type of the object to create is not known. Therefore the use of the wildcard `?` in a constructor call is not allowed.

## Generic methods

A method can declare its own formal type parameters and use them in the method. We illustrate such a *generic method* by implementing a method that takes an array and returns a list that contains the same elements as the array, i.e. converts an array to a list. We note that there is dependency between the element type of the array and the element type of the list: elements in both must have the same type. A formal type parameter `T` can be used to represent this element type. In the implementation below we have used it in the method head, both to specify the array type `T[]` and the list type `List<T>` at (1). The



formal type parameter `T` is specified immediately before the return type of the method (`List<E>`), enclosed by the characters `<` and `>`, analogous to what we have seen for generic types.

```
public static <T> List<T> arrayToList(T[] array) { // (1)
 List<T> list = new ArrayList<T>(); // Create an empty list.
 for (T element : array) // Traverse the array.
 list.add(element); // Copy current element to list.
 return list; // Return the list.
}
```

In a generic method the formal type parameter `T` is only accessible in the method itself. We have used it in the method `arrayToList()` in places where it is necessary to specify the element type.

We can call a generic method in the usual way. The compiler determines the actual type parameter from the method call:

```
String[] strArray = {":-(", ";-)", ":-)"};
List<String> strList = arrayToList(strArray); // T is String.
Integer[] intArray = {2007, 7, 2};
List<Integer> intList = arrayToList(intArray); // T is Integer.
```

We show the declarations of the method headers for a couple of generic methods, leaving the implementation of the method bodies as a programming exercise:

```
static <E> void insertionSort(List<E> list, Comparator<E> comp) { ... } // (4)
static <T extends Comparable<T>> int binsearch(T[] array, T key) { ... } // (5)
```

The method `insertionSort()` at (4) sorts a list according to the specified `Comparator` object. The method uses the insertion sort algorithm. The formal type parameter `E` represents the type of the elements in the list and of elements that can be compared by the `Comparator` object. Using the formal type parameter `E` this way ensures that the `Comparator` object can be applied to sort the elements of the list.

The method `binsearch()` at (5) finds the index of a key in a sorted array, if the key exists in the array. The method uses the algorithm for binary search. The formal type parameter `T` represents the type of the elements in the array and the type of the key. The type `<T extends Comparable<T>>` is an example of a type parameter that specifies a *bound* for `T`. It indicates that `T` must implement the `Comparable<T>` interface, so that it is possible to compare objects of type `T`.

There is no requirement that a generic method must be declared in a generic class. But a generic method in a generic class can, in addition to its own formal type parameters, also avail itself of the formal type parameters of the generic class.



### BEST PRACTICES

If there are dependencies between the return type and the types of the formal parameters, or among the types of the formal parameters themselves, consider a generic implementation for the method.

## 15.9 Sorting and searching methods from the Java APIs

The classes `java.util.Collections` and `java.util.Arrays` in the Java standard library provide a number of overloaded generic methods for sorting and searching. Some selected methods are shown in Table 15.10 and Table 15.11. See also Table 9.2 on page 260.

Program 15.8 demonstrates sorting and searching using methods from the `Collections` class.

### BEST PRACTICES

For sorting and searching in *arrays* use the static methods from the `java.util.Arrays` class, and for sorting and searching in *lists* use the static methods from the `java.util.Collections` class.

**TABLE 15.10 Selected methods from the `Collections` class**

| <code>java.util.Collections</code>                                                                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|-----------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>static &lt;T&gt; int binarySearch(     List&lt;? extends Comparable &lt;? super T&gt;&gt; list,     T key)</pre> | Uses binary search to search for the specified key. The method uses the <i>natural order</i> as defined by the <code>Comparable&lt;T&gt;</code> interface to compare two objects. It returns the index of the object that is equal to the key, if such an object exists in the list. Otherwise the method returns a negative value which corresponds to $(-\text{insertionIndex} - 1)$ , where <code>insertionIndex</code> is the position where the key would have been found, had it existed in the list. The list must be sorted in ascending order beforehand, otherwise the result is undefined. |



### java.util.Collections

|                                                                                                                                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|----------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>static &lt;T&gt; int binarySearch(     List&lt;? extends T&gt; list,     T key,     Comparator&lt;? super T&gt; comp)</pre> | <p>Uses binary search to search for the specified key. The method uses the <code>Comparator&lt;T&gt;</code> object to compare two objects. The method returns the index of the object that is equal to the key, if such an object exists in the array. Otherwise the methods returns a negative value which corresponds to <math>(-\text{insertionIndex} - 1)</math>, where <code>insertionIndex</code> is the position where the key would have been found, had it existed in the array. The list must be sorted in ascending order beforehand, otherwise the result is undefined. If the value of the parameter <code>comp</code> is <code>null</code>, <i>natural order</i> is used.</p> |
| <pre>static &lt;T&gt; void sort(List&lt;T&gt; list)</pre>                                                                        | <p>Sorts a list with elements of type <code>T</code>, in ascending order, based on the natural order of the objects.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <pre>static &lt;T&gt; void sort(List&lt;T&gt; list,     Comparator&lt;? super T&gt; comp)</pre>                                  | <p>Sorts a list with elements of type <code>T</code>, in ascending order, based on the order defined by the <code>Comparator&lt;T&gt;</code> object.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

**TABLE 15.11** Selected generic methods from the `Arrays` class

### java.util.Arrays

|                                                                                                               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|---------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>static &lt;T&gt; int binarySearch(     T[] array,     T key,     Comparator&lt;? super T&gt; comp)</pre> | <p>Uses binary search to search for the specified key. The method uses the <code>Comparator</code> object to compare two objects. The method returns the index of the object that is equal to the key, if such an object exists in the array. Otherwise the methods returns a negative value which corresponds to <math>(-\text{insertionIndex} - 1)</math>, where <code>insertionIndex</code> is the position there the key would have been found had it existed in the array. The array must be sorted in ascending order beforehand, otherwise the result is undefined. If the value of the parameter <code>comp</code> is <code>null</code>, <i>natural order</i> is used.</p> |
|---------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

---

## java.util.Arrays

---

`static <T> List<T> asList(T... arguments)` The parameter declaration `T...` `arguments` indicates that the method can accept zero or more arguments. These arguments are passed to the method as an array. The method returns a fixed-size list that is backed by this array that is passed in the parameter `arguments`.

---

15



### PROGRAM 15.8 Sorting and searching with methods from the Java APIs

```
import java.util.List;
import java.util.Collections;
import java.util.Arrays;

public class SortingAndSearchingLists {
 public static void main(String[] args) {
 Integer[] numArray = {8, 4, 2, 6, 1};
 // Create a list from numArray.
 List<Integer> numList = Arrays.asList(numArray);

 // Sort the list.
 System.out.println("Unsorted list: " + numList);
 Collections.sort(numList);
 System.out.println("Sorted list: " + numList);

 // Search in the list.
 int index = Collections.binarySearch(numList, 4);
 System.out.println("Key " + 4 + " : index " + index); // Key exists.
 System.out.println("Key " + 5 + " : index " + // Key does not exist.
 Collections.binarySearch(numList, 5));
 }
}
```

Program output:

```
Unsorted list: [8, 4, 2, 6, 1]
Sorted list: [1, 2, 4, 6, 8]
Key 4 : index 2
Key 5 : index -4
```

---

### 15.10 Review questions

1. Which methods are found in the `StringBuilder` class, but not in the `String` class?

a `deleteCharAt(int index)`

- b** `reverse()`
  - c** `compareTo(Object obj)`
  - d** `setCharAt(int index, char ch)`
- (a), (b), (d)

Methods in (a), (b) and (d) modify the character sequence in a string builder, and the state of a `String` object cannot be modified.

- 2.** If the character sequence is modified frequently, which class will you choose to represent it? Explain your choice.

The `StringBuilder` class is the best choice, as it implements dynamic strings, in contrast to the `String` class that implements immutable strings.

- 3.** Explain the difference between the following terms:

- a** *Static data structures* and *dynamic data structures*
- b** *Data structure* and *abstract data types*

A static data structure is organized so that the number of elements that can be stored in it is fixed (`array`, `String`). A dynamic data structure can expand and shrink as needed when elements are inserted or removed from it.

A data structure organizes and stores data values in a particular way. An abstract data type (ADT) is a data structure that in addition defines operations on the data structure. Typical operations are insertion and lookup on the data structure. Implementation details are not available to the clients of ADTs.

- 4.** Which statements are true about collections?

- a** Interfaces `Set<E>` and `List<E>` extend the `Collection<E>` interface.
- b** Set operations are a part of the `Collection<E>` interface.
- c** The class `HashSet<E>` provides set operations.
- d** The class `ArrayList<E>` provides set operations.

All: (a), (b), (c), (d)

- 5.** Which statements about lists (`ArrayList<E>`) and sets (`HashSet<E>`) are true?

- a** Elements in a list can be accessed by their positional value.
- b** Elements in a set can be accessed by their positional value.
- c** A list can contain duplicates.
- d** A set can contain duplicates.
- e** Elements in a list are stored in the same order as the order in which they are inserted.
- f** Elements in a set are stored in the same order as the order in which they are inserted.

(a), (c), (e)





Analogous statements about sets are not true.

- 6.** What is an iterator? Which methods are provided by the `Iterator<E>` interface?

- a** `next()`
- b** `add()`
- c** `remove()`
- d** `hasNext()`

An iterator is an object that makes it possible to access all the elements in the underlying collection one at a time in a systematic way.

- (a), (c), (d)

- 7.** Insert the correct identifiers in the code below.

```
Collection<_____> myCollection = new HashSet<String>();
myCollection.add("9"); myCollection.add("1"); myCollection.add("1");

Iterator<_____> iterRef = _____ .iterator();
while (_____.hasNext()) {
 System.out.print(_____.next());
}

Collection<String> myCollection = new HashSet<String>();
myCollection.add("9"); myCollection.add("1"); myCollection.add("1");

Iterator<String> iterRef = myCollection.iterator();
while (iterRef.hasNext()) {
 System.out.print(iterRef.next());
}
```

- 8.** A map contains \_\_\_\_\_.

A map contains *entries*.

- 9.** An entry comprises a \_\_\_\_\_ and a \_\_\_\_\_.

An entry comprises a *key* and a *value*.

- 10.** Which statements about maps (`HashMap<K,V>`) are true? Explain your choice in each case.

- a** A map can have several entries with the same key.
- b** A map can have several entries with the same value.
- c** Entries in a map are stored in the same order in which they are inserted.
- d** If an entry for a key exists from before, the method `put(K key, V value)` will insert a new entry and not overwrite the old one.
- e** The interface `Map<K,V>` provides the method `iterator()` which can be used to access all the entries in a map.

- (b)



A key is unique in a map. A client cannot assume that the entries in a map are stored in the same order as they were inserted. The method `put(K key, V value)` will overwrite the old entry of the key, if it exists from before. The interface `Map<K, V>` does not provide any `iterator()` method.

**11.** Which statements about the hash values of objects are true?

- a** Two objects that are different according to the `equals()` method, must have different hash values.
- b** Two objects that are equal according to the `equals()` method, must have different hash values.
- c** Two objects that are equal according to the `equals()` method, must have the same hash value.

(c)

It is recommended that (a) is fulfilled, but it is not a requirement.

**12.** The method `keySet()` returns a view with all the \_\_\_\_\_, and the method `values()` returns a view with all the \_\_\_\_\_ in a map. What is the difference between the two views?

The method `keySet()` returns a view with all the *keys*, and the method `values()` returns a view with all the *values* in a map. The key view is a set (`Set<E>`), since keys are unique. The value view is a collection (`Collection<E>`) and can have duplicates.

**13.** Which statements about the map views are true?

- a** We can call the method `iterator()` on a map view.
- b** The method `keySet()` returns a view that is a set (`Set<E>`).
- c** The method `values()` returns a view that is a collection (`Collection<E>`).
- d** Changes via the view are apparent in the underlying map.

All: (a), (b), (c), (d)

**14.** What are the possible outputs from the following program? Explain your reasoning.

```

import java.util.HashMap;
import java.util.Map;
import java.util.Set;

public class MapIteration {

 public static void main(String args[]) {
 Map<String, String> map = new HashMap<String, String>();
 map.put("Step 1: ", "Create a view with keys.");
 map.put("Step 2: ", "Declare a key variable in a for(:) loop.");
 map.put("Step 3: ", "Get the value of a key in the loop body");

 Set<String> view = map.keySet();
 for (String key : view)
 System.out.println(map.get(key));
 }
}

```

```

 }
}

a Get the value of a key in the loop body.

 Declare a key variable in a for(:) loop.

 Create a view with keys.

b Create a view with keys.

 Declare a key variable in a for(:) loop.

 Get the value of a key in the loop body.

c Step 1:

 Step 2:

 Step 3:

d Step 2:

 Step 1:

 Step 3:

e Step 1: Create a view with keys.

 Step 2: Declare a key variable in a for(:) loop.

 Step 3: Get the value of a key in the loop body.

```

15



(a), (b)

Both keys and values are strings in the program. Only value strings will be printed. This excludes (c), (d) and (e). The sequence in which the value strings will be printed is dependent on the order in which the keys are looked up in the key view.

**15.** Which statements are true? Assume the following class declarations:

```

class Thingy<T> { }
class A {}
class B extends A implements Comparable { }
class C extends B {}

```

- a** Thingy<? extends B> is a supertype of Thingy<B>, Thingy<? extends C> and Thingy<C>.
- b** Thingy<? super C> is a supertype of Thingy<A>, Thingy<Comparable>, Thingy<B> and Thingy<C>.
- c** Thingy<?> is a supertype of Thingy<Object>, Thingy<A>, Thingy<B> and Thingy<C>.

All: (a), (b), (c)

**16.** Which declarations are valid generic methods?

- a** public static <T> T oneGenericMethod(T t1, T t2) { ... }
- b** public <T> static T oneGenericMethod(T t1, T t2) { ... }
- c** T <T> oneGenericMethod(T t1, T t2) { ... }
- d** <T> T oneGenericMethod(T t1, T t2) { ... }
- e** T oneGenericMethod(T t1, T t2) { ... }

(a), (d)



In (b) and (c) the formal type parameter `T` is not specified before the return type.

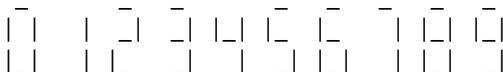
In (e) no formal type parameter is specified.

- 17.** Which data structure (list, set, map) will you choose for the following purposes?  
Why?

- a Find the result of all the dice throws by a player during a game. (`list`)
  - b Separate pupils according to gender before they go into the changing rooms. (`set`)
  - c Keep track of numbers that come up in a lottery drawing (`set, list`)
  - d Keep track of the number of text messages sent from mobile phones. (`map`)
- (a) A list, where the result of each throw can be inserted progressively.
- (b) Two sets, one for boys and one for girls.
- (c) A set or a list, depending on whether the sequence in which the numbers are drawn is important.
- (d) A map, where each entry comprises a phone number and the number of text messages sent from the phone.

## 15.11 Programming exercises

- 1.** A simple display shows a number, where each digit comprises seven segments which can be switched on and off in order to create the digit:



Write a program that implements such a display. The program reads a number from the command line and shows it in the display. For example, running the program with the command:

```
>java DigiDisplay 321
```

will print the following:



Note that the printing of the number spans over three lines, and there are three characters for each digit per line. Use three string builders to create the output for each digit that is to be printed, with a space between the digits. For example, if the current digit is 3, we insert the strings " \_ ", " \_| " and " \_| " respectively in the three string builders, where the spaces are included in the strings.

- 2.** Modify the generic class `Pair<T>` from Program 15.2 on page 471, so that it is possible to have two different types of objects in a pair.



3. Write a program that takes two sorted list and merges them into a single list. For example, given the following two lists:

List no. 1: 12 19 26 33 33 40

List no. 2: 4 12 20 20 33

merging will result in the following sorted list:

Merged list: 4 12 12 19 20 20 26 33 33 33 40

Use the class `ArrayList<E>` to create the lists. The list can for example contain `Integer` objects. The program can read the values from the terminal window.

The following problems are to be solved by creating sets (`HashSet<E>`) based on the three lists:

- Find the values that the two sorted list have in common.
- Find all the duplicate values in the merged list, and remove them from this list.

Find the values that the two sorted list have in common. Find all the duplicate values in the merged list, and remove them from this list.

Given the list above, the following values are common to the two sorted lists:

Common values: 12 33

and the following duplicates were removed from the merged list:

Duplicates: 12 20 33

so that the final merged list becomes:

Final merged list: 4 12 19 20 26 33 40

4. Write a program that reads a string from the command line and prints the number of characters that occur only once in the string. Use sets (`HashSet<E>`) to solve this problem. For example, in the string "banana" only the character 'b' occurs once, so the answer is 1.
5. Write a program that reads a sequence of digits from the terminal window and prints a report about how many times the digits 0, 1, ..., 9 occur in the sequence. Assume that the sequence is terminated by a negative integer. For example, if the user types:

8  
0  
2  
8  
8  
9  
5  
9  
-3

the program prints the following report:

0 occurs 1 time  
 2 occurs 1 time  
 5 occurs 1 time  
 8 occurs 3 times  
 9 occurs 2 times

The program uses a map (`HashMap<K,V>`) to keep track of the frequency of each digit.

- 6.** Create a program that converts a word with uppercase letters and digits to corresponding Morse code (see the table below). Use a map (`HashMap<K,V>`) to implement the Morse code. The program can read the word from the command line. In the output there should be one space between two Morse characters and two spaces between two Morse words.

**TABLE 15.12** Morse code [Table is not a valid element here. please fix.]

| Character | Code        | Character | Code          |
|-----------|-------------|-----------|---------------|
| A         | . -         | S         | ... .         |
| B         | - ... .     | T         | -             |
| C         | - . -. .    | U         | ... . .       |
| D         | - . .. .    | V         | ... . . .     |
| E         | .           | W         | . . . .       |
| F         | . . . . .   | X         | - . . . .     |
| G         | - . . . .   | Y         | - . . . . .   |
| H         | . . . . . . | Z         | - . . . . . . |
| I         | ..          | 0         | - - - -       |
| J         | . - - - .   | 1         | . - - - -     |
| K         | - . - . .   | 2         | . - - - -     |
| L         | . - . . .   | 3         | . . - - -     |
| M         | --          | 4         | . . . - -     |
| N         | - . -       | 5         | . . . . -     |
| O         | - - - .     | 6         | - . . . .     |
| P         | - . - . .   | 7         | - - . . .     |
| Q         | - - . - .   | 8         | - - - . .     |
| R         | - . - . . . | 9         | - - - . . .   |





7. Implement a generic method `insertionSort()` that sorts a list according to the order defined by a `Comparator` object. The method should use the insertion sort algorithm. The formal type parameter `E` represents the element type of the elements in the list and elements that can be compared by the `Comparator<E>` object.

```
static <E> void insertionSort(List<E> list, Comparator<E> comp) { ... }
```

Test the method on a list of `String` objects, where the `Comparator<String>` object is given by `String.CASE_INSENSITIVE_ORDER`. This `Comparator<String>` object compares two strings without taking into consideration whether the strings contains uppercase or lowercase letters.

8. Implement a generic method `binSearch()` that returns the index of a key in an array, if the key exists in the array. The method should use the algorithm for binary search. The formal type parameter `T` represents the element type of the elements in the array and the type of the key. This type implements the `Comparable<T>` interface, so that it is possible to compare elements according to their natural order.

```
static <T extends Comparable<T>> int binSearch(T[] array, T key) { ... }
```

Test the method on a sorted array using suitable keys.

9. Implement a generic method `invertMap()` as it is described below.

```
static <K,V> Map<V,List<K>> invertMap(Map<K,V> map1) { ... }
```

Given a map with entries `<key, value>`, the map creates a new map with entries `<value, list of keys>`. For example, given the map:

```
<k2, v1>
<k1, v3>
<k4, v2>
<k6, v3>
<k5, v1>
<k3, v1>
```

the method creates the following map:

```
<v3, [k6, k1]>
<v1, [k3, k5, k2]>
<v2, [k4]>
```

Test the method on a map with suitable entries.

10. Implement a version of the class `PersonnelRegister` (Program 8.7 on page 222) that uses a dynamic data structure, for example, a list (`ArrayList<E>`).

## Implementing Dynamic Data Structures

### LEARNING OBJECTIVES

By the end of this chapter, you will understand the following:

- What abstractions dynamic data structures like linked lists, stacks and queues represent.
- How to implement collections like linked lists, stacks and queues as generic data types.
- How to use collections like linked lists, stacks and queues.

### INTRODUCTION

In this chapter we will implement some classical data structures: *linked lists*, *stacks* and *queues*. All these data structures are provided by the Java standard library, but we will implement them as an exercise in programming to show how abstract data types (ADTs) can be developed. Defining operations for these ADTs also provides examples of developing algorithms and their implementation.

It is a good idea to review the sections on generic types (Section 15.3, Section 15.8) before reading this chapter. At least some familiarity with the terminology from Section 15.3 will be useful.

### 16.1 Simple linked lists

A *simple linked list* is a *linear* collection of objects, called *nodes*, that are linked together in such a way that a node in the list has a reference to the next node in the list. A node is an example of an object that has a reference to an object of the same type as itself, i.e. another node. Such objects are also called *self referencing* objects. Program 16.1 shows an example of a node. The generic class `Node<E>` has two private fields: `data` and `next`. The field `data` can store an object determined by the type parameter `E`. The field `next` is a refer-



ence of the type `Node<E>` and indicates the next node in the list. In addition a node can contain other pertinent data and auxiliary methods. The class `Node<E>` has methods to set and get information in a node. Which type of data is stored in an object of class `Node<E>` is determined when the object is created with a specific type argument.

In our implementation a list has a *head*, that is a reference to the *first* node in the list, and a *tail*, that is a reference to the *last* node in the list. The head and the tail is used to implement list operations. The class diagram in Figure 16.1 shows the relationship between a linked list and its nodes that store the data and the reference to the next node in the list. Note that the reference `next` refers to a `Node<E>` object.

### PROGRAM 16.1 Class `Node<E>`

```
/** A node holds data and a reference to a node. */
public class Node<E> {

 /** Data in the node. */
 private E data;
 /** Reference to the next node. */
 private Node<E> next;

 /**
 * The constructor initializes the current node with data
 * and reference to the next node.
 * @param data_obj Data to be stored in the current node.
 * @param nodeRef Reference to the next node that is
 * stored in the current node.
 */
 public Node(E data_obj, Node<E> nodeRef) {
 data = data_obj;
 next = nodeRef;
 }

 /** Set data in the current node. */
 public void setData(E obj) { data = obj; }

 /** Set reference to the next node in the current node. */
 public void setNext(Node<E> node) { next = node; }

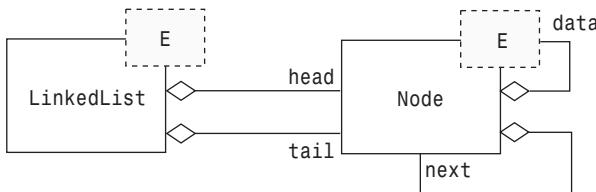
 /** Get data stored in the current node.
 * @return Data in the current node */
 public E getData() { return data; }

 /** Get referense to the next node from the current node.
 * @return Reference to the next node */
 public Node<E> getNext() { return next; }
}
```



FIGURE 16.1

Class diagram for a simple linked list



## Manipulating references in a linked list

A concrete linked list based on the class diagram in Figure 16.1 is shown in Figure 16.2. The list has the type argument `String`. This means that the fields `head` and `tail` in Figure 16.1 have the type `Node<String>`, and we can store strings in the nodes. Figure 16.2 shows such a list with four strings. Note that the field `next` in the last node is `null` to mark that this node is the tail node, i.e. it has no successor. The fields `data` and `next` in a node are `private`, so that a client must use `public` methods defined in the class `Node<E>` in order to access them.

Figure 16.2 also shows a `Node<String>` reference `p` that refers to a node in the list. Given a reference to a node, we can use this reference to set data in and get data from the node:

```

String str = head.getData(); // Get data from the first node.
p.setData("frost"); // Data in the node referenced by p is changed to "frost".

```

We can get hold of the successor to the node referenced by the reference `p`:

```

Node<String> successor = p.getNext(); // Get successor to p.

```

Given that the reference `q` refers to a `Node<String>` object, we can set this node as the successor to the node referenced by the reference `p`:

```

p.setNext(q); // q set as successor to p.

```

We can “move” the reference `p` so that it refers to its successor:

```

p = p.getNext(); // Reference p now refers to its successor.

```

We must be careful when we manipulate the successor to a node, in order to avoid unintended consequences. For example, if there were several nodes in a list, the following assignment will remove all the nodes, except for the first one in the list:

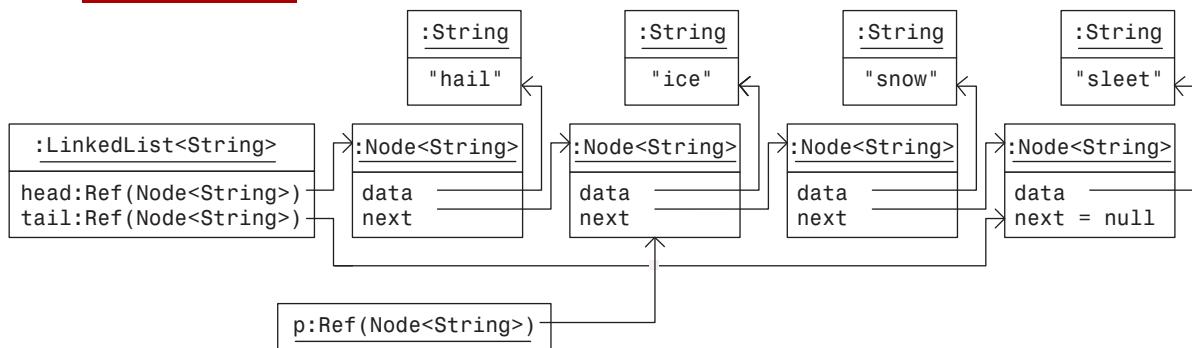
```

head.setNext(null); // All nodes except first node are removed.

```



**FIGURE 16.2** A linked list of strings



## Operations on linked lists

Operations that we are interested in implementing for a linked list, are specified in the generic interface `ILinkedList<E>` in Program 16.2. The interface declares a number of mutator methods for lists. A *mutator* is a method that changes the state of the object on which it is called. We see that we can insert and remove nodes both from the head and the tail of the list. The method `remove()` removes the node containing the data given by the formal parameter. The interface also declares a number of selectors. A *selector* is a method that reads (a part of) the state of the object on which it is called. The interface declares selectors to determine whether the list is empty, which node is the first node or the last node in the list, and how many nodes there are in the list. The method `find()` finds the node containing the data given by the formal parameter. The method `listToArray()` fills an array from data stored in the list.

The interface `ILinkedList<E>` extends the interface `Iterable<E>`, so that it inherits the method `iterator()` from the interface `Iterable<E>`. This method creates an iterator for the list. In other words, we can use a `for(:)` loop to traverse a list that implements the interface `ILinkedList<E>`.

Program 16.3 shows the definition of the class `LinkedList<E>` that implements the interface `ILinkedList<E>`. The class has three private fields, `head`, `tail` and `numberOfNodes`, that represent the head, the tail and the number of nodes in the linked list at any given time, respectively.

Figure 16.3 shows an inheritance hierarchy for linked lists. We will look at the details in implementing a part of the interface for linked lists. Manipulating a linked list is primarily a question of assigning the right reference values to maintain the integrity of the linked list. To get a better overview of reference manipulation, it is useful to draw diagrams (see figures later in this section). Operations on linked lists are also good examples of developing simple algorithms.



## BEST PRACTICES

Defining an interface to represent the contract of your class results in well-defined boundaries for your class, aiding both its use and its implementation.

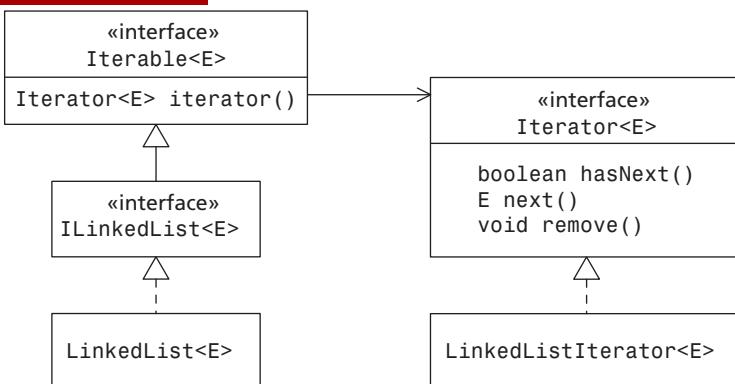
### PROGRAM 16.2 Interface `ILinkedList<E>`

```
public interface ILinkedList<E> extends Iterable<E> {
 // Mutators
 public void insertAtHead(E data_obj);
 public void insertAtTail(E data_obj);
 public E removeFromHead();
 public E removeFromTail();
 boolean remove(E data_obj);

 // Selectors
 public boolean isEmpty();
 int numberofNodes();
 Node<E> first();
 Node<E> last();
 Node<E> find(E data_obj);

 // Other methods
 void listToArray(E[] array);
}
```

**FIGURE 16.3** Inheritance hierarchy for implementing linked lists



(a) Implementing linked lists

(b) Implementing an iterator for linked lists



### PROGRAM 16.3 Classes `LinkedList<E>` and `LinkedListIterator<E>`

```

import java.util.Iterator;
/**
 * This class implements a linked list, allowing nodes to be inserted and
 * removed.
 */
public class LinkedList<E> implements ILinkedList<E> {

 /** Head of the list */
 private Node<E> head;

 /** Tail of the list */
 private Node<E> tail;

 /** Number of nodes in the list */
 private int numberOfNodes;

 // Mutators
 /**
 * Insert a node at the head of the list.
 * @param dataObj Data to be stored in the list.
 */
 public void insertAtHead(E dataObj) { // (1)
 Node<E> p = new Node<E>(dataObj, null);
 if (isEmpty()) {
 head = tail = p;
 } else { // head = new MyNode<E>(dataObj, head);
 p.setNext(head); // Step 1
 head = p; // Step 2
 }
 numberOfNodes++;
 }

 /**
 * Insert a node at the tail of the list.
 * @param dataObj Data to be stored in the node.
 */
 public void insertAtTail(E dataObj) { // (2)
 Node<E> p = new Node<E>(dataObj, null);
 if (isEmpty()) {
 head = tail = p;
 } else {
 tail.setNext(p); // Step 1
 tail = p; // Step 2
 }
 numberOfNodes++;
 }
}

```



```

/**
 * Remove first node in the list.
 * Assume the list is not empty.
 * @return Data in the node which is removed.
 */
public E removeFromHead() { // (3)
 assert numberOfNodes > 0 : "The list cannot be empty.";
 Node<E> p = head; // Step 1
 if (head == tail) {
 head = tail = null; // Only one node in the list.
 } else {
 head = p.getNext(); // Step 2
 }
 numberOfNodes--;
 return p.getData();
}

/**
 * Remove last node in the list.
 * Assume the list is not empty.
 * @return Data in the node which is removed.
 */
public E removeFromTail() { // (4)
 assert numberOfNodes > 0 : "The list cannot be empty.";
 Node<E> p = tail; // Step 1
 if (head == tail) {
 head = tail = null; // Only one node in the list.
 } else {
 Node<E> nextToLast = head; // Step 2
 while (nextToLast.getNext() != tail) {
 nextToLast = nextToLast.getNext();
 }
 tail = nextToLast; // Step 3
 tail.setNext(null); // Step 4
 }
 numberOfNodes--;
 return p.getData();
}

/**
 * Remove node with given data from the list.
 * @param dataObj Data in the node which is removed.
 * @return True, if a node with the given data is removed.
 */
public boolean remove(E dataObj) { // (5)
 // Traverse the list.
 Node<E> currentNode = head;
 Node<E> predecessor = null;
 boolean found = false;
 while (currentNode != null && !found) {

```



```
if (currentNode.getData().equals(dataObj)) {
 found = true;
} else {
 // Not yet found. Update predecessor and currentNode references.
 predecessor = currentNode;
 currentNode = currentNode.getNext();
}
}
if (found) {
 if (predecessor == null) {
 // Found in 1st. node of the list.
 removeFromHead();
 } else if (currentNode.getNext() == null) {
 // Found in last node of the list.
 tail = predecessor;
 tail.setNext(null);
 numberOfNodes--;
 } else {
 // Found inside the list.
 predecessor.setNext(currentNode.getNext());
 numberOfNodes--;
 }
}
return found;
}

// Selectors
/***
 * Determines whether the list is empty.
 * @return True, if the list is empty, otherwise return false.
 */
public boolean isEmpty() { return head == null; }

/**
 * Get the number of nodes in the list.
 * @return Number of nodes in the list.
 */
public int numberOfNodes() { return numberOfNodes; }

/**
 * Get the first node in the list.
 * @return The first node in the list, if any.
 */
public Node<E> first() { return head; }

/**
 * Get the last node in the list.
 * @return The last node in the list.
 */
public Node<E> last() { return tail; }
```



```
/**
 * Find the node with the given data.
 * @param dataObj Data in the node to search for.
 * @return MyNode with the given data, otherwise return null.
 */
public Node<E> find(E dataObj) {
 // Traverse the list, starting from the head.
 Node<E> currentNode = head;
 while (currentNode != null) {
 if (currentNode.getData().equals(dataObj)) {
 return currentNode; // Found.
 }
 currentNode = currentNode.getNext();
 }
 return null; // Not found.
}

// Other help methods
/**
 * Make an iterator.
 * @return the iterator
 */
public Iterator<E> iterator() { // (6)
 return new LinkedListIterator<E>(this);
}

/** Fill the array from the list. */
public void listToArray(E[] toArray) { // (7)
 assert toArray.length == numberofNodes : "Wrong size of array.";
 int i = 0;
 for (E data : this) {
 toArray[i++] = data;
 }
}

import java.util.Iterator;
/**
 * Iterator for linked lists
 */
public class LinkedListIterator<E> implements Iterator<E> {
 private Node<E> currentNode;

 public LinkedListIterator(ILinkedList<E> list) {
 currentNode = list.first();
 }

 public boolean hasNext() {
 return currentNode != null;
```



```

 }

 public E next() {
 E data = currentNode.getData();
 currentNode = currentNode.getNext();
 return data;
 }

 public void remove() {
 throw new UnsupportedOperationException();
 }
}

```

---

## Inserting at the head of a linked list

The algorithm for inserting a new node at the head of a list can be formulated as follows:  
(Please fix the pseudocode font.)

```

If the list is empty:
 The head and the tail are set to refer to the new node
Else:
 Step 1: Set the head to be the successor of the new node.
 Step 2: Set the head to refer to the new node.

```

In Program 16.3 the method `insertAtHead()` at (1) implements inserting a node at the head of the list. Figure 16.4a and Figure 16.4b show the two steps in the method `insertAtHead()` for inserting a node at the head of the list when the list is not empty. It is important that these two steps are executed in the right order:

```

p.setNext(head); // Step 1
head = p; // Step 2

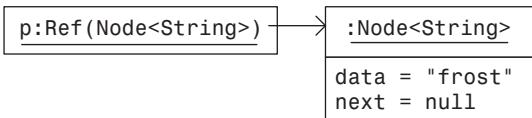
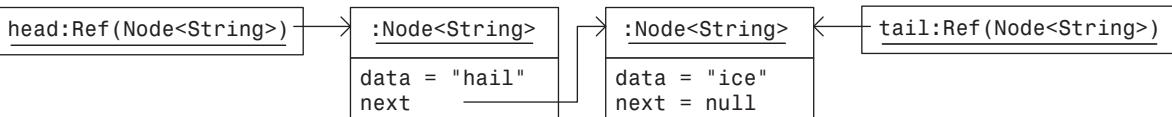
```

The code below creates the linked list shown in Figure 16.4c:

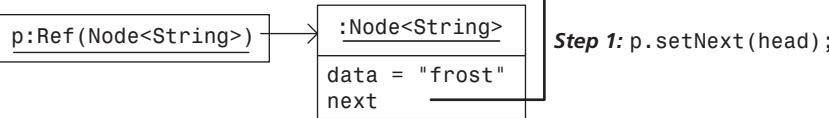
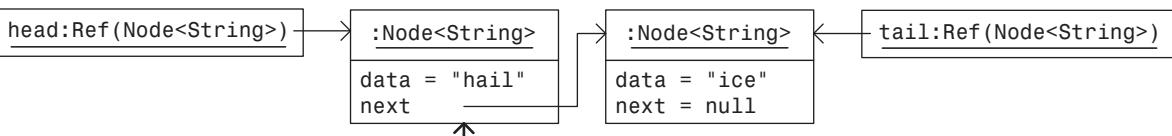
```

LinkedList<String> strList = new LinkedList<String>("ice", null);
strList.insertAtHead("hail");
strList.insertAtHead("frost");

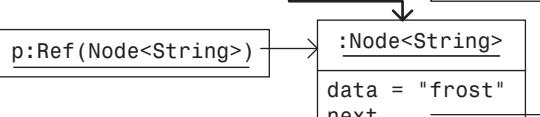
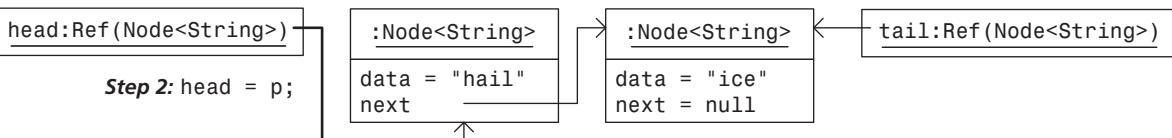
```

**FIGURE 16.4****Inserting at the head of a linked list**

(a) Before inserting at the head of the list



(b) Step 1 in inserting at the head of the list



(c) Step 2 in inserting at the head of the list

**Inserting at the tail of a linked list**

The algorithm for inserting a node at the tail of the list can be formulated as follows:  
 (Please fix the pseudocode font.)

```
If the list is empty:
 The head and the tail are set to refer to the new node.
Else:
 Step 1: Set the new node as the successor of the tail node.
 Step 2: Set the tail to refer to the new node.
```



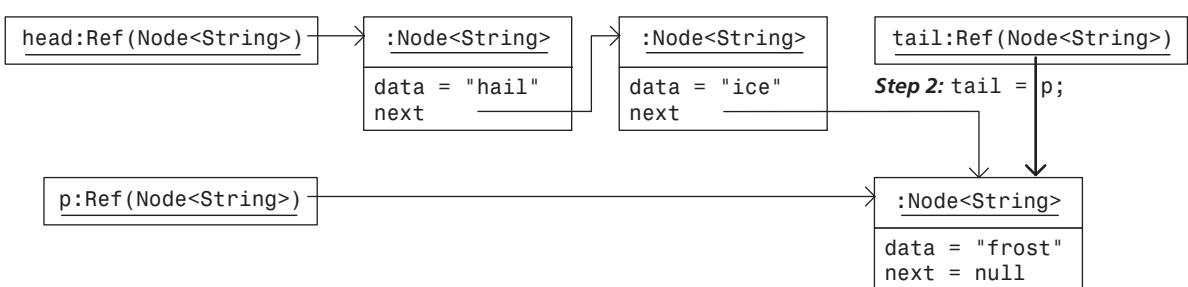
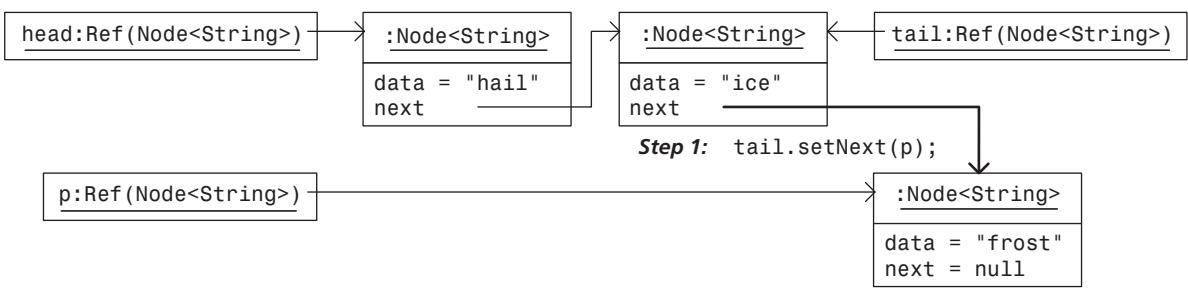
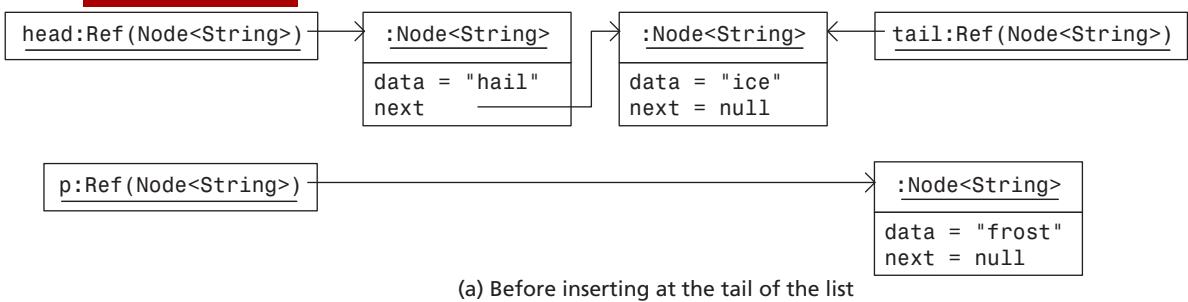
In Program 16.3 the method `insertAtTail()` at (2) implements inserting a node at the tail of the list. Figure 16.5a and Figure 16.5b show the two steps for inserting a node at the tail of the list when the list is not empty:

```
tail.setNext(p); // Step 1
tail = p; // Step 2
```

The code below creates the linked list shown in Figure 16.5c:

```
LinkedList<String> strList = new LinkedList<String>("ice", null);
strList.insertAtHead("hail");
strList.insertAtTail("frost");
```

**FIGURE 16.5** Inserting at the tail of a linked list





## Removing from the head of a linked list

The algorithm for removing a node from the head of the list assumes that the list is not empty: (Please fix the pseudocode font.)

```
Step 1: Let p refer to the first node in the list, i.e. same node as the head.
If the head and the tail refer to the same node, i.e. only one node in the list:
 Both the head and the tail are set to null.
Else:
 Step 2: Set the head to refer to the successor of the first node referenced by p.
```

In Program 16.3 the method `removeFromHead()` at (3) implements removing a node from the head of the list. Figure 16.6a and Figure 16.6b show the two steps for removing a node at the head of the list:

```
p = head; // Step 1
head = p.getNext(); // Step 2
```

The code below creates the linked list shown in Figure 16.6c:

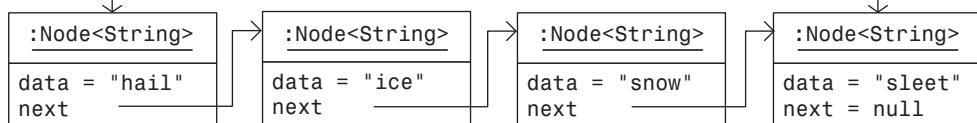
```
LinkedList<String> strList = new LinkedList<String>("ice", null);
strList.insertAtHead("hail");
strList.insertAtHead("frost");
strList.removeFromHead("frost");
```

The `removeFromHead()` method uses an assertion to check that the list is not empty.

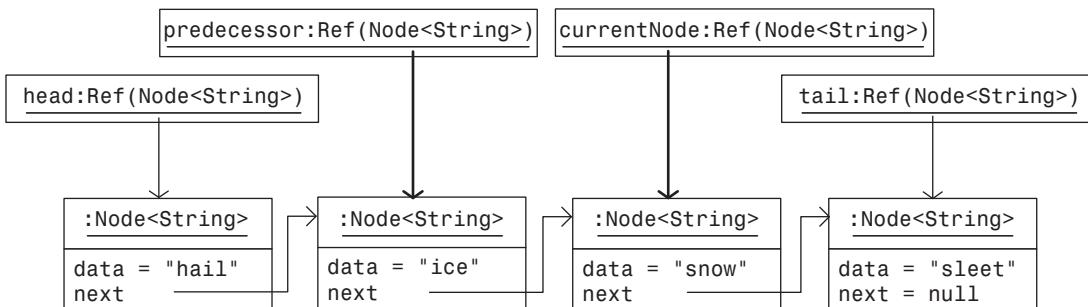


FIGURE 16.6

Removing from the head of a linked list

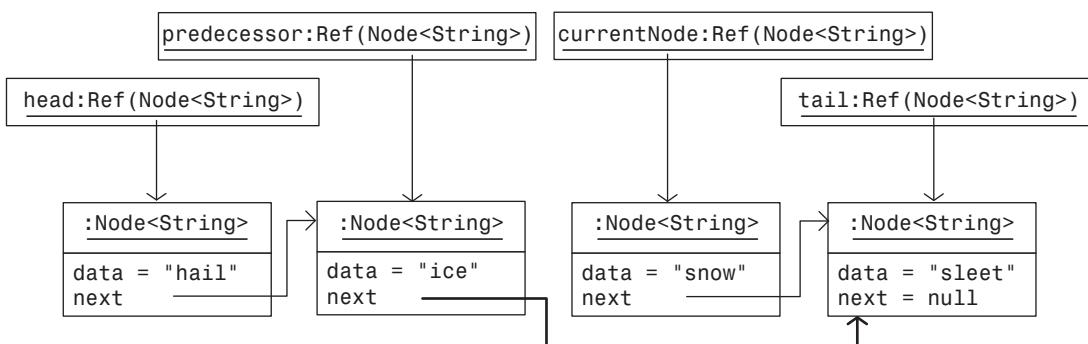


(a) Before removing a node from inside the list



Step 1: Find node that is to be deleted,  
and its predecessor;

(b) Step 1 in removing a node from inside the list



Step 2: `predecessor.setNext(currentNode.getNext());`

(c) Step 2 in removing a node from inside the list

### Removing from the tail of a linked list

Removing a node inside a list requires knowing the predecessor of the node that is to be removed. If we remove the last node, the tail must be updated to refer to the predecessor of the current last node, i.e. the next-to-last node in the list. In order to find the next-to-last node we are forced to traverse the list from the head of the list: (Please fix the pseudocode font.)

Let `nextToLast` refer to the first node in the list, i.e. same node as the head.  
Repeat while not arrived at the next-to-last node:

Move the reference `nextToLast` to the successor.



The pseudocode above can be translated to the following Java code (see step 2 in Figure 16.7c):

```
Node<E> nextToLast = head;
while (nextToLast.getNext() != tail) // Arrived at the next-to-last node?
 nextToLast = nextToLast.getNext(); // Move to the successor.
```

In Program 16.3 the method `removeFromTail()` at (4) implements removing from the tail of the list. Figure 16.7 and Figure 16.8 show the steps to remove a node from the tail of the list. We let reference `p` refer to the node that is to be removed (Figure 16.7b):

```
p = tail; // Step 1
```

Step 2 in Figure 16.7c constitutes the loop for finding the next-to-last node as described above. The steps 3 and 4 are shown in Figure 16.8a and Figure 16.8b, where the tail is updated and the last node marked to indicate the end of the list:

```
tail = nextToLast; // Step 3
tail.setNext(null); // Step 4
```



**FIGURE 16.7** Removing from the tail of a linked list (Step 1 and 2)

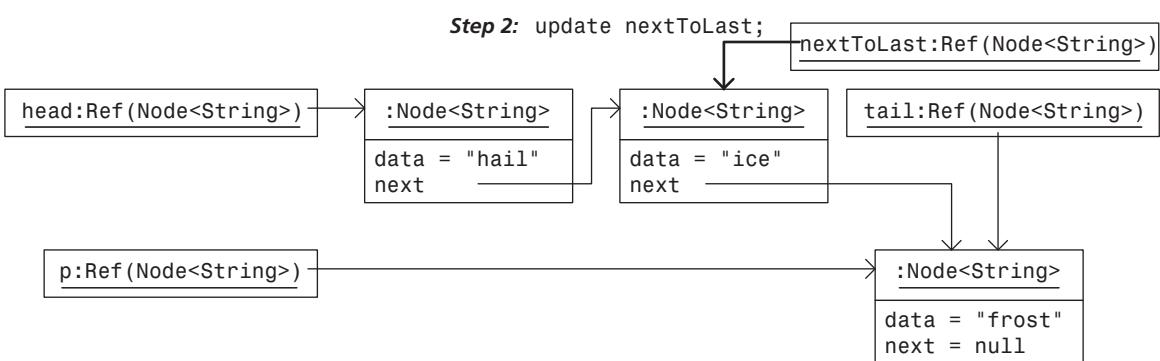
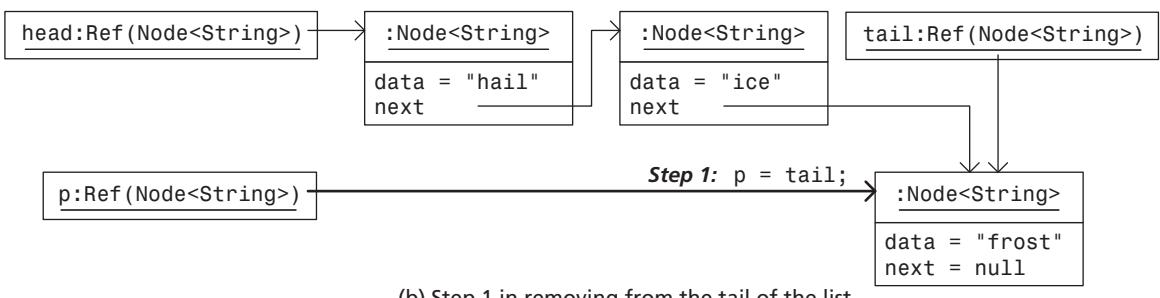
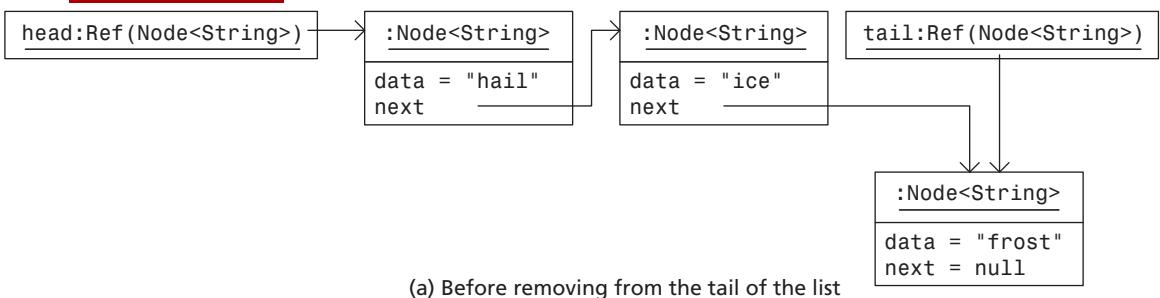
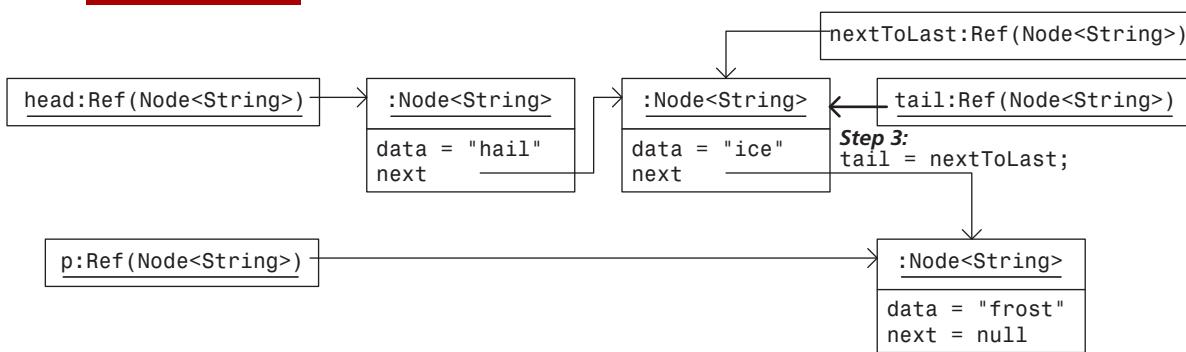


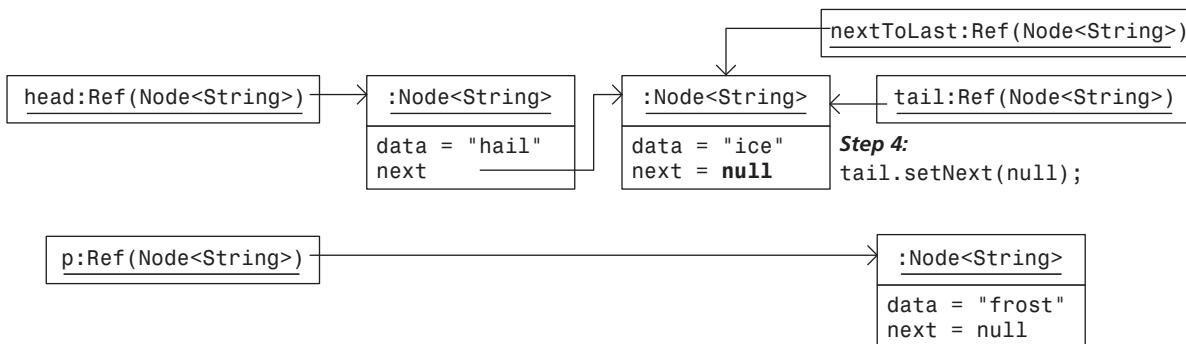


FIGURE 16.8

Removing from the tail of a linked list (Step 3 and 4)



(d) Step 3 in removing from the tail of the list



(e) Step 4 in removing from the tail of the list

## Removing a node inside a linked list

In Program 16.3 the method `remove()` at (5) removes a node with the specified data from the list. We established that removing a node inside a list requires knowing the predecessor of the node that is to be removed. In order to find the node that is to be removed, and its predecessor, we traverse the list from the head with two references that are moved in sync until we have either exhausted the whole list or found the node that is to be removed: (Please fix the pseudocode font.)

Repeat while the end of the list not reached and the node to be removed not found:

If the current node is the one to be removed:

The node is found, and thereby its predecessor.

Else:

Move the reference for the current node and for the predecessor.

The code below finds the node to be removed and its predecessor (see Figure 16.9b):

```
Node<E> currentNode = head;
Node<E> predecessor = null;
boolean found = false;
while (currentNode != null && !found) {
```



```

if (currentNode.getData().equals(dataObj)) {
 found = true;
} else {
 // Not yet found. Update predecessor and currentNode references.
 predecessor = currentNode; // (1)
 currentNode = currentNode.getNext(); // (2)
}
}

```

Note how the references `predecessor` and `currentNode` are initialised, and how these are moved in sync (lines (1) and (2)). The `while` loop terminates either when the whole list has been searched or when a node with the specified data has been found. If a node with the specified data has been found, the list must be updated. There are three cases to consider when removing a node found in the list:

- The node to be removed is the *first node* in the list (`predecessor == null`). We use the method `removeFromHead()` to remove it (see Figure 16.6).
  - The node to be removed is the *last node* in the list (`currentNode.getNext() == null`) (see Figure 16.8). The tail and the last node must be updated: **(Please fix the code indentation.)**
- ```

tail = predecessor;
tail.setNext(null);

```
- The node to be removed is *inside* the list (see Figure 16.9c). The successor to the node to be removed, now becomes the successor to the predecessor: **(Please fix the code indentation.)**
- ```

predecessor.setNext(currentNode.getNext());

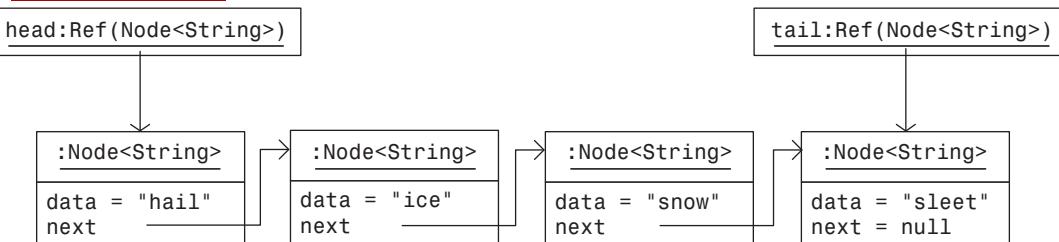
```

### BEST PRACTICES

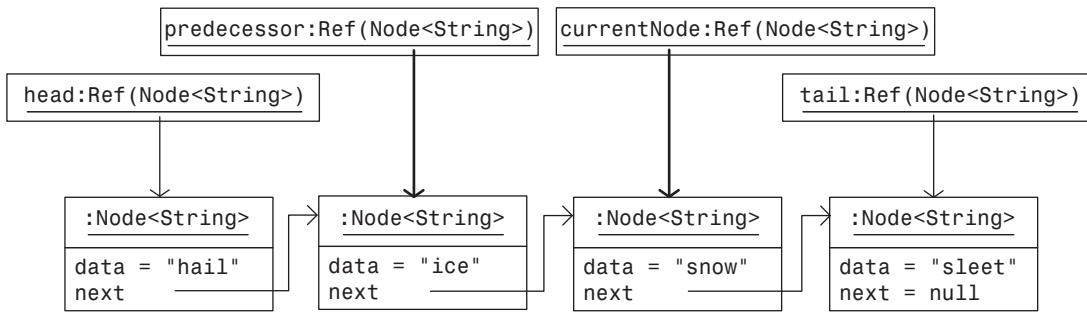
Writing algorithms for the operations of a class in pseudocode is invaluable for documenting and implementing these operations.



**FIGURE 16.9** Removing a node inside a linked list

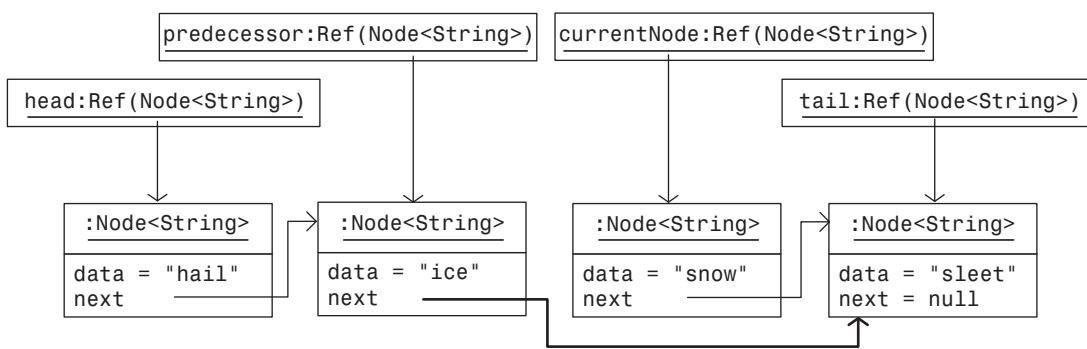


(a) Before removing a node from inside the list



Step 1: Find node that is to be deleted,  
and its predecessor;

(b) Step 1 in removing a node from inside the list



Step 2: predecessor.setNext(currentNode.getNext());

(c) Step 2 in removing a node from inside the list

## Iterator for a linked list

In Program 16.3 the class `LinkedList<E>` implements the interface `ILinkedList<E>`, and thereby indirectly also the interface `Iterable<E>`. The interface `Iterable<E>` specifies only the method `iterator()` that returns an iterator. This iterator must satisfy the interface `Iterator<E>` shown in Table 15.5.



In Program 16.3 the method `iterator()` at (6) returns an iterator for linked lists, implemented by the class `LinkedListIterator<E>`. This class has a field, `currentNode`, that indicates nodes in the list, one by one, as the iterator is used to traverse the list. The reference `currentNode` is initialised with the head of the list. The method `hasNext()` in the class checks whether the reference `currentNode` has the value `null`, i.e. has traversed the whole list. The method `next()` in the class returns data in the current node referenced by the reference `currentNode`, and moves this reference to the next node in the list. The method `remove()` in the interface `Iterator<E>` does nothing but throw an exception to indicate that it cannot be used to remove nodes from the list. We will discuss exceptions in detail in Chapter 18. (Please fix the Xref.)

## Converting to an array

In Program 16.3 the method `listToArray()` at (7) traverses a linked list, and copies reference values of data in the nodes into an array. The method does *not* create copies of the data in the list. It uses a `for(:)` loop to traverse the list, since a linked list implements the interface `Iterable<E>` indirectly via the interface `ILinkedList<E>`. The array must be as large as the number of nodes in the list, and must be created before the method is called:

```
String[] strArray = new String[strList.numberOfNodes()];
strList.listToArray(strArray);
// strArray now contains reference values for the data in the list.
```

## Remarks on linked lists

Section 16.2 provides examples of using the class `LinkedList<E>`. A linked list can also be specialised to store data so that it is sorted in the list. The insertion operations must then compare data values in order to insert them at the correct position in the list.

There are other variants of linked lists. A *double linked list* consists of nodes where each node also has a reference to its predecessor. The last node in the list can refer to the first node in the list, and we thereby get a *circular list*.

The class `java.util.LinkedList<E>` in the Java standard library implements linked lists. This class provides functionality specified in the `java.util.List<E>` interface. In examples where we have used the class `java.util.ArrayList<E>`, we can safely use the class `LinkedList<E>`. The class `LinkedList<E>` is preferable when there are frequent insertions and deletions inside the list.

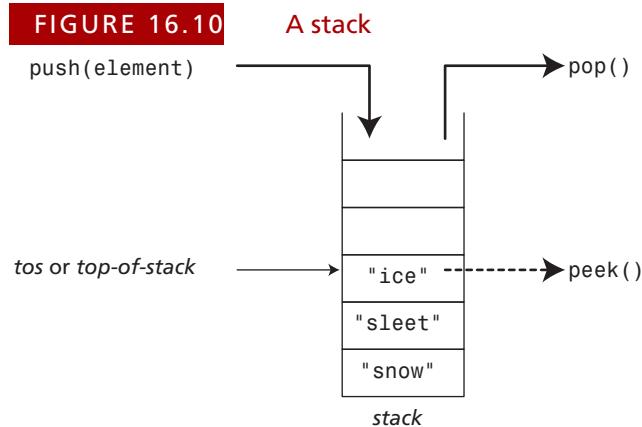
## 16.2 Other data structures: stacks and queues

### The data structure stack

A *stack* is a *LIFO (Last in, first out)* data structure, that is to say, the last value added to the stack, is the first value that is removed from the stack. A stack is illustrated in Figure 16.10, and the operations that can be performed on a stack are described in Table 16.1. The first two operations (`push()`, `pop()`) are mutators, and the remaining two (`peek()`, `isEmpty()`) are selectors. A stack has a height that is implicitly given by a refer-



ence called *tos* (*top of stack*), meaning that this reference always refers to the element on top of the stack.



**TABLE 16.1 Stack operations**

| Stack operation            | Description                                                                                                   |
|----------------------------|---------------------------------------------------------------------------------------------------------------|
| <code>push(element)</code> | Adds the <code>element</code> to the top of the stack.                                                        |
| <code>pop()</code>         | Returns the element that is currently on top of the stack. This element is removed from the stack.            |
| <code>peek()</code>        | Returns the element that is currently on top of the stack. This element is <i>not</i> removed from the stack. |
| <code>isEmpty()</code>     | Returns <code>true</code> if the stack is empty, otherwise <code>false</code> .                               |

There are many ways we can implement a stack. For example, we can use an array or a list. The class `java.util.Stack<E>` in the Java standard library implements a stack, based on a dynamic array (the class `java.util.Vector<E>`). Program 16.4 shows an implementation of a stack based on a *linked* list. We will take a closer look at this implementation.

In Program 16.4 we have used aggregation, and not inheritance, to define the class `StackByAggregation<E>`. This class has a field which is a reference to a linked list, and the stack operations are implemented by linked list operations. This is called *delegation*. A client uses the contract of a stack and does not need to know that a linked list is being used for this purpose. Note that other operations that can be performed on a linked list, cannot be executed on a stack in this implementation. A client cannot break the abstraction that a stack represents.

#### PROGRAM 16.4 Stack implementation by aggregation

```
class StackByAggregation<E> {
 // Field
```



```

private LinkedList<E> stackList; // (1)

// Constructor
public StackByAggregation() { // (2)
 stackList = new LinkedList<E>();
}

// Instance methods
public void push(E data) { // (3)
 stackList.insertAtHead(data);
}

public E pop() { // (4)
 if (empty())
 return null;
 return stackList.removeFromHead();
}

public E peek() { // (5)
 if (empty())
 return null;
 return stackList.first().getData();
}

public boolean empty() { // (6)
 return stackList.isEmpty();
}
}

```

---

Program 16.5 shows a client that uses a stack of strings. The program reads strings from the command line and adds them to a stack ((1), (2) and (3)). The elements are removed one by one in a loop at (4). Elements are usually processed after they are removed from the stack. In our example, uppercase letters in a string on the stack are converted to lowercase letters, and the result is printed to the terminal window. The program output shows that the string arguments are printed in the reverse order to the order in which they were inserted in the stack.

### PROGRAM 16.5 Stack client

```

public class StackClient {
 public static void main(String[] args) {
 // (1) Check if there are program arguments:
 if (args.length == 0) {
 System.out.println("Usage: java StackClient <argument list>");
 return;
 }

 // (2) Create a stack:

```



```

StackByAggregation<String> stack = new StackByAggregation<String>();

// (3) Push all program arguments (strings) on to the stack:
for (int i = 0; i < args.length; i++)
 stack.push(args[i]);

// (4) Pop all elements from the stack:
while (!stack.empty())
 System.out.print(stack.pop().toLowerCase() + " ");
System.out.println();
}
}

```

Running the program:

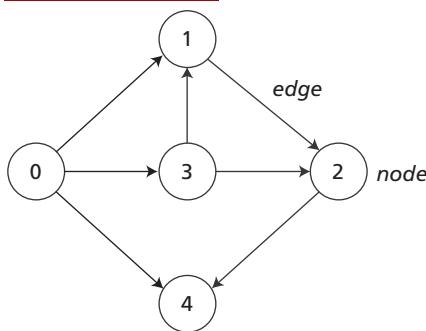
```
>java StackClient Tall STACKS topple easily
easily topple stacks tall
```

---

## Using stacks

There are many problems where a stack is useful. This is particularly true for recursive problems (see Chapter 17), where intermediate results in the algorithm can be stored on the stack and processed systematically from the top of the stack. Figure 16.11a shows a diagram which is called a *graph*. A graph has *nodes* and *edges*. The edges connect the nodes. The edges in Figure 16.11a are *directed*, that is to say, they have an arrow head at one end of the edge. We interpret the graph in Figure 16.11a as follows: each node is a city (which has a number indicated in the node), and an edge from a city indicates which city can be reached directly from this city. For example, from city number 3, we can directly reach city number 1 and city number 2.

There are many ways to implement a graph. One way is to use a two-dimensional array, as shown in Figure 16.11b. Both the row index and the column index refer to a city number. The number of rows (and the number of columns) are equal to the number of cities (i.e. nodes) in the graph. Each row represents cities that can be reached directly from the city represented by the row index. The array stores boolean values. The row with the index 3 has its elements with the indices [3][1] and [3][2] set to true, the same information that the graph represents for city number 3.


**FIGURE 16.11** Graph


(a) Graph

|     | [0] | [1] | [2] | [3] | [4] |
|-----|-----|-----|-----|-----|-----|
| [0] | F   | T   | F   | T   | T   |
| [1] | F   | F   | T   | F   | F   |
| [2] | F   | F   | F   | F   | T   |
| [3] | F   | T   | T   | F   | F   |
| [4] | F   | F   | F   | F   | F   |

T:true F:false

(b) Graph array

We will solve the following problem: which cities can be reached from a given city? With the starting point in city number 3, we see from the graph that we can reach cities with the numbers 1, 2 and 4. How did we figure that out? From city number 3 we can reach directly cities with numbers 1 and 2. From city number 2 we can directly reach city number 4. Therefore it is possible to reach city number 4 from city number 3 via city number 2. From city number 1 we can reach directly city number 2, but it is already in our count. From city number 4 we cannot go any further. The answer to which cities we can reach from city number 3, is cities with the numbers 1, 2 and 4. What we did was to find successively which cities can be reached directly from a given city, and repeat the same process for these cities.

The process above is illustrated in Figure 16.12. We use a stack *and* a set. The stack maintains cities we have yet to check in order to reach other cities directly, and the set maintains all cities we have reached so far in the process. Each step in the process does the following: (Please fix the pseudocode font.)

Pop the city from the top of the stack.

If this city is not in the set:

    Add this city to the set.

    Push all cities that can be reached directly from this city on to the stack.

The steps in the process, when the start city is city number 3, are shown in Figure 16.12. The process above is repeated until the stack is empty, i.e. we cannot reach any more cities.

**FIGURE 16.12**

Steps in finding cities reachable from city no. 3

| Step no. | Stack  | Set          |
|----------|--------|--------------|
| 1.       | <3>    | []           |
| 2.       | <1, 2> | [3]          |
| 3.       | <1, 4> | [3, 2]       |
| 4.       | <1>    | [3, 2, 4]    |
| 5.       | <2>    | [3, 2, 4, 1] |
| 6.       | <>     | [3, 2, 4, 1] |

Program 16.6 shows the implementation of the algorithm. The program creates the graph, at (1), and reads the number of the city to start in from the keyboard at (2). The stack



and the set are created at (3) and (4), respectively. We use the linked list implementation for the stack (Program 16.4) and the class `java.util.HashSet<E>` for the set. The stack is initialised with the start city at (5). The loop given at (6) implements the process we have outlined above. Finally, the set with all the cities that can be reached from the start city is printed.

The graph in Figure 16.11a is an example of a *directed acyclic graph* (a.k.a. *DAG*). It means that the graph is without *cycles*, i.e. we cannot find a *path* from a node and come back to the same node. Even if there were cycles in the graph, the solution in Program 16.6 will still work.

### PROGRAM 16.6 Graph traversal

```
import java.util.HashSet;
import java.util.Scanner;
import java.util.Set;

public class StackClient2 {
 public static void main(String[] args) {

 // (1) Create a city graph:
 boolean[][] cityGraph = {
 {false, true, false, true, true},
 {false, false, true, false, false},
 {false, false, false, false, true},
 {false, true, true, false, false},
 {false, false, false, false, false}
 };

 // (2) Read the city from the keyboard:
 System.out.print("Type the number of the city to start from [0-"
 + (cityGraph.length-1) + "]: ");
 Scanner keyboard = new Scanner(System.in);
 int startCityNum = keyboard.nextInt();
 if (startCityNum < 0 || startCityNum >= cityGraph.length) {
 System.out.print("Unknown city number: " + startCityNum);
 return;
 }

 // (3) Create a city stack:
 StackByAggregation<Integer> cityStack = new StackByAggregation<Integer>();

 // (4) Create a city set:
 Set<Integer> citySet = new HashSet<Integer>();

 // (5) Push start city on the stack:
 cityStack.push(startCityNum);
 // (6) Handle each city found on the stack:
 while (!cityStack.empty()) {
```



```

// Pop current city from the stack.
int currentCityNum = cityStack.pop();
// Check if the city has already been handled.
if (!citySet.contains(currentCityNum)) {
 // Insert current city to the city set.
 citySet.add(currentCityNum);
 // Push all cities that can be reached directly from
 // the current city on to the stack
 for (int j = 0; j < cityGraph[currentCityNum].length; j++) {
 if (cityGraph[currentCityNum][j])
 cityStack.push(j);
 }
}
// Remove start city from the city set, and print the city stack.
citySet.remove(startCityNum);
System.out.print("From city no. " + startCityNum +
 ", the following cities can be reached: " + citySet);
}
}

```

Running the program:

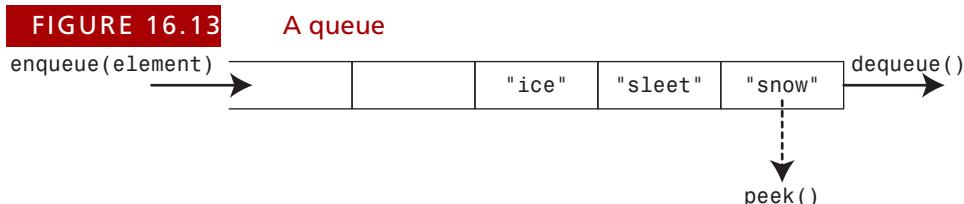
```

>java StackClient2
Type the number of the city to start from [0-4]: 3
From city no. 3, the following cities can be reached: [1, 2, 4]

```

## The data structure queue

As opposed to a stack, a *queue* is a *FIFO (First in, first out)* data structure, i.e. the first value added is the first value removed from the queue. A queue is illustrated in Figure 16.13, and the operations that can be performed on a queue are described in Table 16.2. The first two operations (`enqueue()`, `dequeue()`) are mutators, and the last two (`peek()`, `isEmpty()`) are selectors.



**TABLE 16.2 Queue operations**

| Queue operation               | Description                                |
|-------------------------------|--------------------------------------------|
| <code>enqueue(element)</code> | Adds the element to the back of the queue. |



| Queue operation | Description                                                                                                                 |
|-----------------|-----------------------------------------------------------------------------------------------------------------------------|
| dequeue()       | Returns the element that is currently the first element in the queue.<br>This element is removed from the queue.            |
| peek()          | Returns the element that is currently the first element in the queue.<br>This element is <i>not</i> removed from the queue. |
| isEmpty()       | Returns <code>true</code> if the queue is empty, otherwise <code>false</code> .                                             |

A queue can also be implemented with the help of an array or a list, but we have chosen to use a linked list. The behaviour of a queue can be directly mapped on a linked list. A queue implementation using an array can be less efficient, particularly if the elements in the array must be moved each time an element is added to or removed from the queue.

Program 16.7 shows a queue implementation that *extends* a linked list. The implementation uses inherited methods to implement queue operations. This is done intentionally to show how easy it is for a client to break the queue abstraction implemented by the class `StackByInheritance<E>` (see Program 16.8). A client can call any inherited method in the class `LinkedList<E>`, unless the method is overridden with a suitable implementation in the class `StackByInheritance<E>`. How to implement a queue by *aggregation* using a linked list is left as a programming exercise.

The Java standard library also provides an interface, `java.util.Queue<E>`, that specifies the contract for a queue. The class `java.util.LinkedList<E>` implements this interface, and can be used to create queues.

### PROGRAM 16.7 Queue implementation by inheritance

```
class QueueByInheritance<E> extends LinkedList<E> {

 public void enqueue(E data) { // (1)
 insertAtTail(data);
 }

 public E dequeue() { // (2)
 if (empty())
 return null;
 return removeFromHead();
 }

 public E peek() { // (3)
 if (empty())
 return null;
 return first().getData();
 }

 public boolean empty() { // (4)
 return isEmpty();
 }
}
```



}

## Using queues

Program 16.8 shows a client that uses a queue. Similar to the stack client from Program 16.5, the queue client reads strings from the command line and adds them to a queue ((1), (2) and (3)). The elements are removed one by one from the queue in the loop at (4) and printed to the terminal window.

The example also shows that we can call other linked list operations that the queue inherits in this implementation, and which break the queue abstraction. At (5) we see that we can insert elements at the *front* of queue, and the program output at (6) shows in which order the elements are removed from the queue. The class `StackByInheritance<E>` can avoid this problem by overriding methods from the class `LinkedList<E>` that it does not need. Implementation of these methods in the class `StackByInheritance<E>` can be an empty method body, {}.

### PROGRAM 16.8 Queue client

```
public class QueueClient {
 public static void main(String[] args) {
 // (1) Check if there are any program arguments:
 if (args.length == 0) {
 System.out.println("Usage: java QueueClient <argument list>");
 return;
 }
 // (2) Create a queue:
 QueueByInheritance<String> queue = new QueueByInheritance<String>();

 // (3) Enqueue all the program arguments (strings):
 for (int i = 0; i < args.length; i++)
 queue.enqueue(args[i]);

 // (4) Dequeue the queue for all elements:
 while (!queue.empty())
 System.out.print(queue.dequeue().toUpperCase() + " ");
 System.out.println();

 // (5) Can call methods that queues inherit and break
 // the queue abstraction:
 for (int i = 0; i < args.length; i++)
 queue.insertAtHead(args[i]);

 // (6) Print the queue elements:
 while (!queue.empty())
 System.out.print(queue.dequeue().toUpperCase() + " ");
 System.out.println();
 }
}
```



```
}
```

Running the program:

```
>java QueueClient You are now number two in the queue
YOU ARE NOW NUMBER TWO IN THE QUEUE
QUEUE THE IN TWO NUMBER NOW ARE YOU
```

### BEST PRACTICES

When you need a data structure, consult the Java API documentation first. The Java standard library provides implementation for very many data structures. You will seldom need to implement one yourself.

## 16.3 Review questions

1. What do we mean by the statement that a node in a linked list is *self referencing*?  
 A *self referencing* node is a node that has a field with a reference to another node of the same type.
2. Explain the result of executing the following statements on linked lists  
 (Program 16.3). Assume that the list operations are legal, and that the list has a sufficient number of nodes.
  - a** `p = p.getNext(); // p is a reference to a Node in the list.`
  - b** `p.setNext(p.getNext().getNext());`
  - c** `tail.setNext(head);`
 (a) results in the reference `p` now referring to the next node (i.e. successor) in the list.  
 (b) results in the successor to the reference `p` being removed.  
 (c) results in a *circular* linked list, where the tail is linked to the head.
3. Write the missing code so that data in all the nodes of a linked list (Program 16.3) is printed by the following `for(;;)` loop:
 

```
for (Node p = head; _____; _____)
 System.out.println(p.getData());
```

```
for (Node p = head; p != null; p = p.getNext())
 System.out.println(p.getData());
```
4. Write the missing code so that data in all the nodes of a linked list (Program 16.3) referenced by the following reference:



```
Iterable<Integer> list = new LinkedList<Integer>();
```

is printed by the following `for(:)` loop:

```
for (_____ data : _____)
 System.out.println(_____);
```

```
for (Integer data : list)
 System.out.println(data);
```

- 5.** What do acronyms LIFO and FIFO stand for? A stack is a \_\_\_\_\_ data structure, while a queue is a \_\_\_\_\_ data structure.

LIFO stands for "Last in, first out", and FIFO stands for "First in, first out". A stack is a *LIFO* data structure, while a queue is a *FIFO* data structure.

- 6.** Which data structure (stack or queue) is suitable in the following situations? Explain why.

- a** Organise loading and unloading of fish crates for transport.
- b** A portfolio with case documents, where the documents are processed in the order they are received.
- c** Serve customers, who join a line by a check-out counter.
  - (a) a stack, where the fish crates can be stacked for transport.
  - (b) a queue, so that case documents can be processed in the order they are received.
  - (c) a queue, so that customers can be served in the order they join the line.

## 16.4 Programming exercises

- 1.** Implement a class called `OrderedLinkedList<T>` where the elements in the list are ordered. The class `OrderedLinkedList<T>` must implement the following interface:

```
interface IOrderedLinkedList<T extends Comparable<T>> {
 // Order
 int INCREASING = 1;
 int DECREASING = -1;

 // Mutators
 void insertOrdered(T data_obj);
 T removeFromHead();
 T removeFromTail();
 boolean remove(T data_obj);

 // Selectors
 boolean isEmpty();
 OrderedNode<T> first();
 OrderedNode<T> last();
 int numOfNodes();
 OrderedNode<T> find(T data_obj);
```



```
 void listToArray(T[] array);
}
```

The class `OrderedLinkedList<T>` provides a constructor that lets clients specify whether the elements are in increasing or decreasing order. Data in the list implements the `Comparable<T>` interface. The method `insertOrdered()` inserts data in increasing or decreasing order, depending on how the list is created. The other methods correspond to methods in the `LinkedList<T>` class (Program 16.3).

The class `OrderedNode<T>` has the following declaration:

```
public class OrderedNode<T extends Comparable<T>> {
 /** Data in the node. */
 private T data;
 /** Reference to the next node. */
 private OrderedNode<T> next;

 // Constructor
 public OrderedNode(T data_obj, OrderedNode<T> nodeRef)
 { data = data_obj; next = nodeRef; }

 // Selectors
 public T getData() { return data; }
 public OrderedNode<T> getNext() { return next; }

 // Mutators
 public void setData(T obj) { data = obj; }
 public void setNext(OrderedNode<T> node) { next = node; }
}
```

- 2.** Test the class `OrderedLinkedList<T>` from Exercise 16.1 by implementing a client that reads strings from the command line. The client first stores the strings in an ordered linked list, and then prints the list. The program can be extended to allow the user to type a string in the terminal window, and give feedback whether the string is found in the list.
- 3.** Write a program that reads a string with parenthesis '()' and ')' from the command line, and determines whether the parenthesis are balanced, i.e. each left parenthesis has a corresponding right parenthesis, and vice versa. For example, parenthesis in the string "(((()))" are balanced, while this is not the case for the strings "()" and "())".

Write unit tests for the program.

- 4.** Implement a stack by inheritance based on a linked list (Program 16.3). A client should not be able to break the abstraction a stack represents.

Write unit tests for the stack implementation.

- 5.** Implement a queue by aggregation based on a linked list (Program 16.3).

Write unit tests for the queue implementation.



- 6.** Modify the `main()` method in the class `StableClient2` (Program 16.6) to the following method:

```
public static void findAllCities(int startCityNum) { ... }
```

i.e. the new method does not read the city number from the terminal window, but gets it as a parameter.

Write unit tests for the new method.

- 7.** Write unit tests for the implementation of linked lists in Program 16.3. Write at least one unit test for each method in the interface `ILinkedList<T>` (Program 16.2).

- 8.** Given the following classes `BiNode<E>` and `DoubleLinkedListIterator<E>`:

```
/**
 * A node contains data and references to the next and
 * the previous node.
 */
public class BiNode<E> {
 /** Data in the node. */
 private E data;
 /** Reference to the next node. */
 private BiNode<E> next;
 /** Reference to previous node. */
 private BiNode<E> previous;

 /**
 * Constructor initialises the current node with data and
 * sets references to the next and the previous node.
 * @param data_obj Data to be stored in the current node.
 * @param nextNodeRef Reference to the next node to set
 * for the current node.
 * @param previousNodeRef Reference to the previous node to set
 * for the current node.
 */
 public BiNode(E data_obj,
 BiNode<E> nextNodeRef, BiNode<E> previousNodeRef) {
 data = data_obj;
 next = nextNodeRef;
 previous = previousNodeRef;
 }

 /** Get data stored in the current node.
 * @return Data in the current node */
 public E getData() { return data; }

 /** Set data in the current node. */
 public void setData(E obj) { data = obj; }

 /** Get reference to the next node in the current node.
 * @return Reference to the next node */
 public BiNode<E> getNext() { return next; }
}
```



```

/** Set reference to the next node in the current node. */
public void setNext(BiNode<E> node) { next = node; }

/** Get reference to the previous node in the current node.
 * @return Reference to the previous node */
public BiNode<E> getPrevious() { return previous; }

/** Sets the reference to the previous node in the current node. */
public void setPrevious(BiNode<E> node) { previous = node; }
}

import java.util.Iterator;
/**
 * Iterator for double linked lists
 * Can only iterate in forward direction.
 */
public class DoubleLinkedListIterator<E> implements Iterator<E> {
 private BiNode<E> currentNode;

 public DoubleLinkedListIterator(IDoubleLinkedList<E> list) {
 currentNode = list.first();
 }

 public boolean hasNext() {
 return currentNode != null;
 }

 public E next() {
 E data = currentNode.getData();
 currentNode = currentNode.getNext();
 return data;
 }

 public void remove() {
 throw new UnsupportedOperationException();
 }
}

```

And the following interface `IDoubleLinkedList<E>` is also given:

```

interface IDoubleLinkedList<E> extends Iterable<E> {
 // Mutators
 void insertAtHead(E data_obj);
 void insertAtTail(E data_obj);
 E removeFromHead();
 E removeFromTail();
 boolean remove(E data_obj);

 // Selectors
}

```



```
boolean isEmpty();
int numNodes();
BiNode<E> first();
BiNode<E> last();
BiNode<E> find(E data_obj);

// Other methods
void listToArray(E[] array);
}
```

Complete the implementation of the class DoubleLinkedList<E> given below. Draw a class diagram before starting with the implementation. Write unit tests as you progress with the implementation.

```
import java.util.Iterator;
/**
 * This class implements double linked lists.
 */
public class DoubleLinkedList<E> implements IDoubleLinkedList<E> {

 // Fields
 /** Reference to the first node in the list. */
 private BiNode<E> head;

 /** Reference to the last node in the list. */
 private BiNode<E> tail;

 /** Number of nodes in the list. */
 private int numNodes;

 // ...
}
```

## Recursion

### LEARNING OBJECTIVES

By the end of this chapter, you will understand the following:

- What recursion is and how recursive techniques can be efficiently applied to solving certain kinds of problems.
- How recursion applies a divide-and-conquer approach to problem solving by combining the solution of partial problems to solve the overall problem.
- Why recursive algorithms must always specify at least one simple form of the problem that can be solved without recursion.
- That a recursive algorithm always has an iterative implementation.
- How binary search can be implemented using a recursive algorithm.
- How the game Towers of Hanoi can be solved using a recursive algorithm.
- Why the recursive algorithm Quicksort is an efficient sorting method.
- Potential pitfalls of recursive programming.

### INTRODUCTION

*Recursion* occurs when an operation uses itself as part of its execution. Recursion is used as a problem solving technique where a problem is divided into smaller versions of itself in such a manner that solving the partial problems leads to the overall problem being solved. In Java, recursive programming is implemented by methods that call themselves, i.e. execute recursive method calls. To gain insight into recursive programming, we look at several problems that have recursive solutions.



## 17.1 Recursion and iteration

### Factorials: an example from mathematics

Recursion is a well known concept from mathematics. We will look at the mathematical function *factorial*. The product of the expression where all integers from 1 to  $n$  are factors can be formulated as follows:

$$1! = 1$$

$$n! = n * (n-1)!$$

(fix formatting: indentation for lines with formulas)

This is a *recursive* definition, since it uses itself as part of the definition. The value of the expression  $5!$  is calculated as follows:  $5 * 4 * 3 * 2 * 1 = 120$ . In Java we can easily multiply all numbers from 1 to  $n$  in a loop statement, as illustrated in Program 17.1.

#### PROGRAM 17.1 Iterative calculation of factorial numbers

```
// Iterative calculation of factorial numbers.
class IterativeFactorial {
 public static void main(String[] args) {
 System.out.println("5! = " + factorial(5));
 }

 static int factorial(int n) {
 int value = 1;

 for (int i = 2; i <= n; i++)
 value *= i;

 return value;
 }
}
```

Program output:

5! = 120

Using a loop to compute the answer, as in Program 17.1, is called an *iterative* solution. It is also possible to perform the same calculation using a *recursive method* in Java. Program 17.2 shows an alternative definition of the `factorial()` method. The method `factorial()` in both Program 17.1 and Program 17.2 solves the exact same problem. The difference is that Program 17.2 uses a recursive method call to define the factorial function in the same manner as the mathematical definition.



## PROGRAM 17.2 Recursive calculation of factorial numbers

```
// Recursive calculation of factorial numbers.
import java.util.Scanner;
public class RecursiveFactorial {
 public static void main(String[] args) {
 System.out.print("Type an integer to calculate its factorial: ");
 Scanner keyboard = new Scanner(System.in);
 int n = keyboard.nextInt();

 System.out.println(n + "!" = " + factorial(n));
 }

 static int factorial(int n) {
 if (n == 1)
 return 1; // 1! = 1 (base case)

 return n * factorial(n - 1); // n! = n * (n-1)!
 }
}
```

Program output:

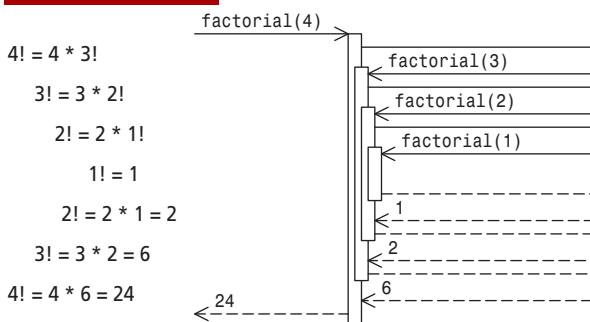
```
Type an integer to calculate its factorial: 12
12! = 479001600
```

---

### Using recursive method calls

One definition of the `factorial()` method is not necessarily better than the other, but it is easier to see that the method in Program 17.2 implements the mathematical definition. All methods that can be defined recursively, can also be defined using iteration, but in many cases the recursive definition can be more elegant.

Figure 17.1 illustrates the recursive method calls invoked in the calculation of 4! using the method definition in Program 17.2. The method accepts the parameter  $n$  and calculates  $n!$  by evaluating the expression  $n * (n-1)!$ . To obtain the value of  $(n-1)!$  the method calls itself with  $n-1$  as argument.


**FIGURE 17.1** Recursive method calls


The recursive calls continue until the evaluation of  $1!$  is reached. This is a *base case* where the method just returns the value 1 without performing further recursive calls. The method calls are *nested* until the base case is reached. The method call for the base call returns the value 1, and subsequently all method calls finish execution one by one and return a value. Each nested method call returns a value that contributes to the factorial in the enclosing method call. Each call to the `factorial()` method then successively returns the factorial of its argument. More details on method execution can be found in Section 10.2 on page 272.

Since the data type `int` can only store integers of a limited range, the `factorial()` method defined in the previous examples will return incorrect results for integers greater than 16. However, if the type of the result is `long`, the program can correctly calculate the factorial for numbers up to and including 20.

## 17.2 Designing recursive algorithms

A *recursive algorithm* solves a task by dividing it into smaller subtasks and uses the same algorithm to solve each subtask until the task becomes trivial. A recursive method can implement a recursive algorithm. The parameters passed to a recursive method describe the extent of the task to be solved. When the method calls itself recursively, it must make sure that the arguments provided in the next recursive call describe a task that is smaller than itself.

The design of all recursive methods is based on handling the following two main categories of cases:

- 1 General cases:** The parameters to the method describe a task that can be divided into smaller subtasks. The method divides the task into one or more subtasks of the same kind as the overall task. These subtasks are then solved by calling the method recursively, and the solution of the overall task is expressed by means of the solutions to the subtasks.
- 2 Base cases:** The parameters to the method describe a simple task that is so trivial that it need not be divided further, and can therefore be solved directly without any further recursive calls. A recursive method can contain more than one such base case.



The general cases are necessary for recursive method calls to occur at all, and the base cases are necessary to ensure that the recursive calls stop at some point.

### BEST PRACTICES

When developing recursive algorithms, focus on identifying the general cases and the base cases.

## 17.3 Infinite recursion

The only thing that changes from one recursive call to the next are the parameter values that specify which subtask the method should solve next. For each recursive method call, the arguments get one step closer to the arguments that constitute a base case, to ensure that the tasks *converge* to a base case. The *recursion depth* is the number of nested method calls that are necessary before a base case is reached.

If a recursive method does not converge towards a base case, the recursion will never stop. Such a recursion is often called *infinite recursion*. The reason for infinite recursion is invariably a logical programming error, and a program with infinite recursion will sooner or later crash during execution, as the JVM cannot handle infinite nesting of method calls. The following two recursive methods illustrate programming errors leading to infinite recursion:

```
static void infiniteRecursion() {
 infiniteRecursion();
}

static int flip(int value) {
 if (value == 0)
 return 0;
 return value + flip(value-2);
}
```

The `infiniteRecursion()` method will always result in infinite recursion, since it lacks a base case to end the recursion. The `flip()` method results in infinite recursion if an odd number, e.g. 9, is given as an argument in the method call.

A few rules of thumb to avoid infinite recursion:

- For each recursive call, the method must converge towards the base case. Consequently, the amount of work described by the arguments to the method must be reduced for each recursion level. The end of recursive calls is reached when the arguments describe a base case.
- A recursive call must never result in the method being called again with the same arguments.



## BEST PRACTICES

Ensure that there is always at least one trivial form of the problem, i.e. a base case, that can be solved directly without any further use of recursive method calls.

## 17.4 Recursive binary search

Section 9.6 on page 253([reduce space after section number](#)) shows an iterative algorithm for binary search in sorted arrays. At each step the search method splits the search area into two parts. After eliminating one half of the search area, a smaller subtask of the same kind remains. Recursion is well suited for solving such subtasks. Binary search can be described by the following recursive algorithm, where *search-area* is the array segment to search in, and *key* is the integer we want to find in the array:([Please fix pseudocode font](#))

```
Operation binarysearch(search-area, key):
 If the search-area is empty:
 Return key-does-not-exist.

 Find the middle-element in the search-area.

 If the middle-element == key:
 Return position-of-key.

 If the key < middle-element:
 Return binarysearch(search-area-before-middle-element, key).
 Else:
 Return binarysearch(search-area-after-middle-element, key).
```

Program 17.3 shows a recursive binary search algorithm that finds integers in an array containing all prime numbers from 1 to 100. By performing recursive binary search in this array, the program can find out whether a given integer is a prime number, and if so, which rank it has in the sequence of primes. Note that the `binarySearch()` method returns the *index* of the prime number in the array, and the position is *index* + 1.

### PROGRAM 17.3    Recursive binary search

```
import java.util.Scanner;

// Recursive binary search for prime numbers.
public class FindRecursively {
 // Constants that signals that a number does not exist in the array.
 static final int NOT_THERE = -1;

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



```

// Array containing prime numbers between 1 and 100 in ascending order:
int[] primeNumbers = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
 53, 59, 61, 67, 71, 73, 79, 83, 89, 97};

System.out.print("Type an integer between 1 and 100: ");
Scanner keyboard = new Scanner(System.in);
int number = keyboard.nextInt();
if ((number < 1) || (number > 100)) {
 System.out.println("The number is not between 1 and 100: " + number);
 return ;
}

int primeNumberIndex = find(primeNumbers, number);
if (primeNumberIndex == NOT_THERE) {
 System.out.println(number + " is not a prime number.");
} else {
 System.out.println(number + " is a prime with position "
 + (primeNumberIndex + 1) + " in the sequence of prime numbers.");
}
}

static int find(int[] sortedArray, int key) {
 return binarySearch(sortedArray, key, 0, sortedArray.length - 1);
}

static int binarySearch(int[] sortedArray, int key, int lower, int upper) {
 if (lower > upper) {
 /* base case, the parameters specify an empty array segment,
 i.e. the key does not exist in the array. */
 return NOT_THERE;
 }

 int middle = (lower + upper) / 2;
 int middleNumber = sortedArray[middle];

 if (key == middleNumber) {
 return middle; // Another base case. The key was found.
 }

 // General case: search in either lower or upper array segment.
 if (key < middleNumber) {
 return binarySearch(sortedArray, key, lower, middle - 1);
 } else {
 return binarySearch(sortedArray, key, middle + 1, upper);
 }
}
}

```



### Program output:

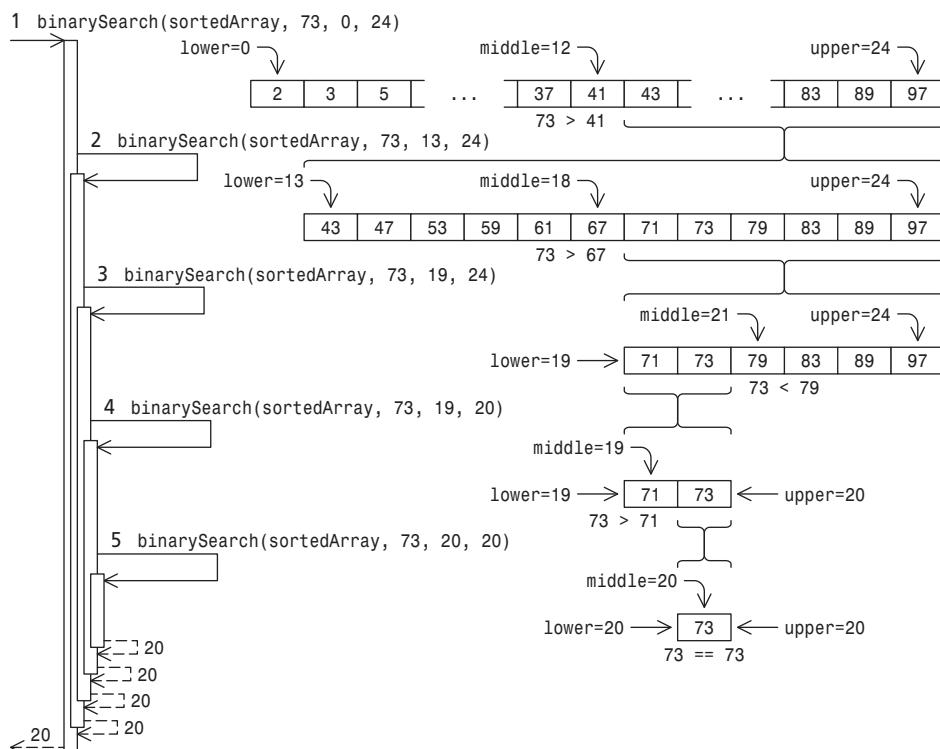
Type an integer between 1 and 100: 73

73 is a prime with position 21 in the sequence of prime numbers.

The `binarySearch()` method has four parameters. The first two arguments specify the sorted array and the key we are searching for in the array. These two parameters do not change in any of the recursive calls. The parameters `lower` and `upper`, that specify limits of the search area, are changed so that the size of the search area is halved for each recursive call.

Figure 17.2 shows how the search for the prime number 73 proceeds during execution. The figure shows how the indices `lower` and `upper`, that designate the lower and the upper limits of the search area, are approaching each other and therefore converging towards the position of the key in the array. At a recursion depth of 5, the index of the prime number 73 found in the array `sortedArray` that contains 25 elements.

**FIGURE 17.2** Binary search for the prime number 73



Since the search area is halved in each step, a binary search in an array of  $n$  elements use maximum  $\log_2(n)$  recursive calls. An array holding all prime numbers up to 821641 will contain 65536 elements, and a search in this array will only require up to 16 steps using the binary search algorithm.

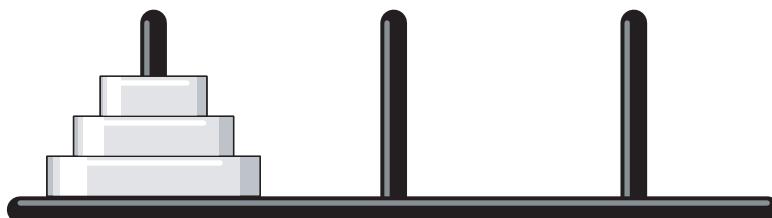


The recursive binary search algorithm is an example of an algorithm that uses *tail recursion*. The last statement that is executed by the recursive method is a new recursive call. Java handles such recursive calls like any other method calls, while in some programming languages the new call replaces the current call. The result is the same, but the execution can be more efficient since there is no nesting of method calls.

## 17.5 Towers of Hanoi

In 1883, the French mathematician Edouard Lucas invented a game he called *Towers of Hanoi*. The game consists of three pegs and a set of rings of different sizes that can be placed on the pegs. The game starts with all rings being stacked on top of each other in a pyramid shape. Figure 17.3 shows the initial setup for a game with three rings.

**FIGURE 17.3** Towers in the Hanoi game



The objective of the game is to move all rings from one peg to another, recreating the tower at the target peg. The game has two rules:

- 1 A ring cannot be placed on top of a smaller ring.
- 2 Only *one* ring can be moved at a time.

The game can be played with any number of rings. Figure 17.3 shows the initial setup in a game with three rings. To solve the task in this game all three rings must be moved to another peg without breaking any of the rules stated above. The number of rings determines how complex the solution is. If only *one* ring is used, the game is trivial, and the task is solved by moving the ring directly to the *target* peg. This solution is shown in Figure 17.4.

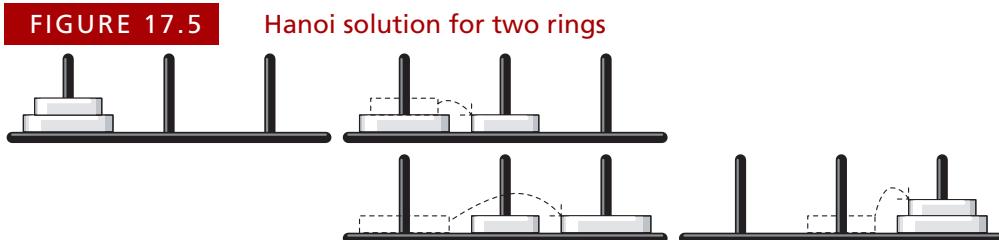
**FIGURE 17.4** Hanoi solution for one ring



When moving more than one ring, the trick is to place the rings on the target peg in the right order. Rule 1 states that the bottom ring on the *from* peg must be placed on the bottom of the target peg. To be able to move the bottom ring, however, all rings on top of it must be moved first.



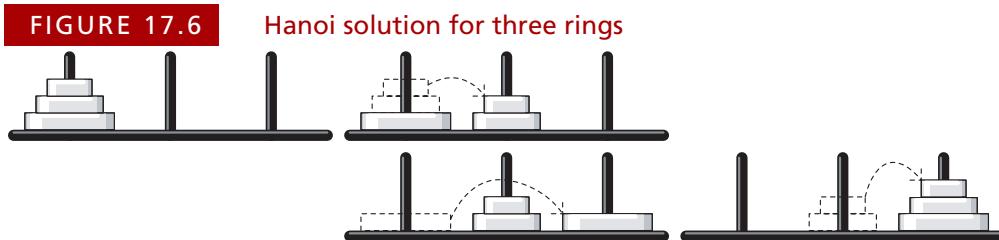
Figure 17.5 shows the solution for a game with two rings. To move the bigger ring, we first need to move the smaller ring to the temporary peg. After the bigger ring is placed on the target peg, the smaller ring can be placed on top of it.



As the number of rings increases, we see that the solution of the game starts to take on a certain pattern. It can be shown that the solution of the Towers of Hanoi game can be described by a recursive algorithm. The solution for moving a tower of rings from one peg to another can be formulated as follows:

- 1 Move all rings, except the bottom one, to the temporary peg.
- 2 Move the bottom ring to the target peg.
- 3 Move all the rings from the temporary peg to the target peg.

Steps 1 and 3 can be described as a smaller Towers of Hanoi game, where the objective is to move a tower with one less ring to another peg. Figure 17.6 shows the solution for a game with three rings. How the two smaller rings are moved to the temporary peg is not shown in the figure, but it is assumed that the solution for moving two rings from Figure 17.5. is used



### Recursive solution

Program 17.4 uses recursion to print all the steps necessary to solve a game with four rings. To move a tower of rings from one peg to another, the `moveRings()` methods uses two recursive method calls: the first call to move all rings, except the largest one at the bottom, to the temporary peg, and the second call to move all rings, that were moved to the temporary peg, to the target peg. Between the recursive calls, the bottom ring is moved to the target peg.



## PROGRAM 17.4 Recursive Tower of Hanoi solution

```
// Towers of Hanoi - Recursive version.
public class Hanoi {
 public static void main(String[] args) {

 int numRings = 4;

 System.out.println("Moving " + numRings + " rings:");
 /* Move all rings from peg 1 to peg 3 by using peg 2 as
 temporary storage. */
 moveRings(numRings, 1, 3, 2);
 }

 static void moveRings(int count, int from, int to, int tmp) {
 if (count < 1) {
 // Base case: No rings to move.
 return ;
 }

 /* To move the bottom ring, we first have to move the count-1
 rings on top of this ring away from the current peg. Let this
 subtask use the "to" peg as temporary storage: */
 moveRings(count - 1, from, tmp, to);

 // The bottom ring can now be moved:
 System.out.println(from + " ---> " + to);

 /* The topmost count-1 rings on the "tmp" peg can now be moved to
 the "to" peg. Let this subtask use the now vacant "from" peg
 as temporary storage: */
 moveRings(count - 1, tmp, to, from);
 }
}
```

Program output:

```
Moving 4 rings:
1 ---> 2
1 ---> 3
2 ---> 3
1 ---> 2
3 ---> 1
3 ---> 2
1 ---> 2
1 ---> 3
2 ---> 3
2 ---> 1
3 ---> 1
2 ---> 3
```



```
1 ---> 2
1 ---> 3
2 ---> 3
```

---

The Hanoi algorithm differs from the recursive algorithms we have seen so far, in that it uses *two* recursive calls at each recursion level.

The `moveRings()` method has the following parameters:

- `count`: The number of rings in the tower that are to be moved in this subtask.
- `from`: The number identifying the peg containing the tower to be moved.
- `to`: The number identifying the peg the tower will be moved to.
- `tmp`: The number identifying the peg that can be used to store rings temporarily.

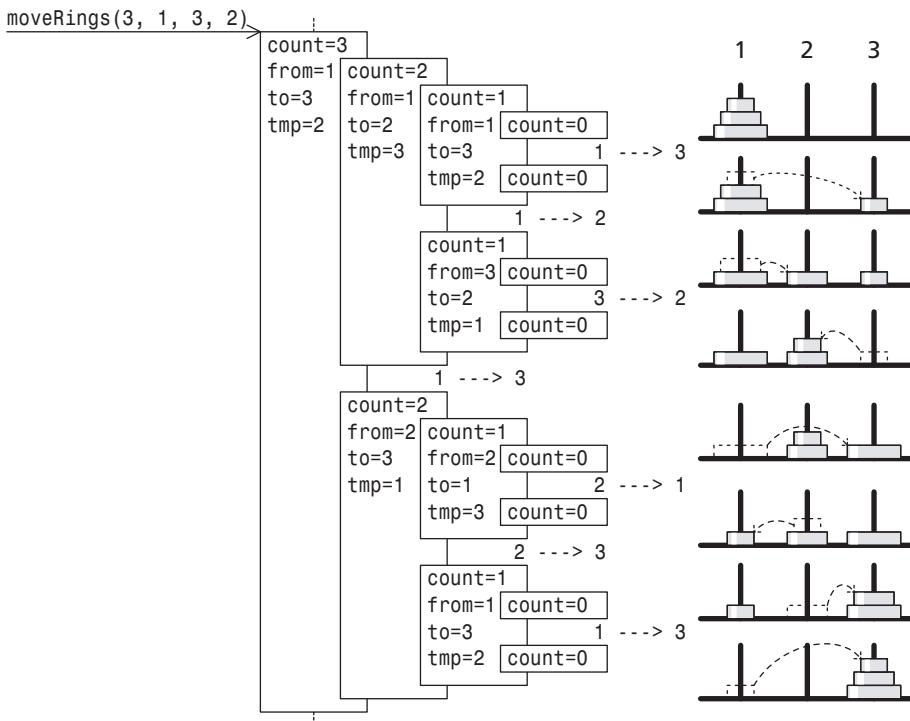
Each execution of the method has its own version of these parameters, that are maintained during recursion. When a recursive method call returns, the parameters will have the values they had before this call was made. Figure 17.7 shows the values of the local parameters when moving a tower with three rings.

During method calls all local variables, including parameters, are stored on an implicit stack. Figure 17.7 illustrates how this stack grows and shrinks during method calls. This stack is often called a *program stack*. Each method execution has its own *stack frame* that contains the local variables for the method call. A stack frame is created on the program stack when a method is called, and removed when the method returns (see also Section 10.2 on page 272). That is why recursive algorithms incur a performance penalty because of all the method calls.

The most common reason for programs crashing during infinite recursion, is that the program stack cannot grow indefinitely (often called *stack overflow*).

**FIGURE 17.7**

Method execution in recursive Hanoi



### BEST PRACTICES

Test your base cases carefully to see that they are executed, in order to avoid stack overflow and its consequences during execution.

## Iterative solution

As mentioned earlier, any recursive algorithm can also be expressed as an iterative algorithm. Usually, the recursive algorithm will be easier to understand, but pays a performance penalty because of all the recursive method calls. To illustrate the point, an iterative version of Towers of Hanoi is shown in Program 17.5. The implementation details for the iterative solution are too complex to be presented here. This implementation is a little more efficient than the recursive version, but it is also far more difficult to interpret. The iterative version also numbers the pegs differently, from 0 to 2, instead of the more intuitive numbering from 1 to 3 used in Program 17.4. Figuring out the intricacies of the iterative solution are left as an exercise for the inquisitive reader.

### PROGRAM 17.5 Iterative Tower of Hanoi

```
// Towers of Hanoi - Iterative version.
public class IterativeHanoi {
```



```

public static void main(String[] args) {
 int numRings = 4;
 System.out.println("Moving " + numRings + " rings:");

 for (int step = 1; step < (1 << numRings); step++)
 System.out.println((step&step - 1) % 3 + " ---> " +
 ((step | step - 1) + 1) % 3);
}
}

```

Program output:

```

Moving 4 rings:
0 ---> 2
0 ---> 1
2 ---> 1
0 ---> 2
1 ---> 0
1 ---> 2
0 ---> 2
0 ---> 1
2 ---> 1
2 ---> 0
1 ---> 0
2 ---> 1
0 ---> 2
0 ---> 1
2 ---> 1

```

## 17.6 Quicksort: a recursive sorting algorithm

One of the more efficient algorithms for sorting elements in an array is called *Quicksort*. This recursive algorithm divides the sorting task into two subtasks, and uses a recursive call to solve each one. This procedure is often called *divide and conquer*.

Most of the work in the Quicksort algorithm consists of dividing the task. Each subtask concerns sorting a subarray (i.e. a segment of an array). The subarray is partitioned into two segments so that each segment can be sorted individually. The partitioning of the subarray is done in such a way that the largest element in the first segment is smaller than all the elements in the second segment. The process is then repeated on each of the two segments.

For example, the sequence of numbers [4, 3, 8, 1, 5, 7, 0, 9, 2] can be partitioned into two segments, where one segment [3, 1, 0, 2] holds values smaller than 4 only, while the second segment [8, 5, 7, 9] only holds elements greater than 4. The value 4 is called the *pivot element*, or simply *pivot*. (Please fix pseudocode font below.)



Algorithm for Quicksort:

```
Operation quicksort(segment):
 If segment is empty:
 Return
 Perform partitioning of segment into two parts: part1 and part2
 quicksort(part1)
 quicksort(part2)
```

Algorithm for partitioning:

```
Operation performPartitioning(segment):
 pivotElement = a chosen element in the segment
 Swap the elements in the segment so that:
 All elements less than pivotElement are placed before pivotElement
 All elements greater than pivotElement are placed after pivotElement
```

Program 17.6 sorts an array of strings, using the Quicksort algorithm. The `sort()` method has the same formal type parameters as the `binsearch()` method from subsection *Generic methods* on page 496, and can sort arrays of all types implementing the `Comparable` interface. The `sort()` method is applied to an array of `String` objects that are `Comparable`. It checks for the base case, performs the partition if necessary, and then calls itself twice to sort each part of the segment.

Figure 17.8 shows how the `performPartition()` method partitions an array into two segments, using the first element in the array as the pivot element.

### PROGRAM 17.6 Sorting with the Quicksort algorithm

```
import java.util.Arrays;

// Recursive sorting algorithm - Quicksort.
public class Quicksort {
 public static void main(String[] args) {
 String[] words = {
 "odd", "dub", "let", "etc", "his", "two", "lot", "who", "tip",
 "kit", "rat", "all", "ham", "lag", "bet", "got", "add", "pop",
 "god", "elk", "cat", "pig", "gas", "hat", "egg", "bun", "hog",
 "car", "sic", "bat", "one", "bag", "her", "ill", "fit", "pee",
 "for", "shy", "nob", "pie", "sun", "mac", "fun", "mad", "was",
 "sob", "mom", "gel", "new", "dam", "sat", "too", "sad", "old",
 "top", "bee", "ink", "put", "sea", "tie", "nor", "pea", "bit",
 "eve", "has", "not", "can", "dig", "hoe", "nod", "wet", "cab",
 "big", "nil", "saw", "pin", "she", "rip", "get", "yet", "see"
 };
 sort(words);
 System.out.println(Arrays.toString(words));
 }

 /** Sorts an array with Comparable elements. */
 public static <T extends Comparable<T>> void sort(T[] elementArray) {
```



```

 sort(elementArray, 0, elementArray.length - 1);
 }

/** Sorts a segment (i.e. a subarray). */
private static <T extends Comparable<T>> void
sort(T[] segment, int first, int last) {

 /* Base case: 0 or one element in segment, no sorting necessary. */
 if (first >= last) {
 return;
 }

 // General case: partition the segment.
 int pivotPosition = performPartition(segment, first, last);

 // Recursive call to sort elements less than pivot element:
 sort(segment, first, pivotPosition-1);

 // Recursive call to sort elements greater than pivot element:
 sort(segment, pivotPosition+1, last);
}

/**
 * Partitions a segment.
 * Returns the index where the pivot element is to be found in the segment,
 * and hence the boundaries of the two smaller segments.
 */
private static <T extends Comparable<T>> int
performPartition(T[] segment, int first, int last) {
 // Choose the first element as pivot element:
 T pivotElement = segment[first];

 // The unpartitioned elements are always between
 // the lower and the upper indices, inclusive:
 int lower = first + 1;
 int upper = last;

 /* Repeat the loop until there are no more elements left to partition.
 The array has the following subarrays while the loop executes:
 first+1 .. lower-1: Elements with value less than the pivot element.
 lower .. upper: Elements not yet partitioned.
 upper+1 .. last: Elements with values greater than the pivot element.
 */
 while (lower <= upper) {
 T elementToPlace = segment[lower];
 boolean placedOK = elementToPlace.compareTo(pivotElement) <= 0;
 if (placedOK) {
 // Element is placed correctly at the lower index.
 // Decrease the unpartitioned segment size by increasing the lower index.
 ++lower;
 }
 }
}

```



```

} else {
 // Swap elements at the ends of the unpartitioned segment.
 segment[lower] = segment[upper];
 segment[upper] = elementToPlace;
 // Element is now placed correctly at the upper index.
 // Decrease the unpartitioned segment size by decreasing the upper index.
 --upper;
}
}

/* The segment is exhausted.
 * Place the pivot element correctly to mark the boundary between
 * elements smaller than the pivot element and
 * those that are greater than the pivot element.
 */
int pivotPosition = lower - 1;
segment[first] = segment[pivotPosition];
segment[pivotPosition] = pivotElement;
return pivotPosition;
}
}

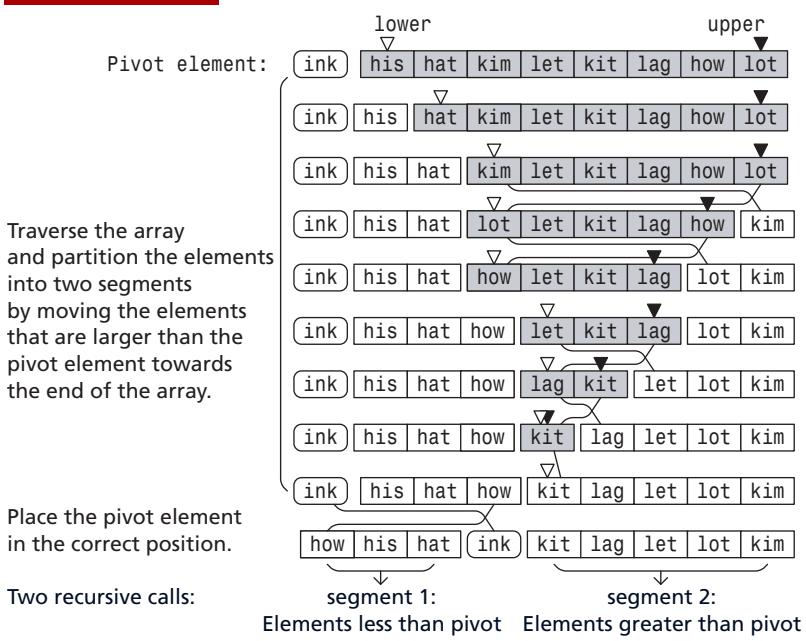
```

Program output:

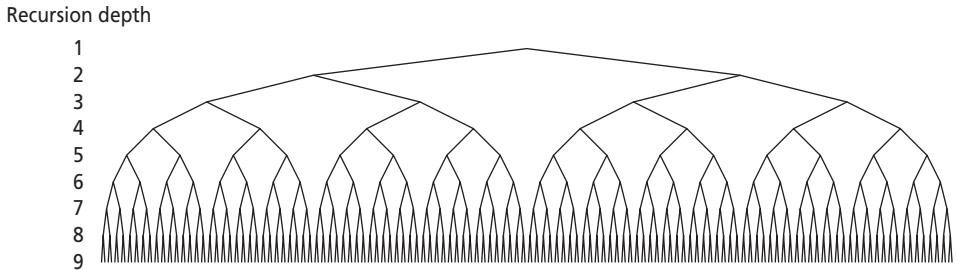
---

```
[add, all, bag, bat, bee, bet, big, bit, bun, cab, can, car, cat, dam, dig, dub,
egg, elk, etc, eve, fit, for, fun, gas, gel, get, god, got, ham, has, hat, her,
his, hoe, hog, ill, ink, kit, lag, let, lot, mac, mad, mom, new, nil, nob, nod,
nor, not, odd, old, one, pea, pee, pie, pig, pin, pop, put, rat, rip, sad, sat,
saw, sea, see, she, shy, sic, sob, sun, tie, tip, too, top, two, was, wet, who,
yet]
```

---


**FIGURE 17.8** Partitioning the array


Note that for recursive algorithms like Quicksort, where the task is divided into two recursive method calls, the recursion depth is not equal to the number of times the method is called. The Quicksort algorithm executes a recursive call for each of the two subtasks after it has partitioned a task. This results in the total number of method calls being doubled for each recursion level. Figure 17.9 illustrates how the number of recursive method calls increases exponentially with the recursion depth. A recursion depth of 9 results in 511 ( $2^9 - 1$ ) method calls.

**FIGURE 17.9** Exponential increase in the number of method calls




## BEST PRACTICES

Be aware of recursive algorithms requiring multiple recursive calls at each recursion level. The overhead imposed by the recursive method calls may have a major impact on the performance of the algorithm.

## 17.7 Fibonacci series: an example of iterative solution

It is important to be aware of the exponential increase in method calls when developing recursive algorithms. Let us review a function where a recursive method is not a good solution. The *Fibonacci series* is a number sequence where the next number in the sequence is the sum of the two previous numbers. The sequence is defined as follows:

$$f_1 = 1$$

$$f_2 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$
 (fix formatting: indentation for lines with formulas)

Program 17.7 shows a naive implementation of a `fibonacci()` method to find the 45th number in the Fibonacci series.

### PROGRAM 17.7 Naive recursive calculation of Fibonacci numbers

```
// Recursive Fibonacci calculation - naive version.
public class NaiveFibonacciTimed {
 public static void main(String[] args) {
 long startTime = System.nanoTime(); // Current time in nanoseconds (1)
 System.out.println("The 45th number in the Fibonacci sequence is " +
 fibonacci(45)); // The code to be timed.
 System.out.println("Execution time: " +
 (System.nanoTime()-startTime) + "ns"); // (2)
 }

 static int fibonacci(int n) {
 if (n < 3)
 return 1;

 return fibonacci(n - 2) + fibonacci(n - 1);
 }
}
```

Program output:

The 45th number in the Fibonacci sequence is 1134903170



Execution time: 12332959636ns

This naive implementation in Program 17.7 will result in 226 906 339 recursive calls. The calculation took more than 12 seconds (1 second =  $10^9$  nanoseconds). The execution time can of course vary due to various factors (for example, CPU speed), but we can get a fairly good idea about the execution time of a program by using the setup ((1) and (2)) shown in Program 17.7.

The problem with the recursive implementation of the `fibonacci()` method is that it does a vast amount of duplicate work. The calculation of  $f_n$  will demand that  $f_{n-1}$  and  $f_{n-2}$  are calculated. Both of these require, for instance, the calculation of  $f_{n-3}$ , and this calculation will be performed separately for both  $f_{n-1}$  and  $f_{n-2}$ . The amount of duplicate work increases exponentially with the Fibonacci number that is calculated. Table 17.1 shows how the number of method calls increases as larger Fibonacci numbers are calculated.

**TABLE 17.1 Number of methods calls when calculating Fibonacci numbers**

| Fibonacci number | Number of method calls |
|------------------|------------------------|
| $f_3$            | 3                      |
| $f_4$            | 5                      |
| $f_5$            | 9                      |
| $f_6$            | 15                     |
| $f_7$            | 25                     |
| $f_8$            | 41                     |

To calculate  $f_n$  it is necessary to calculate all numbers from  $f_1$  to  $f_{n-1}$ . In this case it is more efficient to use an iterative method to calculate the values in the Fibonacci sequence up to  $f_n$  sequentially. Program 17.8 shows an iterative implementation that finds the 45th element by generating the numbers in the Fibonacci series up to and including  $f_{45}$ . When running this version of the program, no recursive methods call are executed, and the program execution takes less than half a millisecond (1 millisecond =  $10^6$  nanosecond).

### PROGRAM 17.8 Iterative calculation of Fibonacci numbers

```
// Iterative Fibonacci calculation.
public class IterativeFibonacciTimed {
 public static void main(String[] args) {
 long startTime = System.nanoTime(); // (1)
 System.out.println("The 45th number in the Fibonacci sequence is " +
 fibonacci(45)); // The code to be timed.
 System.out.println("Execution time: " +
 (System.nanoTime()-startTime) + "ns"); // (2)
 }
}
```



```

static int fibonacci(int n) {
 int nextToLast = 1;
 int last = 1;
 for (int i = 3; i <= n; i++) {
 // Calculate a new Fibonacci number.
 int next = last + nextToLast;

 // Store the last two Fibonacci numbers in the series so far:
 nextToLast = last;
 last = next;
 }
 return last;
}

```

Program output:

The 45th number in the Fibonacci sequence is 1134903170

Execution time: 312331ns

When considering efficiency, the Quicksort algorithm is an example where recursion provides the best solution, while in the case of calculating Fibonacci numbers, the iterative solution is better than the recursive one. One of the challenges of programming is to find the most suitable algorithm for a given problem.

#### BEST PRACTICES

If a problem lends itself naturally to a recursive algorithm, implement a recursive solution first. Then develop an iterative solution, if and only if the performance of the recursive solution is unacceptable.

## 17.8 Review questions

1. A recursive method is a method that executes method calls on \_\_\_\_\_.

**A recursive method is a method that executes method calls on *itself*.**

2. What will the following program print during execution?

```

class Word {
 public static void main(String[] args) {
 printOut(3);
 }
 static void printOut(int i) {
 if (i == 1) {

```



```
 System.out.println("ian");
 } else {
 System.out.print("bar");
 printOut(i - 1);
 }
}
}
```

The program will print the word *barbarian* by means of recursive method calls. The method call `printOut(3)` will print "bar" and call `printOut(2)`, which will also print "bar", and then call `printOut(1)` recursively. The last method call is a base case which will print "ian".

- 3.** What are the main advantages and disadvantages of iteration as opposed to recursion?

An iterative solution is usually more efficient than a recursive solution, since recursive method calls are avoided. A recursive solution is often more elegant and easier to understand than an iterative solution. An iterative solution must often implement a stack of its own to store temporary results, since it cannot use the program stack for this purpose.

- 4.** If the tasks in a recursive method do not converge towards \_\_\_\_\_, the recursion can be \_\_\_\_\_.

If the tasks in a recursive method do not converge towards a *base case*, the recursion can be *infinite*.

- 5.** Which mathematical operation is performed by the following method?

```
static int ts(int num) {
 if (num < 10) {
 return num;
 } else {
 return num % 10 + ts(num/10);
 }
}
```

The method `ts()` calculates the sum of the digits in the number given as parameter. For example, the value returned from the method call `ts(1234)` is 10, since  $1 + 2 + 3 + 4 = 10$ . Each recursive call adds the least significant digit to the sum of digits for the remaining digits in the number.

- 6.** Which statements are true?

- a** Recursion can always be replaced by iteration.
- b** The local variables in a method body are shared by all recursive method calls that are executed by this method.
- c** The recursion depth of a recursive method call corresponds to the largest number of stack frames that the method calls will use at any given point during execution.
- d** All recursive methods must contain a base case to stop the recursion.

(a), (c) and (d)



## 17.9 Programming exercises

1. Write an iterative version of the method given in Question 17.5.
2. Write a recursive method that returns the boolean value `true` if the digits of the integer given as parameter are sorted in ascending order from left to right.
  - Identify the base cases of the method.
  - What determines the recursion depth of the method during execution?
  - Is there a possibility of infinite recursion?
3. Write a method that calculates the *greatest common divisor*, GCD. The method `gcdn,m` can be defined as follows:

If  $m = n$ :  $\text{gcd}_{m,n} = m$   
 If  $m > n$ :  $\text{gcd}_{m,n} = \text{gcd}_{(m-n),n}$   
 If  $m < n$ :  $\text{gcd}_{m,n} = \text{gcd}_{m,(n-m)}$

- a Write `gcd()` as a recursive method.
- b Write `gcd()` as an iterative method.

4. This programming exercise deals with printing a list of celestial bodies that orbit other celestial bodies.
  - a Create a class `CelestialBody` that represents bodies in the solar system. Each body has a name and a collection of other bodies that orbit this body. We can say that a celestial body is owned by the celestial body which it orbits.
  - b Use the class `CelestialBody` and write a method that creates celestial bodies based on the information given in the array below. Insert the objects into a hash map, where the key is the body name.

```
String[][] celestialBody = {
 // Body name orbits
 { "Deimos", "Mars" },
 { "Earth", "Sun" },
 { "Mars", "Sun" },
 { "Mercury", "Sun" },
 { "Moon", "Earth" },
 { "Phobos", "Mars" },
 { "Sun", null },
 { "Venus", "Sun" }
};
```

- c Write a recursive method that uses the `CelestialBody` objects to generate the following report:

```
Sun
 Earth
 Moon
 Mars
 Deimos
 Phobos
```



Mercury

Venus

5. In 1985 the mathematician Ross Honsberger published a new way of calculating the Fibonacci number  $f_n$ :

For  $n$  that are odd numbers:  $f_n = (f_{((n+1)/2)})^2 + (f_{((n-1)/2)})^2$

For  $n$  that are even numbers:  $f_n = (f_{(n/2+1)})^2 - (f_{(n/2-1)})^2$

Use these formulas to write a program that recursively calculates Fibonacci numbers more efficiently than the iterative solution in Program 17.8.

6. Write a program that linearises nested arrays and prints the result. Let the program have the following `main()` method:

```
public static void main(String[] args) {
 Character spacing = ' ';
 Object _2b[] = { "to", spacing, "be" };
 Object n2b[] = { "not", spacing, _2b };
 Object phrase[] = { _2b, " or ", n2b };
 printRecursively(phrase);
 System.out.println(", that is the question.");
}
```

Implement the `printRecursively()` method so that it prints the following when the program is executed:

to be or not to be, that is the question.

7. Write a program that prints all combinations of words from given word groups. Use the following `main()` method in the program:

```
public static void main(String args[]) {
 String[][] wordGroups = {
 { "ugly", "funny", "eager" },
 { "dog", "man", "bear" },
 { "eats", "kicks", "hugs" },
 { "ball", "box", "cake" }
 };

 printCombinations(wordGroups);
}
```

Implement the `printCombinations()` method so that the program prints the following lines:

ugly dog eats ball  
ugly dog eats box  
ugly dog eats cake  
ugly dog kicks ball

... 73 other combinations ... (fix formatting: more space after this line)

eager bear kicks cake  
eager bear hugs ball



eager bear hugs box  
eager bear hugs cake

- 8.** Write a program that recursively computes the sum of all integers from 1 to  $n$ , by rewriting the formula:

$$\text{sum}(1, n) = 1 + 2 + 3 + \dots + n \text{ (fix formatting: more space after this line)}$$

to

$$\text{sum}(1, n) = 1 + (2 + 3 + \dots + n) = 1 + (\text{sum}(2, n))$$

Define the general and base cases in pseudocode before you implement the method.

Then, implement an iterative version of the algorithm and verify that the two programs produce the same results for  $n$  equal to 10, 100 and 1000. Compare the execution time of the two programs. Which would you say is the most efficient?

- 9.** Look up Exercise 5.5 on page 120, which addressed the problem of determining whether a string is a *palindrome* or not. As stated there, a palindrome is a string that has identical character sequence regardless of whether you read it backwards or forwards. The strings "aha", "abba" and "00700" are all examples of palindromes by this definition. A string containing just one character is also a palindrome.

Now, write a recursive algorithm for determining if a given string is a palindrome.

Define the general and base cases in pseudocode before you start writing code. It may be easier to start with the base cases, dealing with strings of length 1 and 2 characters. Then, try to develop the general case, which must handle strings of any length.

Compare your implementation of the recursive algorithm with the iterative program developed in Exercise 5.5 on page 120. Do you experience significant difference in performance? Which of the two programs do you find easier to understand?



## More on exception handling

### LEARNING OBJECTIVES

By the end of this chapter, you will understand the following:

- The inheritance hierarchy for exceptions provided by the Java standard library.
- Defining new exceptions.
- Throwing exceptions programmatically.
- Using the `finally` block with the `try-catch` statement.

### INTRODUCTION

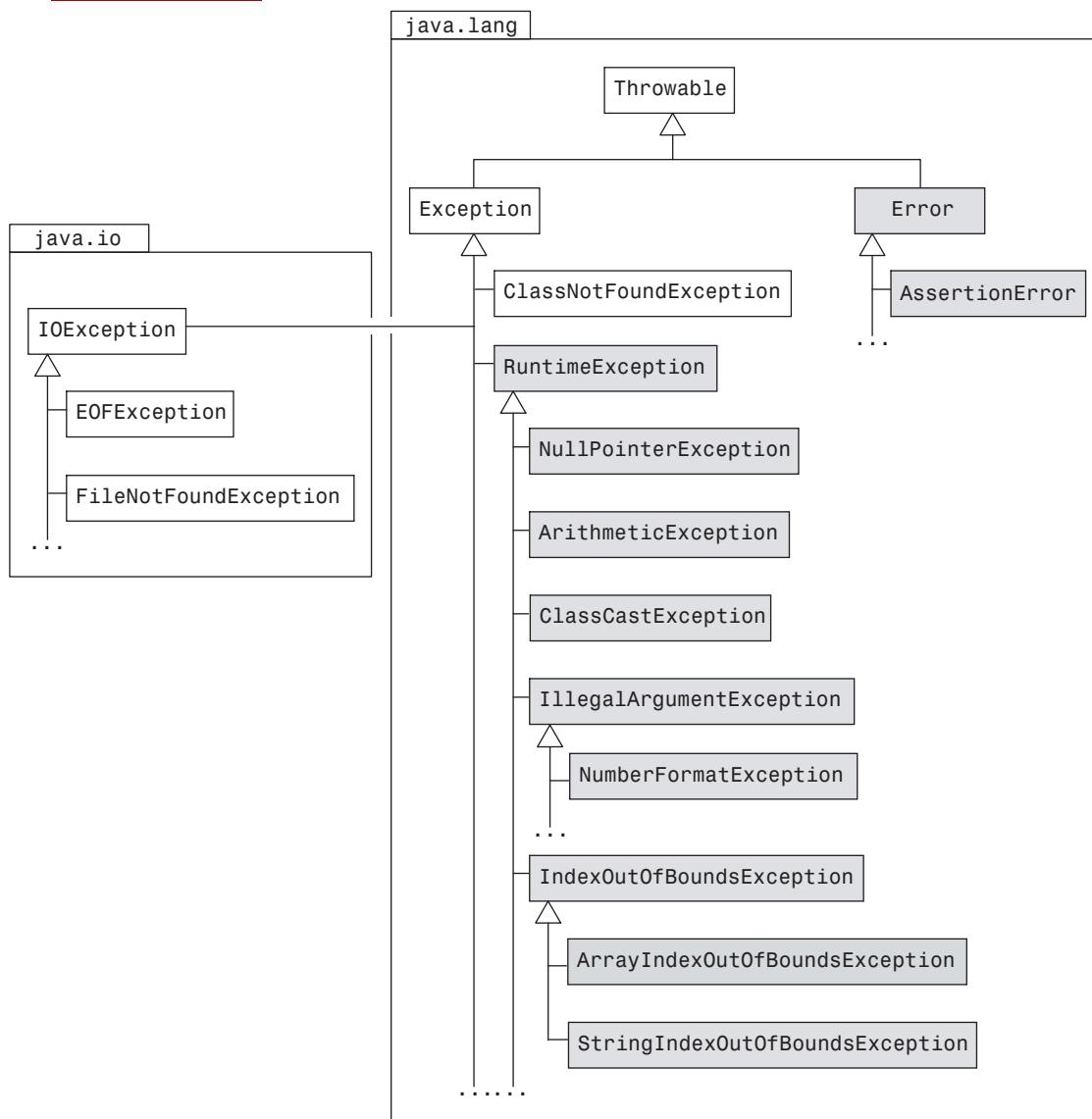
Chapter 10 provided an introduction to this topic, and should be read before proceeding with this chapter. This chapter covers additional aspects relating to exception handling: the inheritance hierarchy for exceptions provided by the Java standard library, defining new checked exceptions and using the `finally` block with the `try-catch` statement.

### 18.1 Exception classes

First we look at the inheritance hierarchy of exceptions we introduced in Chapter 10. Figure 18.1 shows a partial inheritance hierarchy for exception classes from the Java standard library. All exceptions are objects of the class `java.lang.Throwable` or its subclasses. Table 18.1 shows two useful methods that are inherited by all exceptions: the methods `getMessage()` and `printStackTrace()`. Here we will take a closer look at the three main categories of exceptions from the inheritance hierarchy, represented by the classes `Exception`, `RuntimeException` and `Error`.


**TABLE 18.1** Selected methods from the **Throwable** class

| Method                              | Description                                                                                    |
|-------------------------------------|------------------------------------------------------------------------------------------------|
| <code>String getMessage()</code>    | Returns the string in the exception. The string gives further explanation about the exception. |
| <code>void printStackTrace()</code> | Prints in the terminal window the stack trace at the time the exception was thrown.            |

**FIGURE 18.1** Partial hierarchy of exception classes




## The Exception class

Exceptions that are objects of the `Exception` class and its subclasses, *excluding those that are objects of the `RuntimeException` class or its subclasses*, are what we called *checked exceptions* in Section 10.4 on page 283. The compiler checks that if a method can throw a checked exception, the method must also explicitly deal with this type of exception. The method can choose to catch and handle the exception with the help of a try-catch statement, or explicitly propagate it with the help of the `throws` clause. See Section 10.4 on page 283 on how to handle checked exceptions and Table 10.1 on page 283 for some common checked exceptions.

## The `RuntimeException` class

Exceptions that are defined by the `RuntimeException` class and its subclasses, are what we called *unchecked exceptions* in Section 10.5 on page 286. See Table 10.2 on page 286 for some common unchecked exceptions.

Such exceptions are not checked by the compiler, and therefore the program is under no obligation to handle them. However, there are situations where it is appropriate to handle unchecked exceptions. For example, the method `parseInt()` in the class `Integer` converts a string to an `int` value, but what should the method return if the string cannot be converted to an `int` value? Returning any integer from this method will be interpreted as a legal result of a string conversion. Remember that a method in Java can return only one value in a return statement. The method `parseInt()` therefore throws an exception of type `NumberFormatException` to indicate problems with string conversions. A client can certainly catch such an exception to handle this error situation (Program 18.2).

## The Error class

The class `Error` and its subclasses define exceptions for serious problems that should never occur during execution, for example, internal errors in the JVM. Such situations are so serious that a program should not try to handle them, and are best left for the JVM to deal with them.

The `Error` class and its subclasses define unchecked exceptions. The subclass `AssertionError` defines an unchecked exception that is thrown when an assertion fails (see Table 10.2 on page 286). It is also thrown to indicate that a JUnit test failed, as we saw in Chapter 14.

## 18.2 Throwing an exception programmatically

A program can explicitly throw an exception with the `throw` statement, as shown in Program 18.1:

```
throw new ArithmeticException("Distance and time cannot be < 0"); // (4)
```

After the keyword `throw` we must specify an expression that evaluates to a reference value of an object of the class `Throwable` or one of its subclasses. At (4), the exception object is created in the usual way with the `new` operator, together with a constructor call. We choose a suitable exception class that describes the error situation, and pass additional



information about the exception as a string in constructor call. Execution of a throw statement terminates normal execution of the program, and the exception is propagated as described in Section 10.2 on page 272. Execution handling takes place in the usual way, regardless of whether the exception is thrown implicitly by the JVM or explicitly by a throw statement during program execution.

In Program 18.1 we choose to introduce an additional condition on the values of the parameters `distance` and `time` in the method `calculateSpeed()`, namely that their values cannot be less than 0. We can use an assertion to check this condition, but for illustrating exception handling, we use an if statement at (3). If we run the program with the following call in the `main()` method:

```
printSpeed(-100, 10); // (1) Value less than 0.
```

program execution will result in an exception of type `ArithmaticException` being thrown in the method `calculateSpeed()`. This exception is propagated and is caught by the catch block at (2) in the `main()` method. From this point onwards, normal execution of the program is resumed.

### PROGRAM 18.1 Throwing an exception programmatically

```
public class Speed4 {

 public static void main(String[] args) {
 System.out.println("Entering main().");
 try {
 printSpeed(-100, 10); // (1) Distance < 0.
 }
 catch (ArithmaticException exception) { // (2)
 System.out.println(exception + " (handled in main())");
 }
 System.out.println("Returning from main().");
 }

 private static void printSpeed(int kilometers, int hours) {
 System.out.println("Entering printSpeed().");
 int speed = calculateSpeed(kilometers, hours);
 System.out.println("Speed = " +
 kilometers + "/" + hours + " = " + speed);
 System.out.println("Returning from printSpeed().");
 }

 private static int calculateSpeed(int distance, int time) {
 System.out.println("Calculating speed.");
 if (distance < 0 || time < 0) // (3)
 throw new ArithmaticException("distance and time" +
 " cannot be < 0"); // (4)
 return distance/time;
 }
}
```



Program output:

```
Entering main().
Entering printSpeed().
Calculating speed.
java.lang.ArithmaticException: distance and time cannot be < 0 (handled in
main())
Returning from main().
```

---

### 18.3 Handling several types of exceptions

If the code in a `try` block can throw different types of exceptions, we can specify one `catch` block for each type of exception after the `try` block. Program 18.2 shows the use of several `catch` blocks, (5) and (6), that are associated with the same `try` block, (2).

The earlier example about calculating the speed is now augmented in Program 18.2. Before the `main()` method calls the `printSpeed()` method, (7), the program arguments are checked for legal integer values, (1). Program arguments from the command line are read and stored in the string array `args` in the `main()` method. The strings in the `args` array are converted to integers by calling the `parseInt()` method.

Accessing elements in the string array `args`, (3) and (4), with an illegal index, will throw an exception of the type `ArrayIndexOutOfBoundsException`, unless at least two strings are specified as program arguments. Converting to an integer with the `parseInt()` method, (3) and (4), will throw an exception of the type `NumberFormatException` if any of the strings contain characters that are not legal for an integer. The setup with only one `try` block and two associated `catch` blocks in Program 18.2 guarantees that the call to the `printSpeed()` method is only executed if two legal integers are supplied. Using the `try-catch` statement simplifies the checking of the two program arguments considerably, compared to using selection statements to check for every thing that can go wrong when reading program arguments. Program 18.2 shows examples of output, depending on what program arguments are supplied.

#### PROGRAM 18.2 One `try` block and several `catch` blocks

```
public class Speed5 {

 public static void main(String[] args) {
 System.out.println("Entering main()");
 int arg1, arg2;
 try {
 arg1 = Integer.parseInt(args[0]);
 arg2 = Integer.parseInt(args[1]);
 }
 catch (ArrayIndexOutOfBoundsException exception) {
 System.out.println("Specify both kilometers and hours.");
 }
 }
}
```



```

 System.out.println("Usage: java Speed5 <kilometers> <hours>");
 System.out.println(exception + " (handled in main())");
 return;
 }
 catch (NumberFormatException exception) { // (6)
 System.out.println("Kilometers and hours must be integers.");
 System.out.println("Usage: java Speed5 <kilometers> <hours>");
 System.out.println(exception + " (handled in main())");
 return;
 }
 printSpeed(arg1, arg2); // (7)
 System.out.println("Returning from main().");
}

private static void printSpeed(int kilometers, int hours) {
 System.out.println("Entering printSpeed().");
 try {
 int speed = calculateSpeed(kilometers, hours);
 System.out.println("Speed = " +
 kilometers + "/" + hours + " = " + speed);
 }
 catch (ArithmaticException exception) {
 System.out.println(exception + " (handled in printSpeed())");
 }
 System.out.println("Returning from printSpeed().");
}

private static int calculateSpeed(int distance, int time) {
 System.out.println("Calculating speed.");
 if (distance < 0 || time < 0)
 throw new ArithmaticException("distance and time cannot be < 0");
 return distance/time;
}
}

```

Examples of running the program:

```

> java Speed5 100
Entering main().
Specify both kilometers and hours.
Usage: java Speed5 <kilometers> <hours>
java.lang.ArrayIndexOutOfBoundsException: 1 (handled in main())
> java Speed5 200 4u
Entering main().
Kilometers and hours must be integers.
Usage: java Speed5 <kilometers> <hours>
java.lang.NumberFormatException: For input string: "4u" (handled in main())
> java Speed5 200 -10
Entering main().
Entering printSpeed().
Calculating speed.

```



```

java.lang.ArithmaticException: distance and time cannot be < 0 (handled in
printSpeed())
Returning from printSpeed().
Returning from main().
> java Speed5 200 0
Entering main().
Entering printSpeed().
Calculating speed.
java.lang.ArithmaticException: / by zero (handled in printSpeed())
Returning from printSpeed().
Returning from main().
> java Speed5 200 20
Entering main().
Entering printSpeed().
Calculating speed.
Speed = 200/20 = 10
Returning from printSpeed().
Returning from main().

```

## Typical programming errors in exception handling

A common programming error when using several catch blocks to catch different types of exceptions, is to specify a parameter type in a catch block that can *shadow* other exception types in subsequent catch blocks. For example, the superclass `RuntimeException` in the catch block (1) shadows the subclass `ArithmaticException` in the catch block (2):

```

try { ... }
catch (RuntimeException exception) { ... } // (1)
catch (ArithmaticException exception) { ... } // (2)

```

Exceptions of the type `ArithmaticException` are all caught by the catch block (1) and never by the catch block (2), since objects of subclasses can be assigned to a superclass reference. The compiler will report an error in such cases. The sequence of the catch blocks above must be reversed.

It can be tempting to catch all types of exceptions in one catch block by using a more general exception type, for example, either `Exception` or `RuntimeException`. This approach results in different types of exceptions being handled in the same way when they are caught, potentially making it difficult to understand the exact cause of the error. Using specific exception classes, usually with several catch blocks associated with a try block, can increase program understanding.

### BEST PRACTICES

Be as specific as possible when specifying the type of checked exceptions in a catch block or in a throws clause. Avoid exception superclasses that can shadow more precise exceptions that can occur in your program.



### BEST PRACTICES

Don't catch exceptions silently in a `catch` block just to restore normal execution. Provide feedback as to the course of action taken when an exception is caught. This makes the runtime behaviour of the program easier to understand.

### BEST PRACTICES

Don't go overboard in handling every error situation using exceptions. It is also not a good idea to use exception handling for situations that are not runtime errors, and certainly not for control flow in the program.

## 18.4 Defining new exceptions

It is possible to define new exceptions. The recommended practice is to define new checked exceptions by subclassing the `Exception` class. This makes it possible for the compiler to check any methods that use these exceptions. We can, for example, define our own exception to signal problems with calculating speed:

```
class SpeedCalculationException extends Exception {
 SpeedCalculationException(String str) {
 super(str);
 }
}
```

The definition of the class `SpeedCalculationException` shows that it is a subclass of the `Exception` class. All subclasses (except `RuntimeException`) of the `Exception` class define checked exceptions. It is usual to have a constructor that takes a string parameter. The string gives additional explanation about the exception. Most exception classes have such a constructor that allows the program to specify an explanation about the error.

Program 18.3 uses the class `SpeedCalculationException`. This exception is thrown by the `calculateSpeed()` method at (7). The exception is also declared in the `throws` clause of the methods `printSpeed()` and `calculateSpeed()` at (5) and (6), respectively. It is caught in the `catch` block in the `main()` method at (3).

### PROGRAM 18.3 Using user-defined exceptions

```
public class Speed7 {

 public static void main(String[] args) {
 System.out.println("Entering main().");
 try { // (1)
 calculateSpeed();
 } catch (SpeedCalculationException e) {
 System.out.println("Caught an exception: " + e.getMessage());
 }
 }

 void printSpeed() throws SpeedCalculationException { // (5)
 calculateSpeed();
 }

 void calculateSpeed() throws SpeedCalculationException { // (6)
 System.out.println("Calculating speed.");
 if (Math.random() > 0.5) {
 throw new SpeedCalculationException("Speed calculation failed!");
 }
 }
}
```



```

// printSpeed(100, 20); // (2a)
// printSpeed(-100,20); // (2b)
}
catch (SpeedCalculationException exception) { // (3)
 System.out.println(exception + " (handled in main())");
}
finally { // (4)
 System.out.println("Command to use: java Speed7");
}
System.out.println("Returning from main().");

private static void printSpeed(int kilometers, int hours)
 throws SpeedCalculationException { // (5)
 System.out.println("Entering printSpeed().");
 double speed = calculateSpeed(kilometers, hours);
 System.out.println("Speed = " +
 kilometers + "/" + hours + " = " + speed);
 System.out.println("Returning from printSpeed().");
}

private static int calculateSpeed(int distance, int time)
 throws SpeedCalculationException { // (6)
 System.out.println("Calculating speed.");
 if (distance < 0 || time <= 0)
 throw new SpeedCalculationException("distance and time " +
 "must be > 0"); // (7)
 return distance/time;
}
}

```

Output from Program 18.3 when (2a) is in the program, and (2b) is commented out:

```

Entering main().
Entering printSpeed().
Calculating speed.
Speed = 100/20 = 5.0
Returning from printSpeed().
Command to use: java Speed7
Returning from main().

```

Output from Program 18.3 when (2b) is in the program, and (2a) is commented out:

```

Entering main().
Entering printSpeed().
Calculating speed.
SpeedCalculationException: distance and time must be > 0 (handled in main())
Command to use: java Speed7
Returning from main().

```

**BEST PRACTICES**

Define a new checked exception if this will make the error situation easier to understand.

## 18.5 Using the `finally` block

Lastly, we will mention that a try-catch block sequence can be followed by an optional `finally` block. A catch block is not required to use a `finally` block together with a try block. The code in a `finally` block is *always* executed, regardless of how the try-catch statement was executed. Typical use of a `finally` block is for cleaning up, for example to close files or net connections that are still open.

An example using the `finally` block is shown in Program 18.3, where the try-catch block sequence at (1) and (3) has a matching `finally` block at (4). From the output of the program we see that the code in this `finally` block is always executed, regardless of whether an exception occurred in the try block or not.

**BEST PRACTICES**

Put the code to close resources (e.g. files, net connections, etc.) in a `finally` block to ensure that these resources are always closed, regardless of whether any exceptions are thrown. In general, put any clean-up code in a `finally` block.

**BEST PRACTICES**

Avoid returning from a method in the try block. Instead, use the `finally` block for such code, as this block is always executed. This approach makes the control flow cleaner and easier to understand.

## 18.6 Review questions

1. Which statement is true about exceptions in Java?
  - a Exceptions in Java are objects.
  - b All exceptions have the class `Throwable` as their superclass.
  - c It is not possible to define new exception classes.
  - d An exception is either thrown by the program or by the runtime environment.



(a), (b), (d)

It is possible to define new exception classes by subclassing the exception classes from the Java standard library.

**2.** Which classes define unchecked exceptions?

- a** NullPointerException
- b** ArithmeticException
- c** RuntimeException
- d** ArrayIndexOutOfBoundsException
- e** AssertionError

(a), (b), (c), (d)

The class `RuntimeException` and its subclasses define unchecked exceptions, as does the class `Error` and its subclasses.

**3.** Which classes define checked exceptions?

- a** All subclasses of the class `Exception`, except for the class `RuntimeException`
- b** Error
- c** ClassNotFoundException
- d** IOException

(a), (c), (d)

The class `Error` defines unchecked exceptions.

**4.** An exception can be thrown by using a \_\_\_\_\_ statement.

An exception can be thrown by using a `throw` statement.

**5.** Only statementB can cause an exception in the following code. The other statements do not change the control flow in any way in the program.

```
class TryCatchPE1 {
 public static void main(String[] args) {
 try {
 statementA;
 statementB; // can throw an exception
 statementC;
 }
 catch (ExceptionX x) {
 statementD;
 }
 catch (ExceptionY y) {
 statementE;
 }
 statementF;
 }
}
```



Answer the following questions:

- a** If statementB does not throw an exception, which statements will be executed?
- b** After statementB throws an exception of the type ExceptionY, which statements will be executed?
- c** After statementB throws an exception that is not caught, which statements will be executed?
- d** If statementB throws an exception, will statementC be executed?
  - (a) statementA, statementB, statementC and statementF.
  - (b) statementE and statementF. Normal execution resumed after statementE.
  - (c) None. The execution of the method main() is terminated.
  - (c) statementC is never executed if statementB throws an exception.

- 6.** What will the program print?

```
public class TryCatchPE2 {
 public static void main(String[] args) {
 test(0); // (1) Call to the method test().
 }

 private static void test(int choice) {
 System.out.println("test(): entry");
 try {
 System.out.println("try: start");
 if (choice < 0)
 throw new ArithmeticException();
 else if (choice > 0)
 throw new IndexOutOfBoundsException();
 System.out.println("try: done");
 }
 catch (ArithmeticException exception) {
 System.out.println("Exception handled: " + exception);
 }
 System.out.println("test(): exit");
 }
}
```

No exceptions are thrown.

```
test(): entry
try: start
try: done
test(): exit
```

- 7.** What will the program from Question 18.6 print if we replace the call to the method test() in (1) with the following call: test(-1)?

An exception of the type ArithmeticException is thrown, caught and handled.  
Normal execution continues afterwards.



```
test(): entry
try: start
Exception handled: java.lang.ArithmetricException
test(): exit
```

- 8.** What will the program from Question 18.6 print if we replace the call to the method `test()` in (1) with the following call: `test(1)`?

An exception of the type `IndexOutOfBoundsException` is thrown. It is not caught. The execution of the method `test()` is terminated, and the exception propagated. Since the method `main()` does not catch the exception either, its execution is terminated. Finally, the exception is handled by a default exception handler.

```
test(): entry
try: start
Exception in thread "main" java.lang.IndexOutOfBoundsException
 at TryCatchPE2.test(TryCatchPE2.java:14)
 at TryCatchPE2.main(TryCatchPE2.java:4)
```

- 9.** Which code will compile?

- a** finally { } try { } catch(Exception x) { }
- b** try { } finally { }
- c** try { } finally { } catch(Exception y) { }
- d** catch(Exception y) { } finally { }
- e** finally { }

(b)

## 18.7 Programming exercises

- 1.** Why will the program not compile? Insert throws clauses so that it compiles.

```
public class CheckedException {

 public static void main(String[] args) {
 test(-1);
 System.out.println("main(): done");
 }

 private static void test(int choice) {
 if (choice < 0)
 throw new ChoiceException("Choice < 0");
 else if (choice > 0)
 throw new ChoiceException("Choice > 0");
 else
 throw new ArithmetricException("Choice == 0");
 }
}

class ChoiceException extends Exception {
```



```
 public ChoiceException(String message) { super(message); }
}
```

The class `ChoiceException` defines checked exceptions. The method `test()` must either handle an exception of the type `ChoiceException` or propagate it explicitly with the help of a `throws` clause. We have in the solution chosen to let the method propagate it further. Since the method `main()` calls the method `test()`, the method `main()` must also take a stand regarding checked exceptions of the type `ChoiceException`. It also chooses to propagate the exceptions further. The program will now compile. The class `ArithException` defines unchecked exceptions. The method `test()` need not take any special action regarding these exceptions.

```
public class CheckedException {

 public static void main(String[] args) throws ChoiceException {
 test(-1);
 System.out.println("main(): done");
 }

 private static void test(int choice) throws ChoiceException {
 if (choice < 0)
 throw new ChoiceException("Choice < 0");
 else if (choice > 0)
 throw new ChoiceException("Choice > 0");
 else
 throw new ArithException("Choice == 0");
 }
}
```

```
class ChoiceException extends Exception {
 public ChoiceException(String message) { super(message); }
}
```

Write a new version of the program from Exercise 18.1 so that the `test()` method catches any checked exceptions its throws.

The method `test()` uses try-catch blocks to catch all checked exceptions it throws.

```
public class CheckedException2 {

 public static void main(String[] args) {
 test(-1);
 System.out.println("main(): done");
 }

 private static void test(int choice) {
 try {
 if (choice < 0)
 throw new ChoiceException("Choice < 0");
 else if (choice > 0)
 throw new ChoiceException("Choice > 0");
 else
 throw new ArithException("Choice == 0");
 }
 }
}
```



```
 }
 catch (ChoiceException exception) {
 exception.printStackTrace();
 }
}
```

2. Rewrite the program from Exercise 18.1 so that the `main()` method catches all checked exceptions that the `test()` method throws.

The method `main()` uses a `try-catch` statement to catch all checked exceptions the method `test()` throws. The method `test()` must declare these in a `throws` clause.

```
public class CheckedException3 {

 public static void main(String[] args){
 try {
 test(-1);
 }
 catch (ChoiceException exception) {
 exception.printStackTrace();
 }
 System.out.println("main(): done");
 }

 private static void test(int choice) throws ChoiceException {
 if (choice < 0)
 throw new ChoiceException("Choice < 0");
 else if (choice > 0)
 throw new ChoiceException("Choice > 0");
 else
 throw new ArithmeticException("Choice == 0");
 }
}
```



## Files and Streams

### LEARNING OBJECTIVES

By the end of this chapter, you will understand the following:

- Which byte and character streams Java provides for reading values from sources and writing values to destinations.
- How to read and write values to the terminal window.
- How to read and write values in binary form on files.
- How to store and retrieve objects from files.
- How to use random access to read and write records on files.

### INTRODUCTION

This chapter first presents an overview of *streams* that Java provides for communication of data between the program and its environment (Section 19.1)

The terminal window is used for command line-based dialogue between the user and the program. Section E.1 provides a customised class for reading input from the terminal window. This chapter provides an example of reading values from and writing value to the terminal window using classes from the Java standard library (Section 19.2).

In Section 11.1 and Section 11.2 we saw how we could read from and write *records* on *text files*. In this chapter we look at how to use *binary files* for the same purpose (Section 19.3). Java also allows objects to be stored on and retrieved from files. Section 19.4 discusses the *object serialisation* mechanism that can be used for this purpose. Text files and object serialisation use *sequential access*. In contrast, Section 19.5 provides an example of *random access* for reading and writing records on files.

The examples in this chapter make extensive use of the Employee class and the PersonnelRegister class from Program 11.1 on page 295. It is therefore a good idea to review Section 11.1 and Section 11.2 before proceeding with this chapter.



## 19.1 Streams

Handling of sources and destinations for data is based on *data streams* in Java. A *stream* is an object that a program can use to write data to a destination or read data from a source. A stream that is used to read a sequence of data from a source is called an *input stream*, and a stream that is used to write a sequence of data to a destination is called an *output stream*. Since data can only be read or written as a sequence of values with one value at a time, these streams are called *sequential streams*.

A stream can, for example, be connected to a data file, a net connection, the keyboard or the terminal window. The program uses streams to read and write data. The package `java.io` provides a functionally rich assortment of classes that represent different types of streams. These classes can be used to mediate data between different kinds of sources and destinations.

### Overview of byte streams

Byte streams handle *sequences of bytes* (i.e. data consisting of 8 bits, where a bit is the smallest unit of information that has either the value 0 or 1). These streams can be categorised as *byte input streams* with the `InputStream` class as the abstract superclass, and as *byte output streams* with the `OutputStream` class as the abstract superclass (Figure 19.1).

The classes `FileInputStream` and `FileOutputStream` read data from and writes data to files. We used these streams in Section 11.2, and we will use them extensively in this chapter as well.

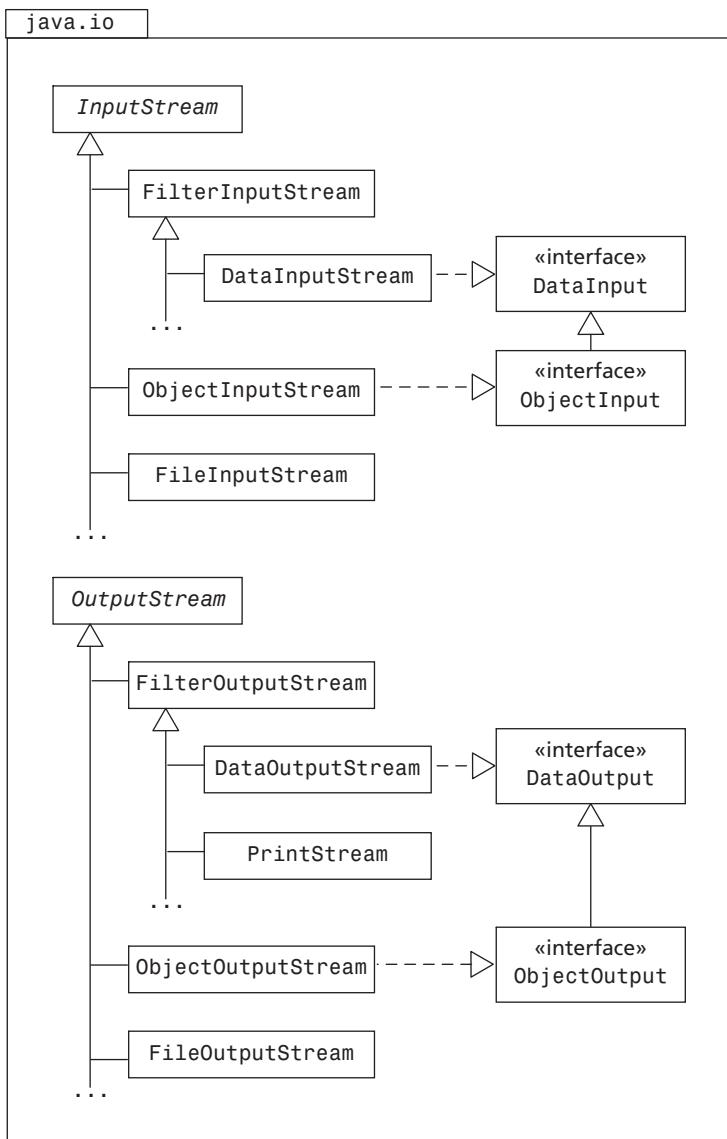
The subclasses of the classes `FilterInputStream` and `FilterOutputStream` are specialised streams, called *filters*, for further processing of data to or from other streams. The subclasses `DataInputStream` and `DataOutputStream` read and write *binary representation* of primitive values to other streams (see Section 19.3). By connecting a filter and a file stream we can write primitive values to or read them from a binary file.

The classes `ObjectOutputStream` and `ObjectInputStream` allow us to store objects on files and to retrieve them later (see Section 19.4).

The class `PrintStream` translates data values to characters, where each character is sent as a fixed number of bytes. We take a closer look at conversion of characters to and from bytes in the subsection *Overview of character streams* on page 587. The reference `System.out` refers to an object of the class `PrintStream`, that we can use to print characters to the terminal window by calling the different `print()` methods in this class (Section 19.2).



**FIGURE 19.1** Byte streams



## Overview of character streams

Character streams, like the byte streams, are also divided into two categories: *readers* and *writers*, that are implemented by the abstract class `Reader` and the abstract class `Writer`, respectively (Figure 19.2). The classes `FileWriter` and `FileReader` are concrete subclasses that write and read character from files, respectively (Section 11.2).

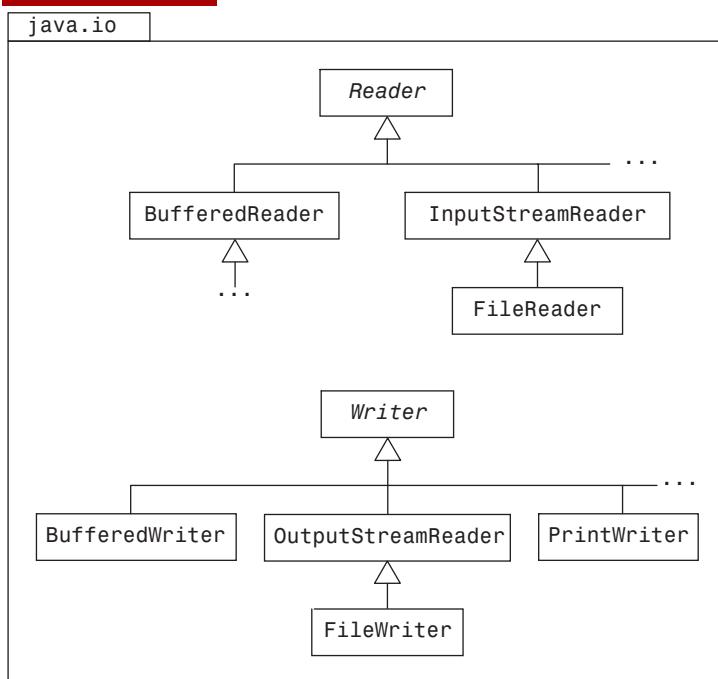
The class `PrintWriter` defines streams that write a *text representation* of primitive values to other streams. For example, the integer 84152 (`int`) will be converted to a sequence of characters that correspond to the digits in the number, "84152". If we wish to write text representation of this integer (or other primitive values) to a file, we can first open the file with a `FileWriter`, that is in turn connected to a `PrintWriter`. Now the `PrintWriter`



object will convert the integer to the correct number of characters and deliver them to the `FileWriter` object. The `FileWriter` object will write the bytes representing these characters in the file (see subsection *Writing to text files* on page 298).

Writing or reading *one* character at a time directly from a file is not particularly effective. Moving data between the internal memory and the external medium is a costly operation, and can reduce the speed of a program considerably. We obtain a better performance by storing characters temporarily in the internal memory and moving several characters at a time between the internal memory and the external medium. Such an area in memory for temporary storing of data is called a *buffer*. The classes `BufferedReader` and `BufferedWriter` provide a solution for such temporary storing of data. These classes handle filling and emptying of such a buffer when necessary (see subsection *Reading from text files* on page 304).

**FIGURE 19.2** Character streams



## 19.2 Terminal window I/O

Three byte streams are automatically opened and associated with a Java program when it starts executing:

- 4 `System.in`, which is an `InputStream` object, is used to read bytes from the keyboard (called *standard in* or *stdin*).
- 5 `System.out`, which is a `PrintStream` object, is used for writing to the terminal window (called *standard out* or *stdout*).



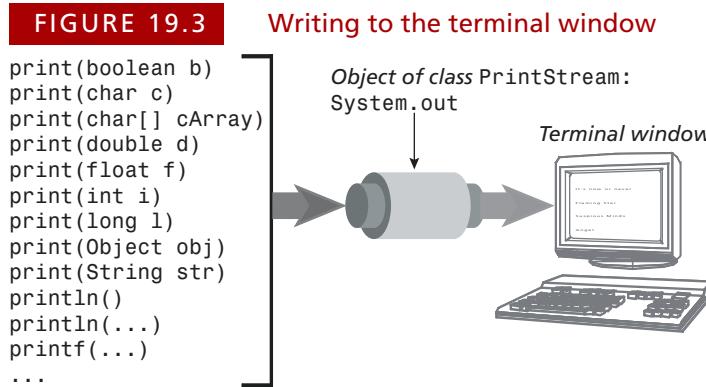
**6** `System.err`, which is also a `PrintStream` object, is used for reporting errors (called *standard error* or *stderr*).

We use the term *terminal window I/O* for reading values from the keyboard and writing values to the terminal window.

## Writing to the terminal window

Section 2.1 showed how the `System.out` object can be used to write to the terminal window. The class `PrintStream` provides a number of `print()` methods for writing text representation of different types of values (see Figure 19.3). We have seen many examples of using the `System.out` object for this purpose, such as:

```
System.out.println("Life is too short. Eat your dessert first!");
```

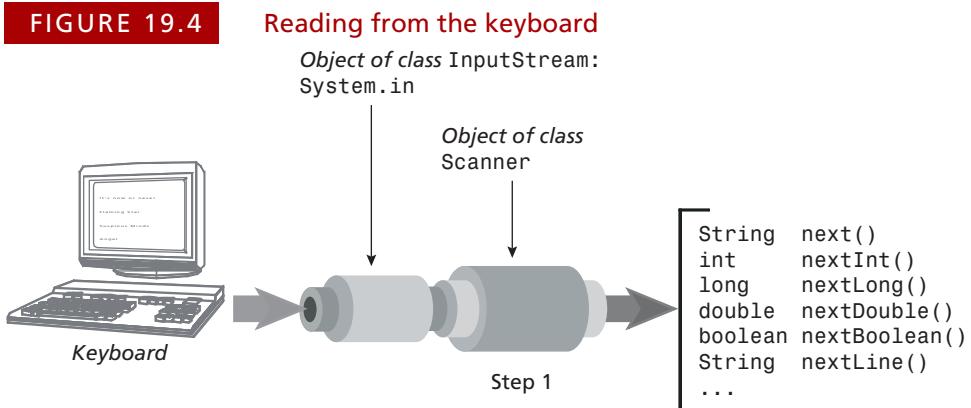


## Reading from the keyboard

As mentioned earlier, a Java program uses the `System.in` object to read what is typed on the keyboard. The problem is that the `System.in` object is a *byte* input stream, and it does not automatically convert bytes to 16-bit Unicode characters or to other primitive values. We connected a `Scanner` object to the `System.in` object in order to convert the bytes to strings and other values (see Figure 19.4).

Program 19.1 uses the `Stream.in` object and the `Scanner` class to read information about employees from the keyboard. The class `CompanyUsingTerminalWindow` defines the method `readAllEmployeesFromTerminalWindow()` at (1) that uses a loop to read employee information. Running of the program shows an example of the dialogue between the program and the user. Note that the program is not very robust, as it does not recover gracefully from errors in the user input.

Appendix E provides a customised class, called `Console`, for reading values from the keyboard. The `Console` class is preferable for this purpose, as it handles exceptions that occur due to errors in the user input. This class also hides the use of the `Stream.in` object and the `Scanner` class, making it easy to read values from the keyboard.

**FIGURE 19.4****PROGRAM 19.1****Reading from the keyboard**

```

class Employee {
 // See the class Employee in Program 11.1 on page 295.
}

class PersonnelRegister {
 // See the class PersonnelRegister in Program 11.1 on page 295.
}

import java.util.Scanner;

/**
 * Class that reads employee data from terminal window,
 * and prints some statistics.
 */
class CompanyUsingTerminalWindow {

 // Field
 private PersonnelRegister register;

 // Constructors
 public CompanyUsingTerminalWindow() {
 register = new PersonnelRegister();
 }
 public CompanyUsingTerminalWindow(PersonnelRegister register) {
 this.register = register;
 }

 // Write statistics.
 void printReport() {
 System.out.println(register);
 }
}

```



```

void writeAllEmployeesToTerminalWindow() {
 Employee[] employees = register.getEmployeeArray();
 int numEmployees = register.getNumEmployees();
 System.out.println("No. of employees: " + numEmployees);
 for (int i = 0; i < numEmployees; i++)
 System.out.println(employees[i]);
}

void readAllEmployeesFromTerminalWindow() { // (1)
 Scanner keyboard = new Scanner(System.in);
 String reply = null;
 do {
 System.out.println("Specify data for an employee.");
 System.out.print("First name: ");
 String firstName = keyboard.nextLine();
 System.out.print("Last name: ");
 String lastName = keyboard.nextLine();
 System.out.print("Hourly rate: ");
 double hourlyRate = keyboard.nextDouble();
 keyboard.nextLine();
 System.out.print("Is it a woman? (y/n): ");
 reply = keyboard.nextLine();
 Gender gender = Gender.MALE;
 if (reply.equals("y"))
 gender = Gender.FEMALE;
 Employee employee = new Employee(firstName, lastName,
 hourlyRate, gender);
 register.registerEmployee(employee);
 System.out.print("Register more employees? (y/n): ");
 reply = keyboard.nextLine();
 } while (reply.equals("y"));
}

public static void main(String[] args) {
 // Create a personnel register.
 PersonnelRegister register = new PersonnelRegister();
 // Create a company.
 CompanyUsingTerminalWindow company
 = new CompanyUsingTerminalWindow(register);

 // Read employee data from terminal window.
 System.out.println("Reading from terminal window.");
 company.readAllEmployeesFromTerminalWindow();

 // Print statistics to terminal window.
 company.writeAllEmployeesToTerminalWindow();
 company.printReport();
}
}

```



Running the program:

```

Reading from terminal window.
Specify data for an employee.
First name: Bill
Last name: Bailey
Hourly rate: 612.0
Is it a woman? (y/n): n
Register more employees? (y/n): y
Specify data for an employee.
First name: Ole
Last name: Olsen
Hourly rate: 515.0
Is it a woman? (y/n): n
Register more employees? (y/n): n
No. of employees: 2
First name: Bill Last name: Bailey Hourly rate: 612.00 Gender: MALE
First name: Ole Last name: Olsen Hourly rate: 515.00 Gender: MALE
The company has 2 employees, where 0.00% are women.

```

---

## 19.3 Binary files

A binary file stores binary representation of primitive values defined in the Java programming language. Numeric values in a binary file occupy storage according to their length defined in the Java programming language (Table 11.1, page 294). Characters are represented in the file in the UTF-16 character encoding scheme. The boolean values `true` and `false` are stored in the file in binary representation given by `(byte)1` and `(byte)0`, respectively. Actually it is not important for the program to know how values are stored in binary form. What is important is that values that are stored in binary form are interpreted correctly when they are read.

### File path

In Section 11.2, we learnt that in order to read from or write to a file, we must first *open* the file. This means that we must create an appropriate stream that is connected to the file. A file is identified by a *file path* in the file system. The file path is specified as a string in the program:

```
String dataFileName = "employeeFile.dat"; // (1)
```

The way we have specified the file name as a string in (1) above requires that the file is in the current directory, i.e. in same directory as the program. If the file is not in the current directory, we have to specify a more precise file path for the file. For example, the file paths in the following lines of code:

```
String filePath1 = "company\\employeeFile.dat"; // Windows
String filePath2 = "company/employeeFile.dat"; // Unix
```



indicate that the file with the name `employeeFile.dat` is in the directory `company`, which in turn is in the current directory. The character used as delimiter between terms in the file path is platform dependent. For Windows, we must use *two* backslash characters (`\`) in the string in order to indicate one backslash character (`\`), since the backslash character has special meaning in strings. Java defines the constant `File.separator` whose value is the platform-dependent file path delimiter. It should be used to construct file paths in order to make the program portable:

```
String fileName = "company" + File.separator + "employeeFile.dat";
```

A single dot (.) and double dots (..) designate special terms in a file path, and are interpreted as the current directory and the parent directory, respectively.

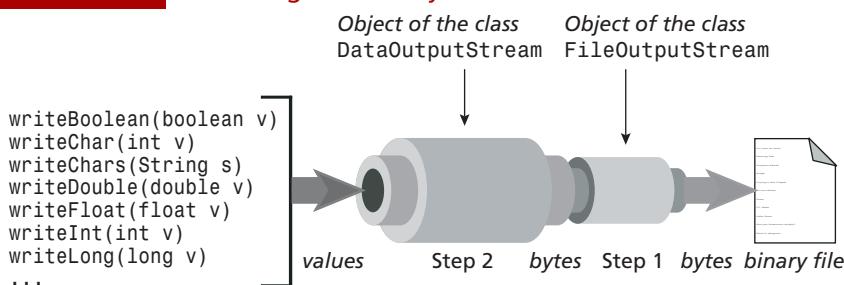
## Writing to binary files

The class `DataOutputStream` writes binary representation of different types of primitive values to an output stream (Figure 19.5). For each primitive data type X, there is a method `writeX()` that can be used to write values of this type in binary form.

If a field in a record has a primitive value, it will always contain a fixed number of bytes that can be interpreted as a single value when read. We need not mark the end of such fields when writing. On the other hand, for a field with a string, we need to keep track of the number of characters in the string, so that the field value is interpreted correctly when read. One solution is to store the string length in the field. Another solution is to write a fixed number of characters for each field that has a string value. If the string does not occupy the whole field, we can fill it with a padding character (Section 19.5). Here we will use a third solution: we will write the whole string, followed by a particular character that marks the end of the string and thereby the end of the field. When reading the value of such a field, we read its characters, including the field terminator character which we can then discard. With this solution, the field terminator character cannot occur as an ordinary character in such a field.

**FIGURE 19.5**

Writing to a binary file



```
FileOutputStream outFileStream = new FileOutputStream(binaryFileName, append); // Step 1
DataOutputStream outBinStream = new DataOutputStream(outFileStream); // Step 2
```

Since we will store the values on a file, we must create a file stream so that the data is stored on a file. This file stream must be connected to a `DataOutputStream` object in order for the binary values to be stored in the file. The procedure for writing binary values on a file is then as follows (see Figure 19.5 and Program 19.2):



- 1** Create a `FileOutputStream` object that opens the file for writing:

```
FileOutputStream outFileStream
 = new FileOutputStream(binaryFileName, append); // (4)
```

The boolean value in the argument `append` specifies whether writing should begin at the start or at the end of any existing content in the file. If the file cannot be opened, the constructor will throw a checked exception of the type `FileNotFoundException`.

- 2** Create a `DataOutputStream` object that is connected to the `FileOutputStream` object from step 1:

```
DataOutputStream outBinStream
 = new DataOutputStream(outFileStream); // (5)
```

- 3** Write values in binary form using the relevant `writeX()` method defined in the `DataOutputStream` class. For example, the following code lines will write the information about an employee:

```
outBinStream.writeChars(employee.firstName + FIELD_TERMINATOR);
outBinStream.writeChars(employee.lastName + FIELD_TERMINATOR);
outBinStream.writeDouble(employee.hourlyRate);
outBinStream.writeInt(employee.gender.ordinal());
```

Note that strings are terminated with the field terminator character. For the gender of an employee, we write the *ordinal value* of the `Gender` enum constant. The ordinal value of an enum constant is given by its position in the list of enum constant.

The first enum constant has the ordinal value 0 and so on.

- 4** Finish by closing the output stream, which also closes the underlying output file:

```
outBinStream.close();
```

Program 19.2 shows the two classes `CompanyAdminBin` and `CompanyUsingBinFiles` that illustrate how to use binary files. These classes maintain the employees of the DOT-COM company using the `Employee` and `PersonnelRegister` classes from Program 11.2 on page 300. The class `CompanyAdminBin` creates a `CompanyUsingBinFiles` object that uses a `PersonnelRegister`. The following method call at (1) ensures that information about all employees in the register is written in binary form on the specified file:

```
company.writeAllEmployeesToBinFile("employeeFile.bin", false); // (2)
```

The second parameter specifies that writing should begin at the start of the file. The method `writeAllEmployeesToBinFile()` creates the necessary streams for writing binary values at (4) and (5). It writes the number of employees to be stored in the file at (6a), followed by information about each employee at (6b). The data output stream is closed at (7), which also closes the underlying file output stream.

The method `writeEmployeeData()` at (10) writes information about each employee as outlined in step 3 above.

Note that the method `writeAllEmployeesToBinFile()` explicitly deals with checked exceptions of the type `FileNotFoundException` in case it is not possible to open the file, and of the type `IOException` in case something goes wrong during writing (see catch blocks at (8) and (9)).



## PROGRAM 19.2 Writing and reading binary values

```

class Employee {
 // See the class Employee in Program 11.1 on page 295.
}

class PersonnelRegister {
 // See the class PersonnelRegister in Program 11.1 on page 295.
}

import java.io.IOException;
/*
 * Company administration using binary files.
 */
public class CompanyAdminBin {

 public static void main(String[] args) throws IOException {
 // Create an array of employees.
 Employee[] employeeInfo = {
 new Employee("Ole", "Olsen", 30.00, Gender.MALE),
 new Employee("Bill", "Bailey", 40.00, Gender.MALE),
 new Employee("Liv", "Larsen", 50.00, Gender.FEMALE)
 };

 // Create a personnel register.
 PersonnelRegister register = new PersonnelRegister(employeeInfo);

 // Create a company that uses binary files. (1)
 CompanyUsingBinFiles company = new CompanyUsingBinFiles(register);

 // Write employee data in the personnel register to a binary file.
 company.writeAllEmployeesToBinFile("employeeFile.bin", false); // (2)

 // Read employee data from a binary file.
 System.out.println("Reading from binary file.");
 company.readAllEmployeesFromBinFile("employeeFile.bin"); // (3)

 // Print employee data to the terminal window.
 company.printAllEmployeesToTerminalWindow();
 company.printReport();
 }
}

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

```



```
class CompanyUsingBinFiles {

 final static char FIELD_TERMINATOR = ',';

 // Field
 private PersonnelRegister register;

 // Constructor
 CompanyUsingBinFiles() {
 register = new PersonnelRegister();
 }
 CompanyUsingBinFiles(PersonnelRegister register) {
 this.register = register;
 }

 // Print statistics
 void printReport() {
 System.out.println(register);
 }
 void printAllEmployeesToTerminalWindow() {
 Employee[] employees = register.getEmployeeArray();
 int numOfEmployees = register.getNumOfEmployees();
 System.out.println("Number of employees: " + numOfEmployees);
 for (int i = 0; i < numOfEmployees; i++) {
 System.out.println(employees[i]);
 }
 }

 void writeAllEmployeesToBinFile(String binaryFileName, boolean append) {
 try {
 FileOutputStream outFileStream
 = new FileOutputStream(binaryFileName, append); // (4)
 DataOutputStream outBinStream
 = new DataOutputStream(outFileStream); // (5)

 // Write the number of employees in the register.
 int numOfEmployees = register.getNumOfEmployees();
 outBinStream.writeInt(numOfEmployees); // (6a)

 // Write data about each employee.
 Employee[] employees = register.getEmployeeArray();
 for (int i = 0; i < numOfEmployees; i++)
 writeEmployeeData(outBinStream, employees[i]); // (6b)

 outBinStream.close(); // (7)
 }
 catch (FileNotFoundException exception) { // (8)
 System.out.print("Error opening file: " + binaryFileName);
 System.exit(1); // Terminate if an exception occurs.
 }
 }
}
```



```

}

catch (IOException exception) { // (9)
 System.out.print("Error writing to file: " + exception);
 System.exit(1); // Terminate if an exception occurs.
}
}

void writeEmployeeData(DataOutputStream outBinStream,
 Employee employee) throws IOException { // (10)
 outBinStream.writeChars(employee.firstName + FIELD_TERMINATOR);
 outBinStream.writeChars(employee.lastName + FIELD_TERMINATOR);
 outBinStream.writeDouble(employee.hourlyRate);
 outBinStream.writeInt(employee.gender.ordinal());
}

void readAllEmployeesFromBinFile(String binaryFileName) {
 try {
 FileInputStream inFileStream
 = new FileInputStream(binaryFileName); // (11)
 DataInputStream inBinStream
 = new DataInputStream(inFileStream); // (12)

 // Read how many employees are in the binary file.
 int numOfEmployees = inBinStream.readInt(); // (13)

 // Create a personnel register.
 register = new PersonnelRegister();

 // Read data about all employees in the file.
 for (int i = 0; i < numOfEmployees; i++) {
 // Read an employee.
 Employee employee = readEmployeeData(inBinStream); // (14)
 // Register the employee.
 register.registerEmployee(employee);
 }

 inBinStream.close(); // (15)
 }
 catch (IOException exception) {
 System.out.print("Error writing to file: " + exception);
 System.exit(1); // Terminate if exception occurs.
 }
}

Employee readEmployeeData(DataInputStream inBinStream)
 throws IOException { // (16)
 String firstName = readBinStringField(inBinStream);
 String lastName = readBinStringField(inBinStream);
 double hourlyRate = inBinStream.readDouble();
 Gender gender = null;
}

```



```

switch (inBinStream.readInt()) { // Read ordinal value for gender.
 case 0: default:
 gender = Gender.FEMALE;
 break;
 case 1:
 gender = Gender.MALE;
 break;
}
return new Employee(firstName, lastName, hourlyRate, gender);
}

private String readBinStringField(DataInputStream inBinStream) // (17)
throws IOException {
// Assumes that the field is terminated by FIELD_TERMINATOR.
StringBuilder charBuffer = new StringBuilder();
while(true) {
 char character = inBinStream.readChar();
 if (character == FIELD_TERMINATOR)
 break;
 else
 charBuffer.append(character);
}
return charBuffer.toString();
}
}

```

Program output:

```

Reading from binary file.
Number of employees: 3
First name: Ole Last name: Olsen Hourly rate: 30.00 Gender: MALE
First name: Bill Last name: Bailey Hourly rate: 40.00 Gender: MALE
First name: Liv Last name: Larsen Hourly rate: 50.00 Gender: FEMALE
The company has 3 employees, where 33.33% are women.

```

## Reading binary values

Correct reading of binary values stored on a file, requires that we know their data types. The values must also be read in the order they were written, i.e. sequentially. The procedure for reading binary values from a file is as follows (see Figure 19.6 and Program 19.2):

- 1 Create a `FileInputStream` object that opens the file to read the binary values from:

```

FileInputStream inFileStream
= new FileInputStream(binaryFileName); // (11)

```

If the file does not exist, an exception of the type `FileNotFoundException` is thrown.

- 2 Create a `DataInputStream` object that is connected to the `FileInputStream` object from step 1 above:

```

DataInputStream inBinStream

```



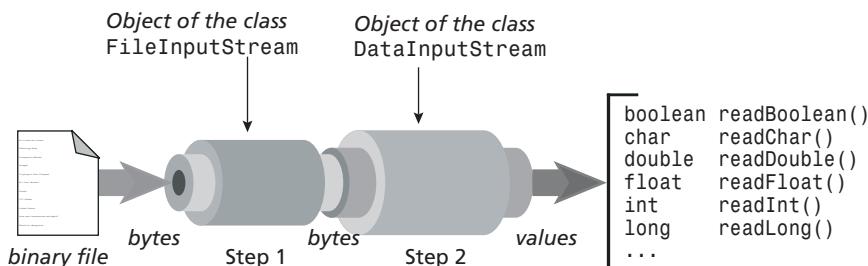
```
= new DataInputStream(inFileStream); // (12)
```

- 3** Read the binary values using the relevant `readX()` method defined in the `DataInputStream` class. For example, the following statement will read the hourly rate of an employee:

```
double hourlyRate = inBinStream.readDouble();
```

- 4** Finish by closing the input stream, which also closes the underlying input file:
- ```
inBinStream.close();
```

FIGURE 19.6 Reading binary values



```
FileInputStream inFileStream = new FileInputStream(binaryFileName); // Step 1
DataInputStream inBinStream = new DataInputStream(inFileStream); // Step 2
```

The class `CompanyAdminBin` in Program 19.2 calls the method `readAllEmployeesFromBinFile()` for reading information about all the employees stored on the binary file:

```
company.readAllEmployeesFromBinFile("employeeFile.bin"); // (3)
```

The method `readAllEmployeesFromBinFile()` creates the necessary streams for reading binary values at (11) and (12). First, it reads how many records are stored in the file at (13). It also creates a new `PersonnelRegister` to store employee information. Secondly, information about each employee is read one at a time from the file at (14), and the employee inserted into the personnel register. Finally, the input stream is closed at (15), thereby also closing the underlying file.

Information about each employee is read by the method `readEmployeeData()` at (16). The first and last names are read by the method `readBinStringField()` at (17). It reads a string that is terminated by the field terminator character. The characters are read one at a time until the field terminator character is read. Note the uses of the `throws` clause to handle exceptions of the type `IOException`, that can be thrown by the `readX()` methods. Program 19.2 uses the character ',' as the field terminator character. Other characters commonly used for this purpose are '\u0000', '|', '\t' and '\n'.

Regarding the gender of an employee, we stored the ordinal value of the `Gender` enum constant `MALE` and `FEMALE` in the file. In the method `readEmployeeData()` at (17), the ordinal value is read and converted to the corresponding `Gender` enum constant using a `switch` statement.



Handling end of file

The number of binary values on the files is not always known beforehand. The end of file (i.e. no more data in the file) during reading is marked by throwing a checked exception of the type `EOFException`. All `readX()` methods throw this type of exception if they are called and there is no more data to read.

Program 19.3 shows reading of integers from a binary file when the number of integers in the file is not known. The method `main()` at (1) catches an exception of the type `FileNotFoundException` at (4), while other exceptions of the type `IOException` are caught at (5). These exceptions can occur because of the call to the method `readFromBinFile()` at (3).

The method `readFromBinFile()` is designed such that it only catches exceptions of the type `EOFException` at (11), and propagates other exceptions of the type `IOException` further, using its `throws` clause at (6). The design of the method `readFromBinFile()` illustrates how a method can be selective in what exceptions it wants to catch or propagate.

Reading of values is done in the `try` block at (7). Opening of the file at (8) can throw an exception of the type `FileNotFoundException`, which is propagated further by the `throws` clause at (6). The loop at (9) terminates when an exception of the type `EOFException` is thrown at (10) during the execution of the call to the method `readInt()`. This exception is caught and handled by the associated `catch` block at (11).

The associated `finally` block at (12) ensures that the streams are closed. This block is executed regardless of any exception thrown in the `try` block at (7). The `if` statement at (13) guarantees that the streams are only closed if they have been created. Note that the closing of streams at (14) also requires handling of exceptions of the type `IOException`, which are propagated further by the `throws` clause at (6).

The solution in Program 19.3 is actually using two nested `try-catch` statements, (2) and (7), that are in two different methods. This is a consequence of the strategy chosen for exception handling.

PROGRAM 19.3 Handling end of file when reading binary values

```

import java.io.DataInputStream;
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class EndOfFileBin {
    public static void main(String[] args) { // (1)
        try { // (2)
            readFromBinFile(args[0]); // (3)
        }
        catch (ArrayIndexOutOfBoundsException exception) {
            System.out.println("File name not specified on command line.");
            exception.printStackTrace();
        }
    }
}

```



```

        catch (FileNotFoundException exception) { // (4)
            System.out.println("File not found: " + exception);
            exception.printStackTrace();
        }
        catch (IOException exception) { // (5)
            System.out.println("Error reading from file: " + exception);
            exception.printStackTrace();
        }
    }

public static void readFromBinFile(String dataFileName)
    throws IOException { // (6)

    FileInputStream inFileStream = null;
    DataInputStream inBinStream = null;

    try {
        inFileStream = new FileInputStream(dataFileName); // (7)
        inBinStream = new DataInputStream(inFileStream); // (8)
        while (true) { // (9)
            int i = inBinStream.readInt(); // (10)
            System.out.println(i);
        }
    }
    catch (EOFException exception) { // (11)
        System.out.println("End of file.");
    }
    finally {
        if (inBinStream != null) // (12)
            inBinStream.close(); // (13)
        // (14)
    }
}
}

```

Program output:

```

> java EndOfFile integers.bin
2007
2008
2009
2010
2011
End of file.

```



BEST PRACTICES

If data is to be used by different Java programs, specially on different platforms, consider using binary files. These files are compatible across platforms, and will be interpreted correctly by a Java program.

BEST PRACTICES

When reading an unknown number of binary values from a `DataInputStream`, catch the `EOFException` to determine that all data has been read.

19.4 Object serialisation

Java provides a mechanism that allows us to store objects. Objects can be *serialised*. This means that all relevant information about an object is written on, for example, a file, as a sequence of bytes, and that the object can be recreated by reading and interpreting these bytes. The new object that is created will have the same state as the old object before it was serialised. The process of writing an object is called *serialisation*. The process of reading the information to recreate an object is called *deserialisation*.

The classes `ObjectOutputStream` and `ObjectInputStream` define *object output streams* and *object input streams*, respectively, that provide methods for serialising and deserialising objects. We can serialise strings and arrays, since these are objects. In addition, these classes provide methods for reading and writing binary representation of primitive data values, identical to the classes `DataInputStream` and `DataOutputStream`. This means that we can read and write binary values and objects with these object streams.

Objects of a class can only be serialised if the class implements the `Serializable` interface. This interface does not specify any methods, but functions as a *marker* that objects of the class can be serialised. This means that a class must explicitly allow its objects to be serialised:

```
class Employee implements Serializable {...}
```

The classes `ObjectOutputStream` and `ObjectInputStream` are only responsible for object serialisation and object deserialisation, and must be connected to other byte streams that act as destination and source for serialised objects. In order to store objects on a file, an `ObjectOutputStream` object is connected to a `FileOutputStream` object, and analogously, an `ObjectInputStream` object is connected to a `FileInputStream` object for reading objects from a file. We will demonstrate object serialisation and deserialisation by storing all employees in the company DOT-COM as objects on a file.



Writing objects

The method `writeObject()` in the class `ObjectOutputStream` is used to write objects. The parameter in this method is of the type `Object`. This means that any object can be passed to the method, and the method will serialise it. Given the reference `outObjStream`, that refers to an `ObjectOutputStream` object, and the reference `employee`, that refers to an `Employee` object, we can serialise this `Employee` object with the following method call:

```
outObjStream.writeObject(employee);
```

The method writes the state of an object, that comprises values of the *field variables* in the object. The state of an object also includes the field variables from the inheritance hierarchy. These data members are also included in the serialisation. Static data members are *not* included, as these belong to a class and not to objects.

How a field in an object is serialised, depends on whether the field has a primitive value, or it refers to an object. A primitive value in a field is written as a binary value. For a field that refers to an object, the reference is used to serialise the object, leading to the whole *object hierarchy* being written recursively, under the assumption that all objects in this hierarchy are serialisable. Given the reference `employees`, that refers to an array of employees in the DOT-COM company, the following method call will serialise the *whole* employee array, including the `Employee` objects in the array:

```
outObjStream.writeObject(employees);
```

The procedure for serialising objects to a file is as follows (see Figure 19.7 and Program 19.4):

- 1 Create a `FileOutputStream` object that opens the file for writing:

```
outFileStream = new FileOutputStream(dataFileName, append); // (3)
```

- 2 Create an `ObjectOutputStream` object that is connected to the `FileOutputStream` object from step 2:

```
outObjStream = new ObjectOutputStream(outFileStream); // (4)
```

- 3 Write objects with the `writeObject()` method. For example, the following method call writes the whole employee array:

```
outObjStream.writeObject(employees); // (5)
```

In addition to writing objects, we can also write primitive values as binary values, for example:

```
outObjStream.writeInt(numOfEmployees); // (6)
```

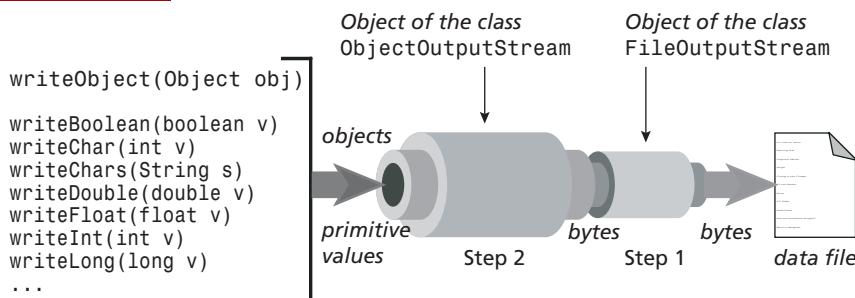
```
outObjStream.writeInt(numOfFemales); // (7)
```

- 4 Finish by closing the object output stream, which also closes the underlying output file:

```
outObjStream.close();
```



FIGURE 19.7 Writing objects and binary values



```

FileOutputStream outFileStream = new FileOutputStream(dataFileName, append); // Step 1
ObjectOutputStream outObjStream = new ObjectOutputStream(outFileStream); // Step 2

```

Program 19.4 illustrates serialising and deserialising of objects. To show that object serialisation can deal with all types of serializable objects, we include different types of employee objects in the personnel register, not only `Employee` objects at (1), as we have done earlier in connection with text and binary files. Since the `Employee` class is `Serializable`, its subclasses are also serialisable. Subclasses of the class `Employee` are shown to the extent it is necessary to understand the example. Complete source code can be downloaded from the book web site.

In the class `CompanyAdminObj`, the method `writeAllEmployeesToObjFile()` is called at (2a) for writing employee objects to a file. The method `writeAllEmployeesToObjFile()` creates the necessary streams at (3) and (4), as it is explained in step 1 and 2 above. The method writes the employee array at (5) and some statistics about the employees at (6) and (7). The file will then contain an employee array in serialised form, and two integers that are the number of employees and the number of women in the company. The streams are closed in a `finally` block at (8). It is necessary to use a try-catch statement in the `finally` block for handling an exception of the type `IOException` that the method `close()` can throw.

PROGRAM 19.4 Object serialisation

```

import java.io.IOException;
/*
 * Company administration using serialised objects on files.
 */
public class CompanyAdminObj {

    public static void main(String[] args) {
        // Create an array of different types of employees.
        Employee[] employees = { // (1)
            new Employee("Ole", "Olsen", 515.00, Gender.MALE),
            new PermanentEmployee("Bill", "Bailey", 612.00, Gender.MALE),
            new PieceWorker("Liv", "Larsen", Gender.FEMALE, 25.00, 75),
            new Manager("Molly", "Malone", 950.00, Gender.FEMALE, 150.00),
            new HourlyWorker("Sally", "Sage", 120.00, Gender.FEMALE)
        }
    }
}

```



```

};

// Create a personnel register.
PersonnelRegister register = new PersonnelRegister(employees);

// Create a company that uses files with serialised objects:
CompanyUsingObjFiles company = new CompanyUsingObjFiles(register);

// Serialise employee data in personnel register to a file.
company.writeAllEmployeesToObjFile("employeeFile.obj", false); // (2a)

// Deserialise employee data from a file.
System.out.println("Reading serialised objects from file.");
company.readAllEmployeesFromObjFile("employeeFile.obj"); // (2b)

// Print employee data to the terminal window.
company.printAllEmployeesToTerminalWindow();
company.printReport();
}

}

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

class CompanyUsingObjFiles {

    // Field
    private PersonnelRegister register;

    // Constructor
    CompanyUsingObjFiles() {
        register = new PersonnelRegister();
    }
    CompanyUsingObjFiles(PersonnelRegister register) {
        this.register = register;
    }

    // Write statistics
    void printReport() {
        System.out.println(register);
    }
    void printAllEmployeesToTerminalWindow() {
        Employee[] employees = register.getEmployeeArray();
        int numOfEmployees = register.getNumOfEmployees();
        System.out.println("Number of employees: " + numOfEmployees);
        for (int i = 0; i < numOfEmployees; i++) {
}

```



```
        System.out.println(employees[i]);
    }

/** 
 * The method serialises employee objects.
 */
void writeAllEmployeesToObjFile(String dataFileName, boolean append) {

    FileOutputStream outFileStream = null;
    ObjectOutputStream outObjStream = null;

    try {
        outFileStream = new FileOutputStream(dataFileName, append); // (3)
        outObjStream = new ObjectOutputStream(outFileStream);          // (4)

        Employee[] employees = register.getEmployeeArray();
        int numOfEmployees = register.getNumOfEmployees();
        int numOfFemales = register.getNumOfFemales();
        outObjStream.writeObject(employees);                           // (5)
        outObjStream.writeInt(numOfEmployees);                         // (6)
        outObjStream.writeInt(numOfFemales);                          // (7)
    }
    catch (FileNotFoundException exception) {
        System.out.print("Cannot open file: " + dataFileName);
        System.exit(1);
    }
    catch (IOException exception) {
        System.out.print("Error writing to file: " + exception);
        System.exit(1);
    }
    finally {
        try {
            if (outObjStream != null)
                outObjStream.close();                                // (8)
        } catch (IOException exception) {
            System.out.print("Error on closing: " + exception);
            System.exit(1);
        }
    }
}

/** 
 * The method deserialises employee objects.
 */
void readAllEmployeesFromObjFile(String dataFileName) {

    FileInputStream inFileStream = null;
    ObjectInputStream inObjStream = null;
    try {
```



```

inFileStream = new FileInputStream(dataFileName);           // (9)
inObjStream = new ObjectInputStream(inFileStream);         // (10)

Employee[] employees = null;
// Read an object from the file.
Object newObject = inObjStream.readObject();              // (11)
// Check whether the object is of the right type.
if (newObject instanceof Employee[])
    employees = (Employee[]) newObject; // Downcasting      // (12)
else
    System.out.println("Employee array not found.");
int numOfEmployees = inObjStream.readInt();                // (13)
int numOfFemales = inObjStream.readInt();                  // (14)
register = new PersonnelRegister(
    employees, numOfEmployees, numOfFemales);             // (15)
}

catch (FileNotFoundException exception) {
    System.out.print("Cannot open file: " + dataFileName);
    System.exit(1);
}
catch (IOException exception) {
    System.out.print("Error reading from file: " + exception);
    System.exit(1);
}
catch (ClassNotFoundException exception) {
    System.out.print("Class definition not found: " + exception);
    System.exit(1);
}
finally {
    try {
        if (inObjStream != null)
            inObjStream.close();
    } catch (IOException exception) {
        System.out.print("Error on closing: " + exception);
        System.exit(1);
    }
}
}

class PersonnelRegister { // PersonnelRegister objects not serialised.
// See the class PersonnelRegister in Program 11.1 on page 295.
}

class Employee implements Serializable { // Subclasses become serialisable.
// See the class Employee in Program 11.1 on page 295.
}

final class PermanentEmployee extends Employee {
// Constructors
}

```



```
public PermanentEmployee() { /* Default values for all fields. */ }
public PermanentEmployee(String firstName, String lastName,
                        double hourlyRate, Gender gender) {
    super(firstName, lastName, hourlyRate, gender);
}

// Overrides method in the superclass Employee.
public double computeSalary(double numOfHours) {
    return hourlyRate * Employee.NORMAL_WORKWEEK;
}

public String toString() {
    return super.toString() + "\nis permanent employee.";
}
}

final class PieceWorker extends Employee {

    // New fields
    private double pricePerUnit;
    private int numOfUnits;

    // Constructors
    public PieceWorker() { /* Default values of fields. */ }
    public PieceWorker(String firstName, String lastName, Gender gender,
                      double pricePerUnit, int numOfUnits) {
        super(firstName, lastName, 0.0, gender);
        this.pricePerUnit = pricePerUnit;
        this.numOfUnits = numOfUnits;
    }

    public String toString() {
        return super.toString()
            + "\nis paid " + String.format("%.2f", pricePerUnit) +
            " per unit, and has made " + numOfUnits + " units.";
    }

    // Overrides method in the superclass Employee
    public double computeSalary(double numOfHours) { // Ignores parameter.
        return computeSalary();
    }

    // New instance methods
    public double computeSalary() { return numOfUnits * pricePerUnit; }

    // ...
}

final class Manager extends Employee {
```



```

// New field
private double managerBonus;

// Constructors
public Manager() { /* Default value for all fields. */ }
public Manager(String firstName, String lastName,
              double hourlyRate, Gender gender,
              double bonus) {
    super(firstName, lastName, hourlyRate, gender);
    managerBonus = bonus;
}

public String toString() {
    return super.toString() + "\ngets " +
        String.format("%.2f", managerBonus) + " as manager bonus.";
}

// Overrides method in the superclass Employee
public double computeSalary(double numOfHours) {
    assert numOfHours >= 0 : "Number of hours worked must be >= 0";
    double weeklySalary = super.computeSalary(numOfHours);
    weeklySalary += managerBonus;
    return weeklySalary;
}

// ...
}

final class HourlyWorker extends Employee {

    // Constructors
    public HourlyWorker() { /* Default values for fields. */ }

    public HourlyWorker(String firstName, String lastName,
                        double hourlyRate, Gender gender) {
        super(firstName, lastName, hourlyRate, gender);
    }

    // Overrides method from the superclass Employee.
    public double computeSalary(double numOfWorkers) {
        assert numOfWorkers >= 0 : "Number of workers should be >= 0";
        return numOfWorkers * hourlyRate;
    }

    public String toString() {
        return super.toString() + "\nis paid by the hour.";
    }
}

```



Program output:

```
Reading serialised objects from file.
Number of employees: 5
First name: Ole    Last name: Olsen    Hourly rate: 515.00 Gender: MALE
First name: Bill   Last name: Bailey   Hourly rate: 612.00 Gender: MALE
is permanent employee.
First name: Liv    Last name: Larsen   Hourly rate: 0.00 Gender: FEMALE
is paid 25.00 per unit, and has made 75 units.
First name: Molly  Last name: Malone   Hourly rate: 950.00 Gender: FEMALE
gets 150.00 as manager bonus.
First name: Sally  Last name: Sage     Hourly rate: 120.00 Gender: FEMALE
is paid by the hour.
The company has 5 employees, where 60.00% are women.
```

Reading objects

During deserialisation, we must make sure that the correct number of objects are read in the right order. Other binary values stored in the file must also be read in the right sequence.

The method `readObject()` in the `ObjectInputStream` class is used to read serialised objects. The method returns a reference of type `Object` that refers to the deserialised object. Given the reference `inObjStream` that refers to an `ObjectInputStream` object, the following call will read an object from the underlying input stream:

```
Object objRef = inObjStream.readObject();
```

The method `readObject()` can, in addition to throwing an exception of the type `IOException`, also throws an exception of the type `ClassNotFoundException`. This exception reports that the class necessary for creating the object cannot be found. The method `readObject()` uses the serialised information together with the class definition to create the object. The method creates the complete object state recursively, including instance members specified in the object's inheritance hierarchy.

The procedure for deserialising objects from a file is as follows (see Figure 19.8 and Program 19.4):

- 1 Create a `FileInputStream` object that opens the file to read objects and binary values from:

```
inFileStream = new FileInputStream(dataFileName); // (9)
```

- 2 Create an `ObjectInputStream` object which is connected to the `FileInputStream` object from step 1:

```
inObjStream = new ObjectInputStream(inFileStream); // (10)
```

- 3 Read an object with the `readObject()` method. The program must keep track of the *type* of the object read, before calling methods defined specifically in the object's type definition.

```
// Read an object from the file.
```

```
Object newObject = inObjStream.readObject(); // (11)
```



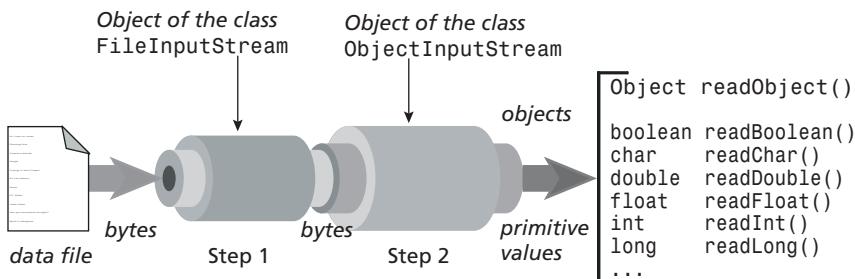
```
// Check whether the object is of the right type.
if (newObject instanceof Employee[])
    employees = (Employee[]) newObject; // Downcasting           // (12)
else
    System.out.println("Employee array not found.");
```

All objects that belong in the employee array are also deserialised when the employee array is read, resulting in the array having the same number and same type of objects that it had when it was serialised (see output from Program 19.4).

- 4** Finish by closing the input stream, which also closes the underlying input file:

```
inObjStream.close();
```

FIGURE 19.8 Reading objects and binary values



```
FileInputStream inFileStream = new FileInputStream(dataFileName); // Step 1
ObjectInputStream inObjStream = new ObjectInputStream(inFileStream); // Step 2
```

The class `CompanyAdminObj` in Program 19.4 calls the method `readAllEmployeesFromObjFile()` on an object of the class `CompanyUsingObjFiles` for reading employee objects from the input file at (2b). This method creates the necessary input streams for deserialising of objects from the file at (9) and (10). The employee array is read at (11) and checked in an if statement at (12). The remaining information is read at (13) and (14). A new personnel register is created at (15), based on the values read. The output shows that the array was created with the same type of objects as the array that was serialised.

The method `readAllEmployeesFromObjFile()` catches the necessary exceptions, and closes the input streams in a safe way, thus saving the `main()` method from dealing with any exceptions.

Effective storing of objects

If we were to use text files or binary files to store different *types* of employee objects, the program would have to keep track of which type of employees were being stored, and make sure that the correct type of employee was created when the information was read. This could, for example, be done by storing an integer code in each record in the file, that indicated which type of object the information was valid for. Object serialisation, on the other hand, allows us to store different types of objects without this extra book keeping. The responsibility of using the deserialised object correctly still lies with the program, as we have seen in step 3 above.



The serialisation process is sequential. This means that objects are deserialised in the order they were serialised. The serialisation mechanism avoids duplication of objects. If we serialise an object that has already been serialised, the process marks this fact in the output stream. When this marker is read during deserialisation, the object has already been deserialised, so the reference value of this object is returned. Program 19.5 illustrates that duplication of objects is avoided during serialisation.

PROGRAM 19.5 Object serialisation without duplication

```

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerialisingWithoutDuplication {

    public static void main(String[] args) {

        // (1) Create a number of advice:
        StringBuilder advice0 = new StringBuilder("Speak no evil");
        StringBuilder[] array0 = {
            advice0,                                     // (2) Common advice.
            new StringBuilder("Hear no evil"),
            new StringBuilder("See no evil")
        };

        String fileName = "advice.data";

        try {
            // Create streams for serialisation.          (3)
            FileOutputStream outFile = new FileOutputStream(fileName, false);
            ObjectOutputStream outStream = new ObjectOutputStream(outFile);

            // Write array0 and advice0 twice.           (4)
            outStream.writeObject(array0);
            outStream.writeObject(advice0);
            outStream.writeObject(array0);
            outStream.writeObject(advice0);

            outStream.close();

            // Create streams for serialisation.          (5)
            FileInputStream inFile = new FileInputStream(fileName);
            ObjectInputStream inStream = new ObjectInputStream(inFile);

            // Read serialised objects in the right order. (6)
            StringBuilder array1[] = (StringBuilder[])inStream.readObject();
            StringBuilder advice1 = (StringBuilder)inStream.readObject();
        }
    }
}

```



```
StringBuilder array2[] = (StringBuilder[])inStream.readObject();
StringBuilder advice2 = (StringBuilder)inStream.readObject();

// Test if two references are equal,i.e. if they refer
// to the same object.                                         (7)
System.out.println("array0 and array1 refer to same object: "
+ (array0 == array1));
System.out.println("advice0 and array1[0] refer to same object: "
+ (advice0 == array1[0]));
System.out.println("array1 and array2 refer to the same object: "
+ (array1 == array2));
System.out.println("advice1 and advice2 refer to the same object: "
+ (advice1 == advice2));
System.out.println(
    "array1[0] and array2[0] refer to the same object: "
    + (array1[0] == array2[0]));
System.out.println(
    "array1[0] and advice2 refer to the same object: "
    + (array1[0] == advice2));

// Test if two objects are equal.                                 (8)
System.out.println(
    "Objects referenced by advice0 and array1[0] " +
    "have the same state: " +
    advice0.toString().equals(array1[0].toString()));

    inStream.close();
}
catch (IOException exception) {
    System.out.print("Error on file I/O: " + exception);
    System.exit(1);
}
catch (ClassNotFoundException exception) {
    System.out.print("Class definition not found: " + exception);
    System.exit(1);
}
}
```

Program output:

```
array0 and array1 refer to same object: false
advice0 and array1[0] refer to same object: false
array1 and array2 refer to the same object: true
advice1 and advice2 refer to the same object: true
array1[0] and array2[0] refer to the same object: true
array1[0] and advice2 refer to the same object: true
Objects referenced by advice0 and array1[0] have the same state: true
```



BEST PRACTICES

Object serialisation is a powerful mechanism for storing and retrieving objects from streams. In many cases it can prove to be a better choice than saving field values individually.

BEST PRACTICES

Many methods that read and write to streams throw checked exceptions. Use several `catch` blocks if necessary to catch *specific* exceptions in your program, rather than using a superclass to catch as many as possible.

BEST PRACTICES

Make sure to close the streams properly after you are done with them. A `finally` block is ideal for ensuring that the code for this purpose is always executed.

19.5 Random access files

Binary files and text files allow either sequential reading from or sequential writing on files. This is called *sequential access*. It is not possible to combine both operations on a sequential file. If we want to read from such a file, the reading will always begin at the start of the file. During writing we have two choices. The writing can either begin from the start of the file, or after values that are already stored in the file. In the first case, any previous contents in the file is overwritten. In the second case, the file is extended with new content.

Overview of the class `RandomAccessFile`

The class `RandomAccessFile` provides what is called a *random access file*. This means that it is possible to both read and write from an arbitrary position in a file.

Table 19.1 gives an overview of selected methods in the class `RandomAccessFile`. The class is not part of the inheritance hierarchy of byte streams and character streams we have seen so far, but inherits directly from the `Object` class. On the other hand, it does implement the interfaces `DataInput` and `DataOutput`, that define methods for reading and writing of binary presentation of primitive data values.

All methods in the class `RandomAccessFile` can throw an exception of the type `IOException` that must be handled by the program. A `RandomAccessFile` stream must also be closed when it is no longer needed.



BEST PRACTICES

If random access provided by the `RandomAccessFile` class is not important, consider using the byte streams `DataInputStream` and `DataOutputStream`. These two byte streams handle binary streams sequentially, and chaining them with other streams can be exploited.

TABLE 19.1 Selection of methods for random access files

<code>java.io.RandomAccessFile</code>	
<code>RandomAccessFile(String fileName, String fileMode)</code> throws <code>FileNotFoundException</code>	Open the file with the specified name. The file mode "r" indicates that values can only be read from the file, and the file mode "rw" allows both reading values from and writing values to the file. The file is created for writing if it does not exist from before.
<code>long length()</code>	Returns the number of bytes stored in the file, i.e. the <i>length</i> of the file.
<code>void setLength(long newLength)</code>	The length of the file is set equal to the parameter value. For example, the call <code>setLength(0)</code> sets the file length to 0, i.e. the previous contents of the file are no longer available, and the file can be considered empty.
<code>long getFilePointer()</code>	Returns the current file pointer position.
<code>void seek(long displacement)</code>	Moves the file pointer to the position given by the parameter value. The new position is always calculated from the start of the file (given by the value 0).
<code>int skipBytes(int n)</code>	Skips n bytes from the current position in the file, moves the file pointer past the n bytes.
<code>boolean readBoolean()</code> <code>char readChar()</code> <code>double readDouble()</code> <code>float readFloat()</code> <code>int readInt()</code> <code>long readLong()</code>	Reads a binary value, depending on the method called. These methods throw an exception of the type <code>EOFException</code> , if we try to read data and it is the end of the file.
<code>void writeBoolean(boolean v)</code> <code>void writeChar(int v)</code> <code>void writeDouble(double v)</code> <code>void writeFloat(float v)</code> <code>void writeInt(int v)</code> <code>void writeLong(long v)</code>	Writes a binary value, using a number of bytes equal to the size of the primitive type.



java.io.RandomAccessFile

<code>void writeChars(String s)</code>	Writes each character in the string as two bytes to the output stream.
<code>void close()</code>	Closes the file.

Setting up random access

For setting up random access to a file, the file must be opened by specifying the file name when a `RandomAccessFile` object is created:

```
try {
    RandomAccessFile raf = new RandomAccessFile(dataFileName, "rw");
} catch (FileNotFoundException exception) { ... }
```

The value of the second parameter in the constructor specifies the *file mode*. The file mode "rw" allows combining both reading and writing operations in the file. A new empty file is created with the specified file name if the file does not exist from before. The second file mode, "r", is for reading from an existing file only. In this case, if the file does not exist, an exception of the type `FileNotFoundException` is thrown. A file contains bytes regardless of what these bytes represent. The number of bytes in a file is called the *file length*, which is returned by the method `length()`.

The file pointer

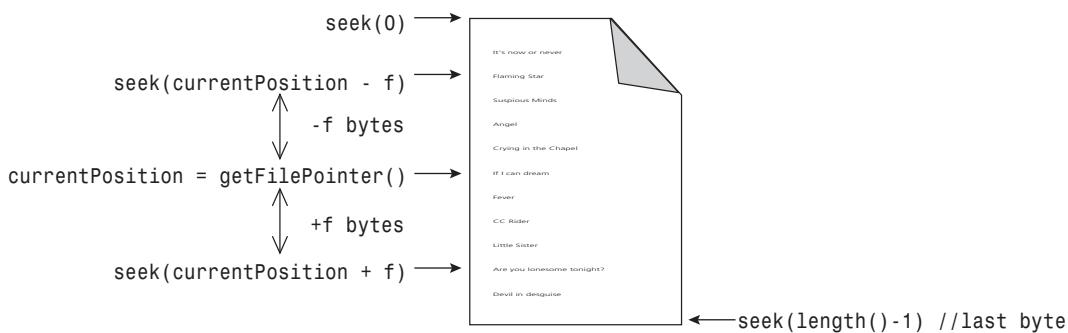
A *file pointer* indicates the position in the file where reading or writing can be done. When a file is opened, the file pointer is always positioned at the start of the file. We can think of the file pointer as a *cursor* between two consecutive bytes in the file. Reading and writing takes place at the byte position *after* this cursor, and the cursor is automatically moved forward, past the number of bytes that are read or written.

Figure 19.9 shows how the file pointer can be moved. File pointer value 0 indicates the start of the file, and the file pointer value (`raf.length()-1`) indicates the last byte in the file. The current position of the file pointer is returned by the method `getFilePointer()`. The method `seek()` can be used to position the file pointer. Figure 19.9 also illustrates how the file pointer can be moved forward or backwards relative to its current position. We see that the positioning of the file pointer with the method `seek()` is always done by specifying the appropriate number of bytes from the start of the file.



FIGURE 19.9

Random access file



If we are only interested in moving the file pointer forward in the file from the current position, we can also use the method `skipBytes()`. Its parameter specifies the number of bytes to skip from the current file pointer position. For example, the following code line can be used to skip five characters from the current position, given that each character is stored as two bytes:

```
raf.skipBytes(10);
```

If we want to extend a file with new data, the file pointer value can be set equal to the file length, i.e. past the last byte in the file, before the write operation is executed:

```
raf.seek(raf.length()); // Set the file pointer to end of file.
```

Reading and writing on random access file

Program 19.6 shows the two classes `CompanyAdminRandomAccess` and `CompanyUsingRandomAccess` that illustrate using a random access file. These classes maintain the employees of the DOT-COM company using the `Employee` and `PersonnelRegister` classes from Program 11.2 on page 300.

Handling records with fixed length

The class `CompanyUsingRandomAccess` reads and writes an employee record in the position indicated by the file pointer. If a particular record is to be read or written in the file, the file pointer must be set correctly. The program should be able to determine the position of any record in the file. It is easier to set the file pointer correctly if all records have the same fixed length. This means that the same field in all records has the same fixed field length. The class `CompanyUsingRandomAccess` defines the different constants at (6) for handling fixed-length records for the employees.

The number of bytes a value of a primitive data type requires is defined in Java (see Table 11.1). We use the corresponding `readX()` or `writeX()` methods that ensure correct binary reading or writing of such values.

For a field with a string value to have a fixed length, we can use the following solution: if the string length is less than the field length, the field is padded with a special character. If the string length is greater than the field length, the string is truncated so that it fits in the field. The method `writeFixedLengthString()` at (9) takes the string length into consideration and always makes sure that the whole field is padded:



```

int numOfPadChars = maxFieldLength - str.length();
StringBuilder padding = new StringBuilder();
for (int k = 0; k < numOfPadChars; k++) {
    padding.append(PAD_CHAR);
}
raf.writeChars(str + padding);

```

Analogously the method `readFixedLengthString()` at (12) ensures that a field containing a string is read correctly. It reads characters from the file until the character used for padding is read. It then skips the remaining padding characters in the file:

```

StringBuilder strBuffer = new StringBuilder();
int i;
for (i = 0; i < maxFieldLength; i++) {
    char character = raf.readChar();
    if (character == PAD_CHAR)
        break;
    else
        strBuffer.append(character);
}
// Skip any PAD_CHARS, each of size 2 bytes.
raf.skipBytes(2*(maxFieldLength - i - 1));

```

Note the use of a `StringBuilder` above. We avoid creating many `String` objects, since a `StringBuilder` object is mutable, unlike a `String` object.

PROGRAM 19.6 Handling of employee records using random access file

```

class Employee {
    // See the class Employee in Program 11.1 on page 295.
}

class PersonnelRegister {
    // See the class PersonnelRegister in Program 11.1 on page 295.
}

/*
 * Company administration using random access files.
 */
class CompanyAdminRandomAccess {

    public static void main(String[] args) {
        // Create an array with different types of employees.
        Employee[] employees = {
            new Employee("Ole", "Olsen", 515.00, Gender.MALE),
            new Employee("Bill", "Bailey", 612.00, Gender.MALE),
            new Employee("Liv", "Larsen", 820.00, Gender.FEMALE)
        };

        // Create a personnel register.
        PersonnelRegister register = new PersonnelRegister(employees);
    }
}

```



```

// Create a company                                (1a)
CompanyUsingRandomAccess company = new CompanyUsingRandomAccess(register);
System.out.println("Employees to be stored:");
company.printAllEmployeesToTerminalWindow();

// Initialise for random access.
company.initRandomAccessFile("employeeFile.data");           // (1b)

// Write all employees to the file.
int numOfEmployees = register.getNumOfEmployees();
for (int i = 0; i < numOfEmployees; i++)
    company.appendFileWithEmployee(register.getEmployee(i));      // (2)

// Update hourly rate directly on file.
company.registerNewHourlyRateForEmployeeOnFile(2, 100.0);        // (3)
company.registerNewHourlyRateForEmployeeOnFile(0, 300.0);
company.registerNewHourlyRateForEmployeeOnFile(1, 400.0);

System.out.println("Read all employees from random access file.");
company.readAllEmployeesFromRandomAccessFile();                  // (4)

// Write statistics.
company.printAllEmployeesToTerminalWindow();
company.printReport();

company.closeStream();                                         // (5)
}

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;

class CompanyUsingRandomAccess {

    final static char FIELD_TERMINATOR    = ',';

    // Constants for handling records with fixed length.          (6)
    final static char PAD_CHAR            = '\u0000';
    final static int FIRSTNAME_LENGTH     = 20; // Unicode character
    final static int LASTNAME_LENGTH      = 30; // Unicode character
    final static int HOURLYRATE_LENGTH    = 8;  // bytes
    final static int GENDER_LENGTH        = 1;  // byte
    final static int RECORD_LENGTH_BYTES = 2*FIRSTNAME_LENGTH +
                                         2*LASTNAME_LENGTH +
                                         HOURLYRATE_LENGTH +
                                         GENDER_LENGTH; // bytes

    // Random access file.
    private RandomAccessFile raf;
}

```



```
// Register with the employees.  
private PersonnelRegister register;  
  
// Constructors  
CompanyUsingRandomAccess() {  
    register = new PersonnelRegister();  
}  
CompanyUsingRandomAccess(PersonnelRegister register) {  
    this.register = register;  
}  
  
// Print statistics.  
void printReport() {  
    System.out.println(register);  
}  
void printAllEmployeesToTerminalWindow() {  
    Employee[] employees = register.getEmployeeArray();  
    int numOfEmployees = register.getNumOfEmployees();  
    System.out.println("No. of employees: " + numOfEmployees);  
    for (int i = 0; i < numOfEmployees; i++) {  
        System.out.println(employees[i]);  
    }  
}  
  
/** Method opens a file for reading and writing. */  
void initRandomAccessFile(String dataFileName) { // (7)  
  
    try {  
        // Open file for reading and writing.  
        raf = new RandomAccessFile(dataFileName, "rw");  
        // Reset the file pointer.  
        raf.setLength(0);  
        // Reset counter for number of records on the file.  
        raf.writeInt(0);  
    }  
    catch (FileNotFoundException exception) {  
        System.out.print("File not found: " + exception);  
        System.exit(1);  
    }  
    catch (IOException exception) {  
        System.out.print("Error writing to file: " + exception);  
        System.exit(1);  
    }  
}  
  
/** Method appends an Employee record to the file. */  
void appendFileWithEmployee(Employee employee) { // (8)  
    try {  
        // Position file pointer after the last record.
```



```

raf.seek(raf.length());
// Write the employee data to the file.
writeEmployeeData(employee);

// Update the number of records in the file.
raf.seek(0);
int numOfEmployeesRegistered = raf.readInt();
raf.seek(0);
raf.writeInt(numOfEmployeesRegistered + 1);
}

catch (IOException exception) {
    System.out.print("Error writing to file: " + exception);
}
}

/** Method writes field values of an employee. */
void writeEmployeeData(Employee employee) throws IOException {
    writeFixedLengthString(employee.firstName, FIRSTNAME_LENGTH);
    writeFixedLengthString(employee.lastName, LASTNAME_LENGTH);
    raf.writeDouble(employee.hourlyRate);
    raf.writeByte(employee.gender.ordinal());
}

/** Method writes a string (padding, if necessary) of max. length. */
private void writeFixedLengthString(String str, int maxFieldLength) // (9)
    throws IOException {
    // Field padded with PAD_CHAR if the length
    // is less than max length.
    int numPadChars = maxFieldLength - str.length();
    StringBuilder padding = new StringBuilder();
    for (int k = 0; k < numPadChars; k++) {
        padding.append(PAD_CHAR);
    }
    raf.writeChars(str + padding);
}

/** Method changes the hourly rate of an Employee record in the file. */
void registerNewHourlyRateForEmployeeOnFile(int employeeNumber,
                                             double hourlyRate) { // (10)
    try {
        // Set file pointer to the correct position.
        raf.seek(4 + employeeNumber * RECORD_LENGTH_BYTES);
        // Read the employee record and update hourly rate.
        Employee employee = readEmployeeData(); // (11)
        employee.hourlyRate = hourlyRate;

        // Position file pointer to rewrite the updated employee record.
        long currentPosition = raf.getFilePointer();
        raf.seek(currentPosition - RECORD_LENGTH_BYTES);
        writeEmployeeData(employee);
    }
}

```



```

    }
    catch (IOException exception) {
        System.out.print("Error reading from file: " + exception);
    }
}

/** Method reads an employee record from current position in the file. */
Employee readEmployeeData() throws IOException {
    String firstName = readFixedLengthString(FIRSTNAME_LENGTH);
    String lastName = readFixedLengthString(LASTNAME_LENGTH);
    double hourlyRate = raf.readDouble();
    Gender gender = null;
    switch (raf.readByte()) { // Read ordinal value for gender.
        case 0: default:
            gender = Gender.FEMALE;
            break;
        case 1:
            gender = Gender.MALE;
            break;
    }
    return new Employee(firstName, lastName, hourlyRate, gender);
}

/** Method reads a string of specified max. length. */
private String readFixedLengthString(int maxFieldLength)
    throws IOException { // (12)
    StringBuilder strBuffer = new StringBuilder();
    int i;
    for (i = 0; i < maxFieldLength; i++) {
        char character = raf.readChar();
        if (character == PAD_CHAR)
            break;
        else
            strBuffer.append(character);
    }
    // Skip any PAD_CHARS, each of size 2 bytes.
    raf.skipBytes(2*(maxFieldLength - i - 1));
    return strBuffer.toString();
}

/** Method reads all employee records. */
void readAllEmployeesFromRandomAccessFile() { // (13)
    try {
        // Set file pointer to beginning of file.
        raf.seek(0);

        // Create a new personnel register.
        register = new PersonnelRegister();

        // Read how many employees are in the file.
    }
}

```



```

        int numOfEmployees = raf.readInt();

        // Read each employee data sequentially.
        for (int i = 0; i < numOfEmployees; i++) {
            // Read an employee record.
            Employee employee = readEmployeeData();
            // Insert employee in the register.
            register.registerEmployee(employee);
        }
    }

    catch (IOException exception) {
        System.out.print("Error reading from file: " + exception);
        System.exit(1);
    }
}

/** Method closes files for random access. */
void closeStream() {
    try {
        if (raf != null)
            raf.close();
    }
    catch (IOException exception) {
        System.out.print("Error on closing: " + exception);
        System.exit(1);
    }
}

/** Method changes hourly rate of an employee and also updates the file. */
void setHourlyRate(int employeeNumber, double hourlyRate) {
    try {
        register.getEmployee(employeeNumber).hourlyRate = hourlyRate;
        writeEmployeeToRandomAccessFile(employeeNumber);
    }
    catch (ArrayIndexOutOfBoundsException exception) {
        System.out.print("Invalid employee number: " + exception);
    }
}

/** Method updates an employee record in the file. */
void writeEmployeeToRandomAccessFile(int employeeNumber) {
    try {
        // Position the file pointer.
        // Must add 4 bytes (int) at the start of the file used
        // for the number of records in the file.
        raf.seek(4 + (employeeNumber)* RECORD_LENGTH_BYTES);
        // Write employee data to file.
        writeEmployeeData(register.getEmployee(employeeNumber));
    }
    catch (IOException exception) {

```



```

        System.out.print("Error writing to file: " + exception);
    }
    catch (ArrayIndexOutOfBoundsException exception) {
        System.out.print("Invalid employee number: " + exception);
    }
}

/** Method writes all employees to the file. */
void writeAllEmployeesToRandomAccessFile() {
    try {
        // Set the file pointer to the start of the file.
        raf.seek(0);

        // Write how many records will be stored in the file.
        int numOfEmployees = register.getNumOfEmployees();
        raf.writeInt(numOfEmployees);

        // Write data about each employee to the file.
        Employee[] employeeArray = register.getEmployeeArray();
        for (int i = 0; i < numOfEmployees; i++)
            writeEmployeeData(employeeArray[i]);
    }
    catch (IOException exception) {
        System.out.print("Error writing to the file: " + exception);
        System.exit(1);
    }
}

/** Method reads an employee record from the file. */
void readEmployeeFromRandomAccessFile(int employeeNumber) {
    try {
        // Position the file pointer.
        // Must add 4 bytes (int) at the start of the file used
        // for number of records in the file.
        raf.seek(4 + (employeeNumber)* RECORD_LENGTH_BYTES);
        // Read the employee record at the current position.
        Employee employee = readEmployeeData();
        // Register in the personnel register.
        register.replaceEmployee(employeeNumber, employee);
    }
    catch (IOException exception) {
        System.out.print("Error reading from file: " + exception);
    }
}
}

```

Program output:

```

Employees to be stored:
No. of employees: 3
First name: Ole    Last name: Olsen    Hourly rate: 515.00 Gender: MALE

```



```

First name: Bill   Last name: Bailey   Hourly rate: 612.00 Gender: MALE
First name: Liv    Last name: Larsen   Hourly rate: 820.00 Gender: FEMALE
Read all employees from random access file.
No. of employees: 3
First name: Ole    Last name: Olsen   Hourly rate: 300.00 Gender: MALE
First name: Bill   Last name: Bailey   Hourly rate: 400.00 Gender: MALE
First name: Liv    Last name: Larsen   Hourly rate: 100.00 Gender: FEMALE
The company has 3 employees, where 33.33% are women.

```

Initialising for random access file

The class `CompanyAdminRandomAccess` creates a company object at (2), and at (3) calls the `initRandomAccessFile()` method to initialise for random access file. At (7) this method creates a `RandomAccessFile` object in a try block, that is associated with the specified data file. It first *resets* the contents of the file by setting the file length to zero. Thereafter it registers the value 0 in the file to indicate that no records have been stored yet:

```

// Open file for reading and writing.
raf = new RandomAccessFile(dataFileName, "rw");
// Reset the file pointer.
raf.setLength(0);
// Reset counter for number of records in the file.
raf.writeInt(0);

```

Extending a file with new records

At (2) the class `CompanyAdminRandomAccess` calls the method `appendFileWithEmployee()` repeatedly to write information about all employees in the file. At (8) this method extends the file with a new record each time it is called:

```

// Position file pointer after the last record.
raf.seek(raf.length());
// Write the employee info to the file.
writeEmployeeData(employee);

```

The method updates information about the number of records stored at the start of the file:

```

raf.seek(0);
int numOfEmployeesRegistered = raf.readInt();
raf.seek(0);
raf.writeInt(numOfEmployeesRegistered + 1);

```

Random access of records

At (3) the class `CompanyAdminRandomAccess` calls the method `registerNewHourlyRateForEmployeeOnFile` to change, directly in the file, the hourly rate for an employee, given by an *employee number*. At (10) this method refers to the records in the file using an index value, 0 being the index of the first record in the file. We can position the file pointer at the record that corresponds to the index given by the `employeeNumber`, as follows:



```
raf.seek(4 + employeeNumber * RECORD_LENGTH_BYTES);
```

After reading the record of the employee at (11), the hourly rate is adjusted. The file pointer is moved back one record so that the old field values in this record also can be overwritten:

```
long currentPosition = raf.getFilePointer();
raf.seek(currentPosition - RECORD_LENGTH_BYTES);
writeEmployeeData(employee);
```

Sequential reading of records

At (4) the class `CompanyAdminRandomAccess` calls the method `readAllEmployeesFromRandomAccessFile()` to read all employee records from the file. At (13) this method creates a new register in which the new `Employee` objects are registered. It sets the file pointer to the start of the file and reads one record at a time, sequentially.

The output from the program shows that 3 records were stored in the file, and that the information about the hourly rate for the employees was correctly updated in the file.

19.6 Review questions

1. An _____ stream writes data from the program to a destination.
An _____ stream reads data from a source to the program.
An output stream writes data from the program to a destination.
An input stream reads data from a source to the program.
2. Which statements are true about streams?
 - a A byte stream handles 8-bit binary data.
 - b A character stream handles 16-bit Unicode characters.
 - c All classes that inherit from the `Reader` and `Writer` classes represent byte streams.
 - d All classes that inherit from the `InputStream` and `OutputStream` classes represent character streams.
(a) and (b)
Classes that inherit from Reader and Writer represent character streams, while classes that inherit from InputStream and OutputStream represent byte streams.
3. Which classes in the Java standard library define inheritance hierarchies for reading from and writing to byte streams?
 - a The abstract classes `InputStream` and `OutputStream` and their subclasses
 - b The abstract classes `Reader` and `Writer`
 - c The classes `FileWriter` and `FileReader`
(a)



`InputStream` and `OutputStream` are superclasses for byte streams, while `Reader` and `Writer` are superclasses for character streams.

4. Which classes have methods for writing binary representation of primitive data values?

- a `FileOutputStream`
- b `DataOutputStream`
- c `PrintStream`
- d `Writer`
- e `ObjectOutputStream`
- f `RandomAccessFile`

(b), (e), (f)

The classes `DataOutputStream`, `ObjectOutputStream` and `RandomAccessFile` implement the interface `DataOutput`, that defines methods for writing binary representation of primitive data values.

5. Which of these files contain binary data:

- a a file with Java byte code
- b a file with Java source code
- c a file with image data
- d a file with audio data

Binary data in (a), (c) and (d). Text in (b).

6. The classes _____ and _____ can be used to create byte streams for sequential files.

The classes `FileInputStream` and `FileOutputStream` can be used to create byte streams for sequential files.

7. Which classes have constructors that accept a file name as parameter?

- a `DataOutputStream`
- b `PrintWriter`
- c `FileOutputStream`
- d `ObjectOutputStream`
- e `BufferedReader`

(b), (c)

The others can be connected to an underlying data stream.

8. Which methods are found in the class `ObjectOutputStream`?

- a `writeInt(int in)`
- b `writeChar(int in)`



c `writeChar(char t)`

(a), (b)

Integers in (b) are written as a 16-bit character.

- 9.** What does a `readX()` method in the `DataInputStream` class do when there is no more data?

- a** It returns the value `null`.
- b** It returns the string "EOF" (End Of File).
- c** It returns the value -1.
- d** It throws an exception of the type `EOFException`.

(d)

- 10.** The class _____ has the method _____ that can be used to serialise objects to an output stream. The type of the parameter is _____.

The class _____ has the method _____ that can be used to read serialised objects from an input stream. The return type of this method is _____.

The class `ObjectOutputStream` has the method `writeObject()` that can be used to serialise objects to an output stream. The type of the parameter is `Object`.

The class `ObjectInputStream` has the method `readObject()` that can be used to read serialised objects from an input stream. The return type of this method is `Object`.

- 11.** Which interface must a class implement so that its objects can be serialised? Which methods are defined by this interface?

The interface is called `Serializable` and it has no methods. It is only a marker that the objects of the class can be serialised.

- 12.** Three byte streams are opened automatically for a Java program.

_____ is an object of the class `InputStream`, and can be used to read bytes from the keyboard.

_____ is an object of the class `OutputStream`, and can be used to write to the terminal window.

_____ is an object of the class `PrintStream`, and is meant for writing errors to the terminal window.

`System.in` is an object of the class `InputStream`, and can be used to read bytes from the keyboard.

`System.out` is an object of the class `PrintStream`, and can be used to write to the terminal window.

`System.err` is an object of the class `PrintStream`, and is meant for reporting errors to the terminal window.



- 13.** Explain what kind of streams are created below.

- a** `FileOutputStream file1 = new FileOutputStream("myFile", true);
DataOutputStream stream1 = new DataOutputStream(file1);`
- b** `FileReader file2 = new FileReader("myFile");
BufferedReader stream2 = new BufferedReader(file2);`
- c** `InputStreamReader file3 = new InputStreamReader(System.in);
BufferedReader stream3 = new BufferedReader(file3);`
- d** `FileOutputStream file4 = new FileOutputStream("myFile");
ObjectOutputStream stream4 = new ObjectOutputStream(file4);`

- (a) Creates a stream for writing binary representation of data values to the binary file "myFile". If the file exists from before, it is extended with the new data.
- (b) Creates a stream for reading text lines from the text file "myFile".
- (c) Creates a stream for reading text lines from the keyboard.
- (d) Creates a stream for writing objects to the file "myFile". The contents of the file are overwritten, if the file exists from before.

- 14.** Assume that the variable `objInStream` refers to an object of the class `ObjectInputStream`. This stream can be used for reading objects of different types, among them those of the class `Employee`. What is wrong with the following code for reading an `Employee` object:

```
Employee employee = objInStream.readObject();
```

The method `readObject()` returns the reference value of the object read in a reference of the type `Object`. The assignment above is downcasting, and requires an explicit cast.

If the object read is not of the type `Employee`, the type conversion will result in an exception of the type `ClassCastException` at runtime. Before applying the cast, the `instanceof` operator can be used to test whether the object is actually of the type `Employee`:

```
Object objRef = objInStream.readObject();  
if (objRef instanceof Employee) {  
    Employee employee = (Employee) objRef;  
    ...  
} else {  
    System.out.println("Not an employee.");  
}
```

- 15.** Which file modes are valid for the class `RandomAccessFile`.

- a** "R"
- b** "r"
- c** "RW"
- d** "rw"



(b), (d)

The file mode "rw" will create the file if it does not exist (under the assumption that the file permissions allow it).

16. Which statements about the file pointer are true?

- a The method `seek(long displacement)` places the file pointer in relation to the start of the file.
- b The call `seek(1)` will position the file pointer at the start of the file.
- c The call `seek(length())` will position the file pointer at the end of the file.
- d The call `seek(length()-1)` will position the file pointer at the last byte in the file.
- e Between each call to the `seek()` method the file must be closed and opened again.

(a), (c), (d)

The call `seek(0)` will position the file pointer at the start of the file. It is not necessary to close and open the file between calls to the `seek()` method.

17. Which statements about the class `RandomAccessFile` are true?

- a Records in such a file must have the same length.
- b Before reading a record, any records proceeding this record must be read first.
- c The class has methods for reading binary representation of primitive data values.
- d It is possible to open a file for both read- and write operations with this class.

(c), (d)

The records need not be of the same length, but often they are. The whole point of random access is lost if the records can only be read sequentially.

19.7 Programming exercises

1. Write a new version of Program 16.6 so that values for the two dimensional array `cityGraph` are read from a file. The file is specified on the command line. The file provides information about connection between cities. Each line in the file consists of two integers, n and m . Each integer represents a city number. The city with number m can be reached from the city with number n . The numbers are separated by a delimiter character. See also Program 11.2 for tips on reading from text files.
2. Write a version of the program from Exercise 19.1 that reads the values for the two dimensional array `cityGraph` in Program 16.6 from a file. How would you organise the values on the binary file?
3. Write a program that reads a sequence of integers from a binary file and prints a report on how many times each number occurs in the sequence. Assume that the sequence is terminated with a negative integer. Use a suitable map to store the frequency of each number.



- 4.** Write a program that reads a sequence of temperature values (`double`) from a binary file. The values represent temperature measurements for a month, one measurement per day. The values in the file are stored continuously, with the temperature for the first day of the month given first. Since the number of days in a month can vary, the number of values stored in the file is not fixed.

The program calculates the mean value, x , and the range. The range is defined as the difference between the highest and the lowest temperature registered in the month.

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$$

- 5.** Modify the program from Exercise 19.4 so that it also calculates the variance, $Var(x)$, and the standard deviation, S .

$$Var(x) = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2, S = \sqrt{Var(x)}$$

The program also reports the number of days that had a temperature with ± 5 degrees deviation from the variance.

Calculation of the variance requires that all temperature values are stored in the program. A list (`ArrayList<E>`) can be used to store temperature values as `Double` objects.

- 6.** Write a new version of the program from Exercise 13.3 that reads data about different types of employees from a file. The program uses object serialisation for storing information on a file.
- 7.** Write a new version of the program from Exercise 12.7, so that information about students, lecturers and courses is read from a file. The program uses object serialisation for storing information on a file.
- 8.** Write a new version of the program from Exercise 12.3, so that information about videos and DVDs is read from a file. The program uses object serialisation for storing information on a file.
- 9.** Write a new version of the program from Exercise 8.3, so that the product catalogue is stored on a file. The program uses object serialisation for storing information on a file.
- 10.** Write a new version of Program 6.4 on page 140, so that values for the two-dimensional array `weeklyData` are read from a file. Choose a suitable solution for storing array values on a file.
- 11.** Write a new version of Exercise 3.7 on page 69 so that the student IDs and the corresponding points are read from a file. The program prints a grade list in the terminal window. Choose a suitable solution for storing values on a file.



- 12.** In Exercise 11.18, the museum program was designed to store the items on a text file.

Write two new versions of the program, depending on how information is stored:

- on a binary file.
- on a file using object serialisation.

Design the `Museum` class so that it can choose which version to use for storing the museum items.

Graphical User Interfaces

LEARNING OBJECTIVES

By the end of this chapter, you will understand the following:

- What components and containers Java provides for developing a GUI.
- How layout managers can be used to arrange components in a GUI container.
- How to construct the component hierarchy of a GUI.
- How to use event-driven programming to manage GUI-based interaction between the user and the program.
- How to use anonymous classes to implement event listeners.
- How to systematically design and implement a GUI, based on the event delegation model.

INTRODUCTION

In this chapter we look at how to develop applications where the dialogue between the user and the program takes place via a *graphical user interface* (often called a *GUI*). The Java standard library provides two extensive packages (`java.awt`, `javax.swing`) that make developing such applications easy.

As an introduction to creating GUI-based dialogs, Section 11.3 showed how the class `javax.swing.JOptionPane` can be used to design simple GUI dialogue boxes. Section E.2 presents a class, `GUIDialog`, that hides the use of the class `javax.swing.JOptionPane` to read numerical values and strings typed in GUI dialogue boxes. In this chapter, we go further and explore how to develop more complex GUIs.

(Please fix this: the Bold element is used incorrectly in this chapter for specific keys (e.g. Enter) and button names (e.g. OK, Cancel). A new element must be defined, for example KeyCap/SmallCaps, for this purpose.)



20.1 Building GUIs

A graphical user interface provides a user-friendly way to conduct dialogue between the user and the program. The user interacts with the GUI by clicking on buttons, by making selections in menus, by clicking on check boxes, and by opening and closing windows, etc. Actions that the user performs in the GUI, result in corresponding operations in the program being executed. Clicking on a button marked `Save` causes, for example, information to be written to a file, and selecting the menu command `Print` causes the printout to be sent to the printer.

Designing GUI-based dialogue

There are two important aspects to consider regarding GUI-based dialogue:

- Actions performed by the user in the GUI, can occur in arbitrary sequence.
- The program must be informed about user actions in the GUI, in order to execute operations that correspond to these actions.

The first point requires that we must think carefully about how the program will be used. Unlike earlier programs, we can no longer assume that user actions occur in any specific order. The program must be able to handle many different action sequences. This puts greater demands on the program design, but it also gives greater flexibility to the user. User-driven programs are in a (conceptually) higher league than sequential programs. Structuring the program so that it behaves correctly in all cases is just as important as binding the right operation to a button being pressed in the GUI.

The second point above implies that we must understand the relationship between the GUI and the program. The GUI consists of graphical components drawn on the screen. A graphical representation on the screen is usually associated with a corresponding object in the program. When the user clicks on a graphical representation of a button marked `Save`, for example, the associated object in the program is informed. This happens without the program having to do anything. The program has the responsibility for what happens *after* the button object has been informed. The connection between the button object and the operation in the program for saving the information must be programmed explicitly.

Delegating events

The scenario described in the previous subsection can be seen as consisting of two steps: the graphical object is informed when the user does something in the GUI – this happens automatically – and after that the response to the user action takes place in the program – this must be implemented. Java also provides support for the second step, called the *event delegation model*.

An *event* is an object that contains information about how the GUI is manipulated by the user. An event can give information to the program about what action the user has taken in the GUI (pressed the mouse button, moved the mouse pointer, pressed a key, closed a window, moved a window, scrolled down, made a menu selection, and so on), or how the GUI has changed (window has been closed, opened, uncovered or hidden, and so on).



In our example with the button object, this object will create an event with relevant information indicating that the graphical button marked "Save" was clicked on. The button object is called the *source* of the event. A source object can send the event to other objects by calling a method in them, passing the event as an argument. Objects that receive events in this way, are called *listeners* (aka. as *event handlers*) and are responsible for the appropriate handling of events. In the event delegation model, sources *delegate* the events to listeners for handling, and the events are handled in the order they occur. We say that sources *generate* events, and that GUI-based applications are *event-driven*. Java provides ready-made components that act as sources for different types of events.

GUI building packages

Java provides an extensive class library, *JFC (Java Foundation Classes)*, that contains the *AWT (Abstract Window Toolkit)* and *Swing* packages for developing GUI-based applications. Swing is in the `javax.swing` package. It builds on the AWT, that is found in the `java.awt` package, thus making it possible to use the services from the AWT.

The graphical packages in Java contain classes for many standard graphical components (windows, buttons, menus, and so on) that can be used in GUI development. The packages provide *containers*, that are special components customised for grouping other components. In addition, there are *layout managers* that can be associated with containers for placing components on the screen according to a specific layout policy.

Basics of GUI development

Development of GUI-based applications requires understanding of the following topics:

- *the inheritance hierarchy* of the components and the containers, that specifies the behaviour and properties of the components in the graphical user interface. We will give an overview of important classes in AWT and Swing for this purpose.
- the structure of *the component hierarchy*, that specifies how components are put together to build the graphical user interface. For this purpose we will use containers and layout managers.
- how the event delegation model makes use of sources and listeners for handling events that occur during the interaction with the user. We give an overview of some selected sources and the events they generate, and we show how we can register listeners with sources in order to receive events.

Subsequent sections will elaborate on these topics one by one.

20.2 Components and containers

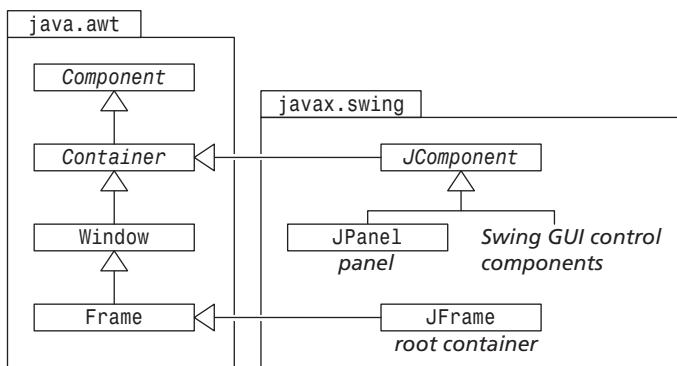
Before embarking on GUI development, it is useful to get an overview of the different types of graphical components that are available. Classes in AWT and Swing make extensive use of inheritance for implementing graphical components. Figure 20.1 shows a partial *inheritance hierarchy* of classes that form the basis for GUI development in Java. Graphical user interfaces will typically use properties and behaviour of the following specialised components:



- *root container* that is the starting point for the component hierarchy that comprises the whole graphical user interface
- *panels* used to group components and thereby give structure to the component hierarchy
- *control components* that the user conducts dialogue with, and that generate events

FIGURE 20.1

Partial inheritance hierarchy of containers and components



Components

Figure 20.1 shows how Swing components are dependent on components in AWT. The abstract class `Component` defines common properties and behaviour of all components in AWT and Swing. The class `Component` provides support for, among other things, changing the component size, control of fonts and colours, and drawing of components and their contents. Classes in Swing inherit from this or other AWT classes, and provide more advanced GUI functionality. The names of the components in the Swing package usually start with a 'J'.

Table 20.1 shows some selected methods from the `Component` class that are sufficient to get going with GUI building. A component has a background colour and a foreground colour that can be set. A button with *foreground colour* black and *background colour* blue results in the text in the button being shown with black colour on blue background. (See the overview of predefined colours in Table 20.3.) The method call `setVisible(true)` is used when the whole GUI has been built and we are ready to show it on the screen.

BEST PRACTICES

Use components from the Swing package as much as possible to build your GUI, as the Swing package provides a more comprehensive set of GUI components than the AWT.


TABLE 20.1 Selected methods common for all components

Method	Description
<code>void setForeground(Color color)</code>	Sets the component's foreground colour to the value specified by the parameter <code>color</code> .
<code>void setBackground(Color color)</code>	Sets the component's background colour to the value specified by the parameter <code>color</code> .
<code>void setVisible(boolean b)</code>	The component is made visible if the parameter <code>b</code> has the value <code>true</code> , otherwise it is made invisible.

Labels

To illustrate designing GUIs, we first introduce a simple, but useful component provided by the class `JLabel`. This class inherits from the class `JComponent` and subsequently from the `Component` class, thereby inheriting the common properties described above. A `JLabel` object can contain a label text and/or an image. Labels are typically used to explain the purpose of the components in the GUI to the user. A `JLabel` object does not generate any events, and its contents cannot be changed by the user.

Table 20.2 shows a constructor for creating a `JLabel` object with a label text, and methods for setting and getting this text. Similarly, there are methods in the class `JLabel` for handling an image (an `Icon` object) in a label. The following code creates two labels:

```
JLabel headlines = new JLabel("The end of the world is near!");
JLabel message = new JLabel("Don't forget to bring your credit card.");
```

TABLE 20.2 Selected constructors and methods for labels

<code>javax.swing.JLabel</code>	
<code>JLabel(String text)</code>	Creates a label with the text specified by the parameter <code>text</code> .
<code>String getText()</code>	Returns the text in the label.
<code>void setText(String text)</code>	Sets the string specified by the parameter <code>text</code> as the text in the label. Any previous label text is overwritten.

Colours

All components have a foreground- and a background colour. If we do not set the colour in a component, it will have the colours of the container in which it is placed. The class `Color` is used to represent colours. This class also has a number of predefined colours that we can use (shown in Table 20.3).

```
headlines.setBackground(Color.blue); // blue background
headlines.setForeground(Color.black); // black text on blue background
```


TABLE 20.3 Predefined colours

java.awt.Color		
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		

Containers

All objects of the abstract class `Container` and its subclasses are *containers* (Figure 20.1). A container has the property that it can contain other components. Since a container is also a component (because the class `Container` is a subclass of the class `Component`), a container can contain other containers. This nesting of containers and other components defines a *component hierarchy* (not to be confused with the inheritance hierarchy). Such a hierarchy has a *root container*, that is usually an object of the class `JFrame`, while nested containers are often panels of the class `JPanel`. As we will soon see, root containers and panels are specialised containers implemented by the classes `JFrame` and `JPanel`.

Arranging components using layout managers

How should the components in a container be placed and shown in a window on the screen? What will be the size of a component in relation to other components and to the container in which it is placed? Spatial relationships between components define placement and size of the components in a window on the screen. In Java, the simplest way to handle such spatial relationships is to use a *layout manager* that can be associated with a container. The layout manager takes care of the visual placement of components in the container, and updates the container and its contents on the screen whenever necessary. For example, when the user changes the window size or the window becomes uncovered. Section 20.4 discusses different layout managers in Java.

Table 20.4 shows the method `setLayout()` from the class `Container`, that can be used to associate a layout manager with a container. The layout manager `FlowLayout` will, for example, show the components row-wise from left to right on the screen, in the order they were added to the container. The position of the components in a container is determined by the order in which they are added to the container. It is important to note that the placement of a component on the screen is determined by its position in the container. Table 20.4 shows the overloaded method `add()`, that adds a new component after the last component in the container, i.e. the end of the container. There are also methods for removing components from a container. In the code below the `headlines` label will be placed before the `message` label in the container:

```
container.setLayout(new FlowLayout()); // Sets a layout manager.
container.add(headlines);           // headlines in 1st position.
container.add(message);            // message in 2nd position.
```


TABLE 20.4 Selected methods common to all containers

java.awt.Container	
<code>void setLayout(LayoutManager mngr)</code>	Sets the layout manager specified by the parameter <code>mngr</code> as the layout manager for the container.
<code>Component add(Component comp)</code>	Adds the component specified by the parameter <code>comp</code> to the end of the container.
<code>void add(Component comp, Object info)</code>	Adds the component specified by the parameter <code>comp</code> to the end of the container. The parameter <code>info</code> can provide additional information about the placement of the component in the container.

Windows and frames

The class `Window` creates a *top-level* container that is a window without any title, menus or *borders*. Such a window occupies an area on the screen, but is invisible. A *top-level window* cannot be placed inside a container.

The root container and its content pane

The class `JFrame` is used to create what is usually meant by a *window* on the screen. Such a *frame* is a top-level window that can have a title, menus, borders, a *cursor* and an icon. A `JFrame` object is called a *root container*, and constitutes the starting point for a window-based application. A root container cannot be incorporated into other containers.

A selection of methods from the class `JFrame` is shown in Table 20.5. Unlike a regular container, the components are not added directly to a root container, but to its *content pane*, which is a regular container. This content pane constitutes the root of a component hierarchy (see Figure 20.2a). The method `getContentPane()` retrieves the content pane of a frame. The method `add()`, inherited from the `Container` class, is used to add components to the content pane of the root container.

A content pane can, just like other containers, use a layout manager to manage its components. We can use the inherited `setLayout()` method from the `Container` class to associate a layout manager with the content pane. If we do not explicitly set a layout manager for the content pane, the content pane will use the `BorderLayout` manager (Section 20.4).

TABLE 20.5 Selected methods for the root container `JFrame`

javax.swing.JFrame	
<code>Container getContentPane()</code>	Returns the content pane of the frame.



javax.swing.JFrame

<code>void setDefaultCloseOperation(int operation)</code>	Sets the operation that should be executed when the frame is closed, and after all <code>WindowListener</code> objects (Section 20.5) have been notified. The parameter <code>operation</code> can have one of the values defined by the <code>JFrame</code> class: <code>DO NOTHING ON CLOSE</code> (the program must undertake proper closing and disposing of the frame itself), <code>HIDE ON CLOSE</code> (the default operation that hides the frame, but does not dispose of it), <code>DISPOSE ON CLOSE</code> (ensures proper closing and disposing of the frame), <code>EXIT ON CLOSE</code> (specifies that the <i>application must be terminated</i> when the frame is closed).
<code>void pack()</code>	Adjusts the size of the frame, so that the components can retain their preferred size and layout in the frame.
<code>void dispose()</code>	Releases all screen resources used by the frame. The method ensures proper closing and disposing of the frame.

Panels

A panel is an object of the class `JPanel` that is a subclass of the `JComponent` class. A panel defines a window that does not have any title, menus or borders, but that may contain other components. Such a panel is a container (and also a component because of inheritance). This means that a panel can contain other panels. This property makes a panel an ideal candidate for grouping components, and is frequently used in GUI design. Components can be added to a `JPanel` object using the inherited `add()` method. A panel uses the `FlowLayout` manager as its default layout manager (Section 20.4).

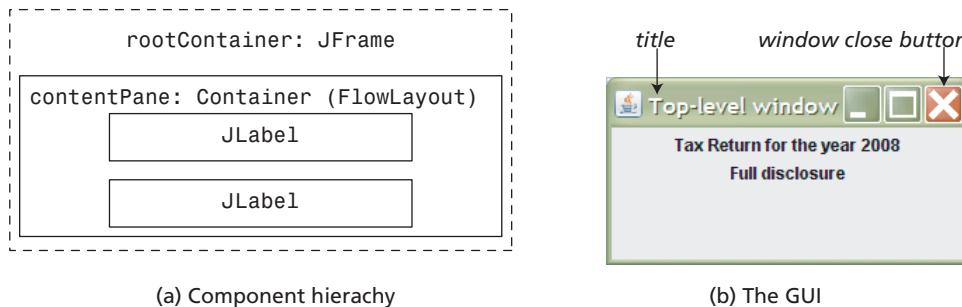
See Program 20.2 and Program 20.3 for examples with GUIs that use panels.

Steps in developing GUI

Program 20.1 illustrates the general steps in designing a typical GUI. We want to create the GUI shown in Figure 20.2b. Figure 20.2a shows the component hierarchy for this GUI. Two `JLabel` objects are used for the two text lines. We add them to the content pane of the root container, which is then shown on the screen.



FIGURE 20.2 GUI top-level window



The explanation below refers to lines with the same number in Program 20.1.

- 1 Avoid using fully qualified names for classes, for example `Container` instead of `java.awt.Container`, an `import` statement must be included in the source code to refer to the classes in the AWT package.

```
import java.awt.Container; // (1)
```

- 2 We will use the classes from the Swing package, and an `import` statement makes it easier to refer to them.

```
import javax.swing.JFrame; // (2)
```

- 3 One common way to create a top-level window is to extend the `JFrame` class. This allows the behaviour from the `JFrame` class to be overridden if necessary. The subclass has the responsibility of constructing the complete component hierarchy. In Program 20.1, the class `GUIFrame` is the subclass of the class `JFrame` and has all properties and behaviour of a frame.

```
class GUIFrame extends JFrame { // (3)
```

- 4 Components that comprise the component hierarchy are usually declared as private field variables. We need two labels for our purpose. Note that no labels are created, only references are declared.

```
private JLabel label1;
private JLabel label2;
```

- 5 The GUI is created in a constructor of the frame. When the frame is created, the GUI is also created.

```
GUIFrame() { // (5)
```

- 6 The constructor in the subclass can set a suitable title in the frame with the `super()` call.

```
super("Top-level window");
```

- 7 All the components needed to build the component hierarchy, are created. Labels are created, and their foreground colour is set to black, so that the text is drawn in this colour.

```
label1 = new JLabel("Tax Return for the year 2008");
label1.setForeground(Color.black);
label2 = new JLabel("Full disclosure");
label2.setForeground(Color.black);
```



- 8** The root of the component hierarchy is the content pane of the frame. We fetch it with the `getContentPane()` method from the `JFrame` class (Table 20.5).

```
Container contentPane = getContentPane();
```

- 9** We will associate the `FlowLayout` manager with the content pane. A layout manager is associated with the content pane by calling the `setLayout()` method from the `Container` class (Table 20.4).

```
contentPane.setLayout(new FlowLayout());
```

- 10** We construct the component hierarchy by adding the labels to the content pane.

```
contentPane.add(label1);
contentPane.add(label2);
```

- 11** An application usually terminates when the user clicks on the *window close button* in the frame (Figure 20.2b). We can achieve this by calling the `setDefaultCloseOperation()` method with the parameter `JFrame.EXIT_ON_CLOSE` (Table 20.5).

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

- 12** The GUI is now constructed, but still invisible. We *pack* the component hierarchy first by calling the method `pack()`. This method adapts the size of the components in the frame. The complete component hierarchy is then made visible by calling the method `setVisible()` with the argument value `true`.

```
pack();
setVisible(true);
```

Program 20.1 also shows a client for the class `GUIFrame`. The `main()` method in the `GUIFrameClient` class creates an object of the class `GUIFrame` that results in the GUI being created and shown on the screen. If the GUI is not exactly as in Figure 20.2b, the window size can be adjusted on the screen to match the GUI in Figure 20.2b. There is not much the user can do with this GUI, but it is possible to perform the usual window operations on the GUI (for example, moving, hiding and uncovering of the window). If the user clicks on the window close button, the GUI will disappear, and the application will terminate.

From Program 20.1, it might seem that there is a lot of programming involved in setting up the GUI, but the setup can be copied and customised for other GUI-based applications.

PROGRAM 20.1 Steps in GUI development

```
import java.awt.Container;           // (1) Classes from the awt package
import java.awt.FlowLayout;
import java.awt.Color;

import javax.swing.JFrame;           // (2) Classes from the swing package
import javax.swing.JLabel;

class GUIFrame extends JFrame { // (3)

    // (4) Components
    private JLabel label1;
```



```

private JLabel label2;

GUIFrame() { // (5)
    // (6) Set suitable title for the frame.
    super("Top-level window");

    // (7) Create components.
    label1 = new JLabel("Tax Return for the year 2008");
    label1.setForeground(Color.black);
    label2 = new JLabel("Full disclosure");
    label2.setForeground(Color.black);

    // (8) Get the content pane for the frame.
    Container contentPane = getContentPane();

    // (9) Set layout manager for the content pane.
    contentPane.setLayout(new FlowLayout());

    // (10) Construct the component hierarchy.
    contentPane.add(label1);
    contentPane.add(label2);

    // (11) Terminate if the frame is closed.
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // (12) Show the GUI.
    pack();
    setVisible(true);
}

}

public class GUIFrameClient {
    public static void main(String[] args) {
        GUIFrame gui = new GUIFrame();
    }
}

```

GUI and the `main()` method

In all earlier examples, the program terminates when the `main()` method has finished executing. This is not the case with Program 20.1. The program does not terminate after the creation of the `GUIFrame` object in the `main()` method of the class `GUIClient`. What happens after the GUI has been created, is that the runtime environment automatically takes over the monitoring of the interaction between the user and the GUI. This GUI monitor ensures that user actions in the GUI are reported to the program. The `main()` method can complete execution as in Program 20.1, or continue with the execution of other actions, but that does not affect the user's interaction with the GUI.



BEST PRACTICES

Always provide the user a way to exit your GUI application properly.

20.3 GUI control components

Table 20.6 shows a very small selection of *control components* in the Swing package. User interaction takes place mainly via such components. The control components generate events during dialogue with the user, and must be linked to suitable actions in the program (Section 20.5). All control components are subclasses of the `JComponent` class. This means that they are also containers, but such control components are seldom used as containers. It is quite common to group control components in panels, that in turn can be nested within other panels in order to build a component hierarchy. The component hierarchy is placed in the content pane of a top-level window, as shown in Program 20.1.

The use of control components makes exchange of information between the user and the program possible. For example, the program can get input from a text field that the user can write in, while information can be shown to the user in another text field (without the possibility of editing).

TABLE 20.6 Selected GUI control components

GUI control component	Description
<code>JTextField</code>	A component that implements one line of text, called a <i>text field</i> . The text can be either editable or not.
<code>JButton</code>	A <i>push button</i> with a text and/or drawing, that can be clicked on with a mouse.
<code>JCheckBox</code>	A <i>checkbox</i> that can be <i>checked</i> or <i>unchecked</i> .
<code>JRadioButton</code>	A <i>radio button</i> that can be <i>selected</i> . A group of such buttons can be organised in such a way that only one radio button is selected at any given time.

Text fields

Table 20.7 shows a selection of constructors and methods from the class `JTextField`. We do not have to program the usual editing operations (cut, copy, paste, etc.) on characters in such a field. The class `JTextField` provides for that. For an example of a GUI with text fields, see Program 20.2. The rest of this chapter contains several examples that utilise text fields.


TABLE 20.7 Selected constructors and methods for text fields

<code>javafx.swing.JTextField</code>	
<code>JTextField(String text)</code>	Creates a text field that contains the text specified by the <code>String</code> parameter <code>text</code> .
<code>JTextField(int numColumns)</code>	Creates an empty text field. The parameter <code>numOfColumns</code> is used to calculate the width of the field. Note that the number of characters that can be written in the field, can be greater than the value of this parameter.
<code>JTextField(String text, int numColumns)</code>	Creates a text field that contains the text specified by the parameter <code>text</code> . The parameter <code>numOfColumns</code> specifies the preferred size of the field.
<code>String getText()</code>	Returns the text in the text field.
<code>void setText(String text)</code>	Sets the text specified by the parameter <code>text</code> in the text field by overwriting the previous contents of the text field.
<code>void setEditable(boolean b)</code>	The text field can be edited if the parameter <code>b</code> is <code>true</code> , and cannot be edited if the parameter <code>b</code> is <code>false</code> .
<code>boolean isEditable()</code>	Determines whether the text field can be edited.
<code>void addActionListener(ActionListener listener)</code>	Registers the <code>listener</code> who is interested in <code>ActionEvents</code> from the text field. An <code>ActionEvent</code> is generated when the user hits the Enter key.

Buttons

A push button (`JButton`), a checkbox (`JCheckBox`) and a radio button (`JRadioButton`) inherit common behaviour from the `javafx.swing.AbstractButton` class. Figure 20.5 shows the graphical representation of these buttons.

Table 20.8 shows a selection of constructors for buttons. A button can have text as a label, and can be in one of two states: *selected* or *unselected*, and the appearance of the button will indicate which state it is in. A button can be *enabled* or *disabled*. An enabled button will react to user actions, while a disabled button will not, i.e. only an enabled button can be selected. A button is enabled when it is created.

Table 20.9 shows a selection of methods for buttons. There are methods for handling the button label and the state of a button, and for making a button enabled or disabled.

The way radio buttons are usually used is a little bit different from other buttons. When radio buttons are placed in a `ButtonGroup` object, the `ButtonGroup` object will ensure that the user can only select *one* button in this group of radio buttons. The buttons can be added by calling the `add()` method on the `ButtonGroup` object. A `ButtonGroup` object must *not* be placed in the component hierarchy, only the radio buttons. A `ButtonGroup` object is



a *logical* component for the handling of buttons, and not a GUI component that can be shown on the screen.

Program 20.3 shows an example that creates a GUI with different types of buttons. In Section 20.5 we will show how to link actions to buttons in the program.

TABLE 20.8 Selected constructors for buttons

<code>javax.swing.JButton, javax.swing.JCheckBox, javax.swing.JRadioButton</code>	
<code>JButton(String text)</code>	Creates a button with the label specified by the parameter <code>text</code> . The button is active, but not selected.
<code>JCheckBox(String text)</code>	
<code>JRadioButton(String text)</code>	
<code>JCheckBox(String text, boolean selected)</code>	Creates a button with the label specified by the parameter <code>text</code> . It is selected if the parameter <code>selected</code> is <code>true</code> .
<code>JRadioButton(String text, boolean selected)</code>	

TABLE 20.9 Selected methods for buttons

<code>javax.swing.JButton, javax.swing.JCheckBox, javax.swing.JRadioButton</code>	
<code>String getText()</code>	Returns the text in the label of the button.
<code>void setText(String text)</code>	Sets the text specified by the parameter <code>text</code> as the label of the button.
<code>String setSelected(boolean selected)</code>	Sets the state of the button to be selected or unselected, depending on whether the parameter <code>selected</code> is <code>true</code> or <code>false</code> .
<code>boolean isSelected()</code>	Returns whether the button is selected or not.
<code>boolean setEnabled(boolean active)</code>	Sets the button to active or inactive, depending on whether the parameter <code>active</code> is <code>true</code> or <code>false</code> .
<code>void addActionListener(ActionListener listener)</code>	Registers the <code>listener</code> who is interested in <code>ActionEvents</code> from the button. An <code>ActionEvent</code> is generated when the user clicks on the button, depending on whether the button is selected or not (see Section 20.5).

GUI design with panels

Program 20.2 illustrates the use of panels and control components, and implements the GUI shown in Figure 20.4. The program allows the user to write an arbitrary text in an input field, whose contents are then repeated in an echo field when the user hits the Enter key in the input field. This example only shows the design of the GUI, while associating program actions to user actions is postponed until Section 20.5, which discusses event



handling.

It is natural to group the label and the text field for the input field in a panel, as for the echo field. Figure 20.4 shows the containers, the panels and the control components that form the component hierarchy. For containers, we have explicitly shown the layout managers. The steps in building the GUI in Program 20.2 are similar to the steps in Program 20.1. For panels, we have set different background colours, so that it is possible to see the area they occupy in the window. The following code constructs the component hierarchy:

```
panel1.add(inputLabel);
panel1.add(inputField);
panel2.add(echoLabel);
panel2.add(echoField);
contentPane.add(panel1);
contentPane.add(panel2);
contentPane.add(doneButton);
```

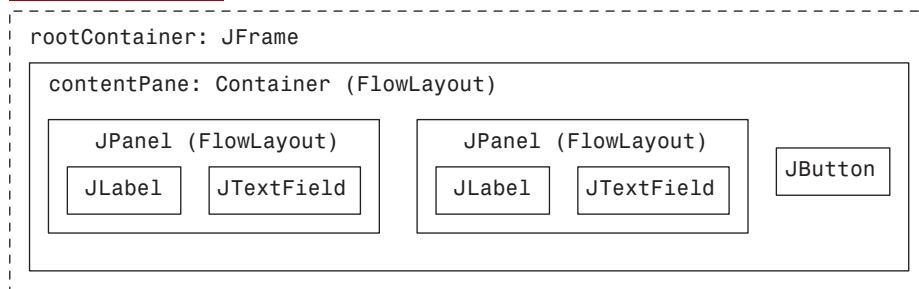
Note the order in which components are inserted into the containers to achieve the GUI shown in Figure 20.3.

We can write and edit the text in the input field in the GUI. Arrow keys can be used in the input field to move the cursor. We cannot write in the echo field, since it is non-editable. We can click on the «Done» button and see that it is *depressed*. In Section 20.5 we will implement program actions that correspond to events generated during dialogue with the user.

FIGURE 20.3 GUI design with panels (Program 20.2)



FIGURE 20.4 Component hierarchy for GUI design with panels (Figure 20.3)





PROGRAM 20.2 GUI design with panels

```
import java.awt.Color;
import java.awt.Container;
import java.awt.FlowLayout;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

class EchoWithGUIOnly extends JFrame {

    // Components
    private JPanel panel1, panel2;
    private JButton doneButton;
    private JLabel inputLabel;
    private JLabel echoLabel;
    private JTextField inputField;
    private JTextField echoField;

    EchoWithGUIOnly() {
        // Set suitable title for the frame.
        super("Echo Input");

        // Create GUI components.
        panel1 = new JPanel();
        panel1.setBackground(Color.yellow);
        panel2 = new JPanel();
        panel2.setBackground(Color.pink);

        doneButton = new JButton("Done");
        doneButton.setBackground(Color.cyan);

        inputLabel = new JLabel("Input field:");
        echoLabel = new JLabel("Echo field:");

        inputField = new JTextField("Type here." + "End with Enter key.", 10);
        inputField.setEditable(true);

        echoField = new JTextField("", 10);
        echoField.setEditable(false);
        echoField.setBackground(Color.lightGray);

        // Get the content pane for the frame.
        Container contentPane = getContentPane();
        contentPane.setBackground(Color.white);
```



```

// Set layout manager for the content pane.
contentPane.setLayout(new FlowLayout());

// Construct the component hierarchy.
panel1.add(inputLabel);
panel1.add(inputField);
panel2.add(echoLabel);
panel2.add(echoField);
contentPane.add(panel1);
contentPane.add(panel2);
contentPane.add(doneButton);

// Terminate if the frame is closed.
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Show the GUI.
pack();
setVisible(true);
}

}

public class EchoWithGUIOnlyClient {
    public static void main(String[] args) {
        EchoWithGUIOnly gui = new EchoWithGUIOnly();
    }
}

```

Using different types of buttons

Program 20.3 illustrates the use of different types of buttons, and implements the GUI shown in Figure 20.5. The program allows the user to choose one pizza size, but zero or more pizza toppings. Again we only show the design of the GUI, while associating program actions to user interaction is postponed until Section 20.5.

Figure 20.6 shows container, panels and different types of buttons that comprise the component hierarchy. It is natural to group pizza sizes in a panel, and similarly for pizza toppings. Each pizza size is indicated by a radio button, while each pizza topping is indicated by a checkbox.

We organise the radio buttons for pizza sizes using a `ButtonGroup` object, so that only one pizza size can selected:

```

ButtonGroup group = new ButtonGroup();
group.add(smallPizza);
group.add(mediumPizza);
group.add(largePizza);

```

Radio buttons for the pizza sizes must be added to the component hierarchy, but not the `ButtonGroup` object:

```
panel1 = new JPanel();
```



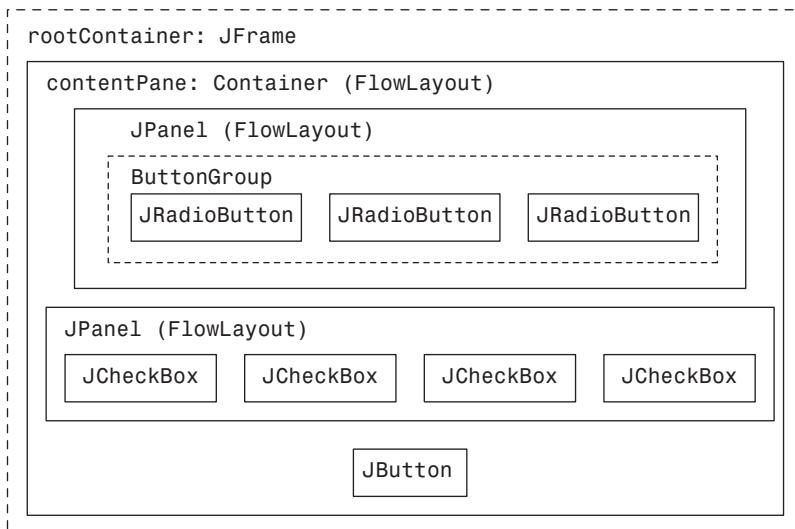
```
panel1.add(smallPizza);
panel1.add(mediumPizza);
panel1.add(largePizza);
```

It is possible that the GUI window must be adjusted for the appearance to be as shown in Figure 20.5. The buttons in the GUI behave as they should; they can be selected and unselected.

FIGURE 20.5 GUI for ordering pizza (Program 20.3)



FIGURE 20.6 Component hierarchy of GUI for ordering pizza (Figure 20.5)



PROGRAM 20.3 Ordering pizza

```
import java.awt.Container;
import java.awt.FlowLayout;

import javax.swing.ButtonGroup;
import javax.swing.JButton;
import javax.swing.JCheckBox;
```



```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JRadioButton;

class PizzaWithGUIOnly extends JFrame {

    // Components
    private JPanel panel1, panel2;

    // Radiobuttons for pizza sizes
    JRadioButton smallPizza;
    JRadioButton mediumPizza;
    JRadioButton largePizza;

    // Checkboxes for pizza toppings
    JCheckBox pizzaTopping1;
    JCheckBox pizzaTopping2;
    JCheckBox pizzaTopping3;
    JCheckBox pizzaTopping4;

    // Button to order a pizza
    private JButton orderButton;

    PizzaWithGUIOnly() {
        // Set suitable title for the frame.
        super("Pizza Shack");

        // Create radiobuttons for the different pizza sizes.
        smallPizza = new JRadioButton("Small pizza");
        mediumPizza = new JRadioButton("Medium pizza");
        largePizza = new JRadioButton("Large pizza");
        largePizza.setSelected(true);

        // Create a ButtonGroup object for grouping the radiobuttons.
        ButtonGroup group = new ButtonGroup();
        group.add(smallPizza);
        group.add(mediumPizza);
        group.add(largePizza);

        // Create a panel for pizza sizes
        panel1 = new JPanel();
        panel1.add(smallPizza);
        panel1.add(mediumPizza);
        panel1.add(largePizza);

        // Create checkboxes for pizza toppings.
        pizzaTopping1 = new JCheckBox("Chicken");
        pizzaTopping2 = new JCheckBox("Pepperoni");
        pizzaTopping3 = new JCheckBox("Meat balls");
        pizzaTopping4 = new JCheckBox("Beef");
    }
}
```



```
    pizzaTopping1.setSelected(true);

    // Create a panel for toppings
    panel2 = new JPanel();
    panel2.add(pizzaTopping1);
    panel2.add(pizzaTopping2);
    panel2.add(pizzaTopping3);
    panel2.add(pizzaTopping4);

    // Create a button for ordering pizza.
    orderButton = new JButton("Order");

    // Get the content pane for the frame.
    Container contentPane = getContentPane();

    // Set layout manager for the content pane.
    contentPane.setLayout(new FlowLayout());

    // Construct the component hierarchy.
    contentPane.add(panel1);
    contentPane.add(panel2);
    contentPane.add(orderButton);

    // Terminate if the frame is closed.
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Show the GUI.
    pack();
    setVisible(true);
}

}

public class Pizzeria {
    public static void main(String[] args) {
        PizzaWithGUIOnly pizzaOrder = new PizzaWithGUIOnly();
    }
}
```

20.4 Designing layout

A container uses a layout manager to position components in the container. A layout manager is associated with a container by calling the `setLayout()` method.

The component hierarchy is built by adding components to containers with the `add()` method. Each component can specify a *preferred size* that should be used for drawing the component on the screen. Whether this request is fulfilled depends on the layout manager



used. As we shall see, some layout managers will *stretch* the components when spatial relations between the components change, depending on the size of the container.

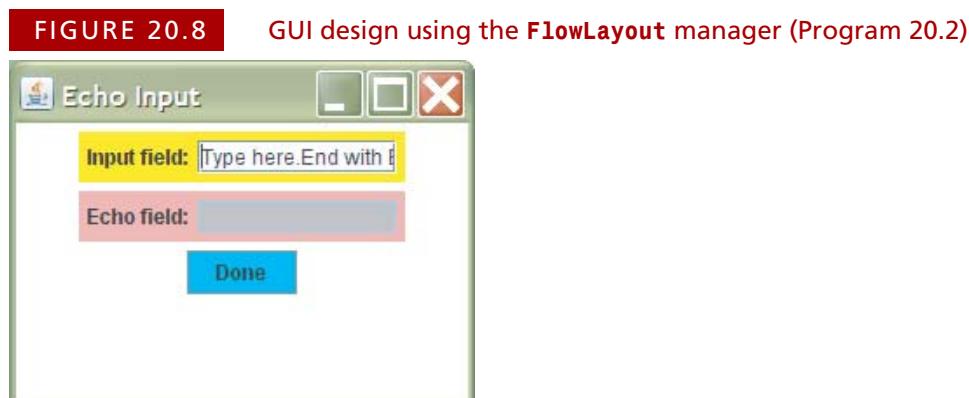
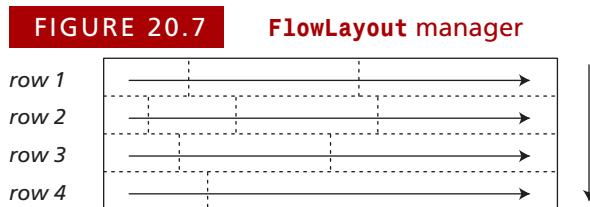
The rest of this section discusses the three most commonly used layout managers provided by the AWT package:

- `FlowLayout`
- `BorderLayout`
- `GridLayout`

FlowLayout

A `FlowLayout` manager positions components row-wise from left to right, and from top to bottom in the container (Figure 20.7). It creates new rows depending on the number of components in the container and the width of the container. This means that the number of components in each row can change (and thereby the number of rows) if the container width is changed (compare Figure 20.3 and Figure 20.8). This layout manager is the default layout manager for panels, meaning that components in a panel are arranged with a `FlowLayout` manager if a layout manager is not specified (Program 20.2).

It is also important to note that a `FlowLayout` manager does *not* stretch the components when the container size changes (Program 20.2).



BorderLayout

A `BorderLayout` manager arranges components in regions specified by the four *compass* directions (NORTH, SOUTH, EAST, WEST) and the centre (CENTER) of the container. These



regions are defined by the constants shown in Figure 20.9. This means that when inserting a component in a container, a region must be specified, or the component is placed in the centre region. It is always the component that was last inserted in a region, that is shown on the screen, and it is not necessary to insert a component in all regions.

A `BorderLayout` manager will *not* maintain the preferred size of the components, but spatial relations are maintained regardless of whether the size of the container is changed, and a component will stay in the region it was placed. We can use a panel (with `FlowLayout` manager) to avoid stretching, as illustrated in Program 20.6.

A `BorderLayout` manager is the default layout manager for all frames, including the content pane of a root container.

Program 20.4 uses the `BorderLayout` manager to implement the GUI shown in Figure 20.10. The component hierarchy is shown in Figure 20.11. We use the `add()` method with the additional parameter to specify the region the component should be placed in.

```
// Create and set a Borderlayout manager.
contentPane.setLayout(new BorderLayout());

// Construct the component hierarchy.
contentPane.add(drawButton, BorderLayout.NORTH);
contentPane.add(zoomInButton, BorderLayout.WEST);
contentPane.add(zoomOutButton, BorderLayout.EAST);
contentPane.add(banner, BorderLayout.CENTER);
contentPane.add(messageField, BorderLayout.SOUTH);
```

FIGURE 20.9 `BorderLayout` manager



FIGURE 20.10 GUI design using the `BorderLayout` manager (Program 20.4)

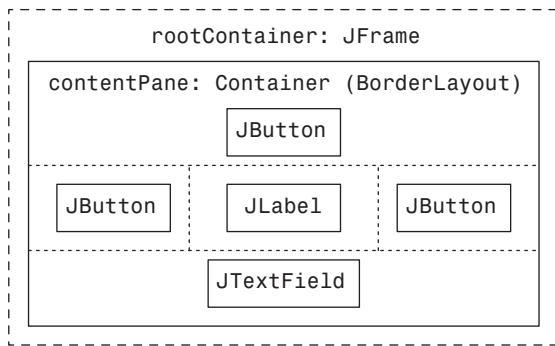


(a)

(b)



FIGURE 20.11 Component hierarchy using `BorderLayout` manager



PROGRAM 20.4 GUI design with `BorderLayout` manager

```

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Container;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.SwingConstants;

class BorderLayoutDemo extends JFrame {

    // Components
    private JButton drawButton;
    private JButton zoomInButton;
    private JButton zoomOutButton;
    private JLabel banner;
    private JTextField messageField;

    BorderLayoutDemo() {
        // Set suitable title for the frame.
        super("BorderLayoutDemo");

        // Create 3 buttons.
        drawButton = new JButton("Draw");
        zoomInButton = new JButton("Zoom in");
        zoomOutButton = new JButton("Zoom out");

        // Create a label.
        banner = new JLabel("Win a tour to Java!", SwingConstants.CENTER);
        banner.setForeground(Color.blue);

        // Create a text field.
    }
}
  
```



```

messageField = new JTextField("MESSAGES");
messageField.setEditable(false);
messageField.setBackground(Color.white);

// Get the content pane of the frame.
Container contentPane = getContentPane();

// Create and set a Borderlayout manager.
contentPane.setLayout(new BorderLayout());

// Construct the component hierarchy.
contentPane.add(drawButton, BorderLayout.NORTH);
contentPane.add(zoomInButton, BorderLayout.WEST);
contentPane.add(zoomOutButton, BorderLayout.EAST);
contentPane.add(banner, BorderLayout.CENTER);
contentPane.add(messageField, BorderLayout.SOUTH);

// Terminate if the frame is closed.
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Show the GUI.
pack();
setVisible(true);
}

}

public class BorderLayoutClient {
    public static void main(String[] args) {
        BorderLayoutDemo gui = new BorderLayoutDemo();
    }
}

```

GridLayout

A `GridLayout` manager divides the area in the container into a *two-dimensional grid* which consists of rows and columns (Figure 20.12). Components are arranged row-wise, where each component occupies a *cell* of equal size. The cell size is determined by the number of components that are to be placed in the container, and the size of the container.

Program 20.5 illustrates the use of a `GridLayout` manager. The GUI created is shown in Figure 20.13, and the corresponding component hierarchy is shown in Figure 20.14. The example uses a 2×2 grid to arrange two labels and two text fields in the content pane:

```

contentPane.setLayout(new GridLayout(2,2));
contentPane.add(xJLabel); // [0,0]
contentPane.add(xInput); // [0,1]
contentPane.add(yJLabel); // [1,0]
contentPane.add(yInput); // [1,1]

```



Note that the order in which the components are inserted determines their position in the grid. It is not possible to place a component directly in a specific cell.

FIGURE 20.12 **GridLayout manager**

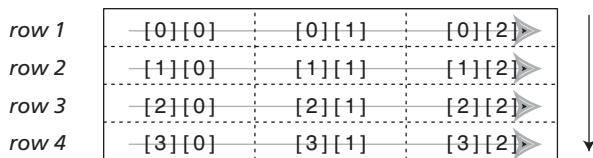


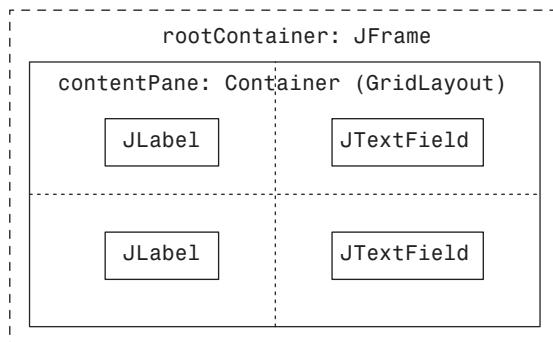
FIGURE 20.13 GUI design using **GridLayout manager** (Program 20.5)



(a)

(b)

FIGURE 20.14 Component hierarchy with **GridLayout manager**



PROGRAM 20.5 GUI design with **GridLayout manager**

```

import java.awt.Color;
import java.awt.Container;
import java.awt.GridLayout;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
  
```



```
class GridLayoutDemo extends JFrame {  
  
    // Components  
    private JLabel xJLabel;  
    private JLabel yJLabel;  
    private JTextField xInput;  
    private JTextField yInput;  
  
    GridLayoutDemo() {  
        // Set a suitable title for the frame.  
        super("GridLayoutDemo");  
  
        // Create two labels and two text fields.  
        xJLabel = new JLabel("X coordinate:");  
        yJLabel = new JLabel("Y coordinate:");  
        xInput = new JTextField(5);  
        yInput = new JTextField(5);  
        xInput.setBackground(Color.yellow);  
        yInput.setBackground(Color.yellow);  
  
        // Get the content pane for the frame.  
        Container contentPane = getContentPane();  
  
        // Create and set a GridLayout with a 2x2 grid.  
        contentPane.setLayout(new GridLayout(2, 2));  
  
        // Construct the component hierarchy.  
        contentPane.add(xJLabel); // [0,0]  
        contentPane.add(xInput); // [0,1]  
        contentPane.add(yJLabel); // [1,0]  
        contentPane.add(yInput); // [1,1]  
  
        // Terminate if the frame is closed.  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        // Show the GUI.  
        pack();  
        setVisible(true);  
    }  
}  
  
public class GridLayoutClient {  
    public static void main(String[] args) {  
        GridLayoutDemo gui = new GridLayoutDemo();  
    }  
}
```



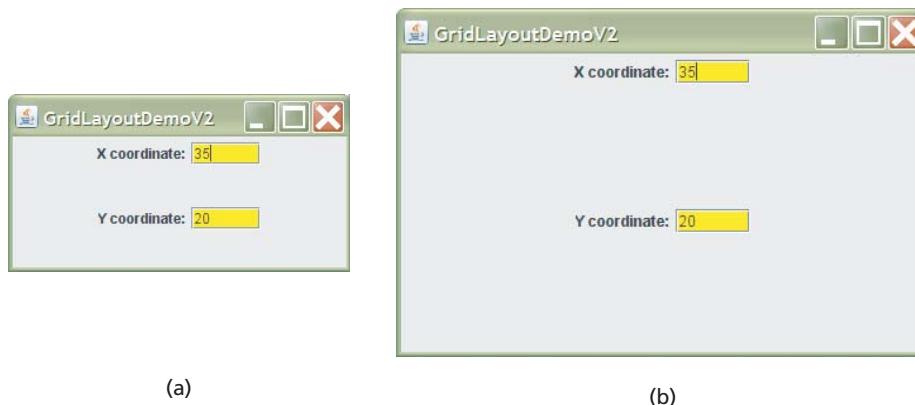
A `GridLayout` manager ignores the preferred size of a component, and the component is stretched in order to fill the cell (Figure 20.13). One common technique to avoid the components from being stretched, is to first place them in a panel (using a `FlowLayout` manager) and then inserting the panel into the container that uses a `GridLayout` manager. This technique is used in Program 20.6, and the result is shown in Figure 20.15. We use two panels for grouping a label with its corresponding text field. The content pane now uses a 2×1 grid to arrange the panels (Figure 20.16):

```
// Create and set a GridLayout with a 2 x 1 grid.
contentPane.setLayout(new GridLayout(2,1));

// Construct the component hierarchy.
xInputPanel.add(xJLabel);
xInputPanel.add(xInput);
yInputPanel.add(yJLabel);
yInputPanel.add(yInput);
contentPane.add(xInputPanel); // [0,0]
contentPane.add(yInputPanel); // [1,0]
```

FIGURE 20.15

GUI avoiding component stretching (Program 20.6)

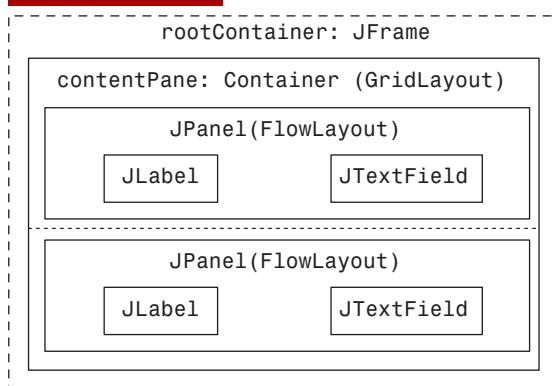


(a)

(b)

FIGURE 20.16

Component hierarchy with panels (Program 20.6)





PROGRAM 20.6 GUI design avoiding component stretching

```
import java.awt.Color;
import java.awt.Container;
import java.awt.GridLayout;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

class GridLayoutDemoV2 extends JFrame {

    // Components
    private JLabel xJLabel;
    private JLabel yJLabel;
    private JTextField xInput;
    private JTextField yInput;
    private JPanel xInputPanel;
    private JPanel yInputPanel;

    GridLayoutDemoV2() {
        // Set a suitable title for the frame.
        super("GridLayoutDemoV2");

        // Create two labels and two text fields.
        xJLabel = new JLabel("X coordinate:");
        yJLabel = new JLabel("Y coordinate:");
        xInput = new JTextField(5);
        yInput = new JTextField(5);
        xInput.setBackground(Color.yellow);
        yInput.setBackground(Color.yellow);
        xInputPanel = new JPanel();
        yInputPanel = new JPanel();

        // Get the content pane for the frame.
        Container contentPane = getContentPane();

        // Create and set a GridLayout with a 2x1 grid.
        contentPane.setLayout(new GridLayout(2, 1));

        // Construct the component hierarchy.
        xInputPanel.add(xJLabel);
        xInputPanel.add(xInput);
        yInputPanel.add(yJLabel);
        yInputPanel.add(yInput);
        contentPane.add(xInputPanel); // [0,0]
        contentPane.add(yInputPanel); // [1,0]
```



```

// Terminate if the frame is closed.
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Show the GUI.
pack();
setVisible(true);
}

}

public class GridLayoutClientV2 {
    public static void main(String[] args) {
        GridLayoutDemoV2 gui = new GridLayoutDemoV2();
    }
}

```

BEST PRACTICES

Before implementing the GUI, it is worthwhile making a sketch of your GUI and drawing the component hierarchy to identify the containers, components and the layout managers.

20.5 Event-driven programming

Events

We will concentrate on events that are generated when the user interacts with the program via GUI control components, since the handling of such events is crucial for the program to function as desired. Events occur not only as a result of user actions (for example, when the user presses the mouse button, moves the mouse, presses a key, selects from a menu, and so on), but can also occur when the screen contents change and need updating, for example.

The abstract class `java.awt.AWTEvent` is the superclass for all GUI event classes. Events are objects of subclasses defined in the `java.awt.event` package. These objects represent different categories of events that can occur in a GUI-based application. Objects of the class `ActionEvent` represent events that occur when a button is clicked on, or when the Enter key is typed in a text field, or when a selection is made from a menu. Objects of the class `WindowEvent` represent events that can occur when different operations are executed on a window, for example, when a window is closed, or when it is opened. In both cases, an event object encapsulates all information that identifies the event.

The class `AWTEvent` has methods that can be used to access information about an event. Table 20.10 shows the method `getSource()`, that can be called on an event in order to identify *the source* of the event.



To illustrate event-driven programming, we will mainly use events represented by the classes `ActionEvent` and `WindowEvent`. Handling of other types of events is essentially the same as far as the approach is concerned.

TABLE 20.10 Source identification for events

<code>java.awt.AWTEvent</code>	
<code>Object getSource()</code>	Returns the reference value of the source that generated the event.

Event delegation model

A source is a component that can generate events in response to user actions in the GUI. A listener is an object that wants to be informed about events when these occur. The source can communicate information about events to the listener if the following two requirements are met:

- 1 The source must have a reference to the listener.
- 2 The source can use this reference to call a specific method in the listener when an event occurs. The event is passed as an argument in the call to this method.

Requirement (1) implies that a listener interested in receiving information about a specific event, must *beforehand* register itself with source(s) that can generate this event. A listener interested in XEvent events, must be registered by calling the `addXListener()` method in the source and passing the reference value of the listener as argument. For example, a listener interested in `ActionEvent` events from a `JButton` source object must be registered by calling the `addActionListener()` method in the `JButton` class, as follows (Table 20.11):

```
button.addActionListener(buttonListener);
```

Requirement (2) implies that the listener must guarantee that a specific method exists by implementing a *listener interface* for this event. The listener must implement the `XListener` interface in order to receive XEvents. For example, a listener interested in `ActionEvent` events must implement the `ActionListener` interface that specifies the `actionPerformed()` method (Table 20.12):

```
class Listener implements ActionListener {
    ...
    public void actionPerformed(ActionEvent event) {
        // implementation
    }
    ...
}
```

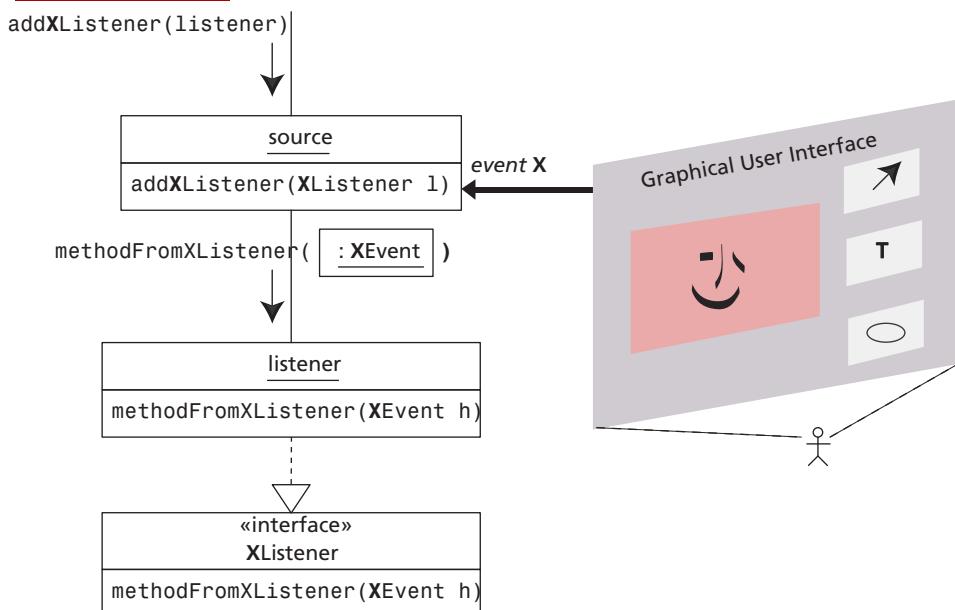
With the setup outlined above and illustrated in Figure 20.17, a source will call a specific method in all listeners that are registered with it, ensuring that they receive information about a specific event type when this event occurs. The method `actionPerformed()` is called in the listener, passing an `ActionEvent` object as argument, when an event of this type occurs. Any object can be a listener, as long as it implements the correct listener



interface (`XListener`) for specific events (`XEvent`), and registers itself (`addXListener()`) with a source that generates these events.

Note that a source can generate different types of events and can have several listeners. A listener can receive information about different types of events from different sources. If a source wants to be a listener to events it generates itself, the requirements for registering with the source and implementing the required listener interfaces must still be fulfilled.

FIGURE 20.17 The event delegation model



A listener interested in `XEvent` events, must register with the source using the `addXListener()` method. The listener must implement the `XListener` interface in order to receive events of type `XEvent`. The listener is informed about events of type `XEvent` via `methodFromXListener()` in the `XListener` interface.

Classes `ActionEvent` and `WindowEvent`

Table 20.11 gives an overview of the events represented by the classes `ActionEvent` and `WindowEvent`. The table shows sources that generate these events, and methods that these sources provide for the registering and removing of listeners from the sources of these events. Note that the formal parameter `l` in the `addActionListener()` method has the reference type `ActionListener`. Any call to this method to register an object that does *not* implement the `ActionListener` interface, will be reported by the compiler. Listener registration and implementation of the required listener interface ensures that event delegation will work at runtime.

An `ActionEvent` is generated when a button (`JButton`) is clicked, and also when the Enter key is typed in a text field (`JTextField`). The `Window` class generates `WindowEvents`, and inheritance implies that top-level windows such as frames (`JFrame`) also generate this event.



Table 20.12 shows the methods in the listener interfaces `ActionListener` and `WindowListener`. The `ActionListener` interface has only one method, `actionPerformed()`, while the `WindowListener` interface has several methods. Each method in the `WindowListener` interface corresponds to a specific kind of `WindowEvent`. The method `windowClosing()` is called when a window is closed. A listener wanting to close the program when a top-level window is closed, will only be interested in implementing the `windowClosing()` method. However, the listener must also implement the remaining methods of the `WindowListener` interface. Such a listener can declare *stubs* for the other methods, for example:

```
public void windowActivated(WindowEvent event) {} // empty method body
```

The listener will be informed about the other kinds of `WindowEvent` events, but chooses to do nothing about them by executing an empty method body.

TABLE 20.11 Selected events and their sources

Event	Event sources	Methods in the sources for registering and removing listeners of a specific type of events
ActionEvent	JButton, JTextField, JToggleButton, JCheckBox, JRadioButton, JMenuItem, JMenu, JCheckBoxMenuItem, JRadioButtonMenuItem	void addActionListener(ActionListener l) void removeActionListener(ActionListener l)
WindowEvent	Window and all its subclasses (for example, JFrame and JDialog)	void addWindowListener(WindowListener l) void removeWindowListener(WindowListener l)

TABLE 20.12 Selected listener interfaces

Listener interfaces in the java.awt.event package	Methods in the listener interface
ActionListener	void actionPerformed(ActionEvent event)
WindowListener	void windowClosing(WindowEvent event) void windowClosed(WindowEvent event) void windowOpened(WindowEvent event) void windowIconified(WindowEvent event) void windowDeiconified(WindowEvent event) void windowActivated(WindowEvent event) void windowDeactivated(WindowEvent event)



Programming paradigms for event handling

The examples below demonstrate event-driven programming in the design of GUI for simple user interactions. The programs are primarily intended to illustrate the different programming paradigms for handling events.

External listeners

The GUI shown in Figure 20.18 comprises a single window with a button. The main window changes color when the user clicks on the button. The class diagram in Figure 20.19 shows how the application is constructed. The class `SimpleWindow1` implements a top-level window (`JFrame`), that has a button (the reference `colorButton` of the type `JButton`) and an external listener (the reference `listener` of the type `ExternalListener1`). In addition, the class `SimpleWindow1` has the field `colorToggle` that is used for changing the colour, and the method `changeColor()`, that changes the colour in the window. Objects of the class `JButton` generate `ActionEvent` events when the button is clicked. The class `ExternalListener1` implements the `ActionListener` interface, so that objects of this class can handle `ActionEvents`. The class `ExternalListener1` has a reference to the main window (given by the reference `app`), so that the window can be informed about changing the color.

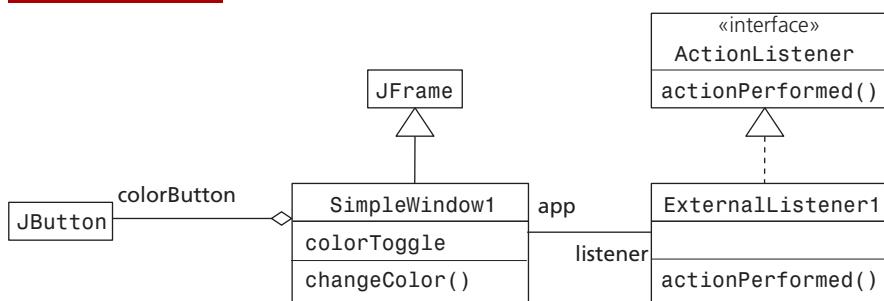
FIGURE 20.18 A simple GUI application (Program 20.7)



(a)

(b)

FIGURE 20.19 Class diagram for Program 20.7

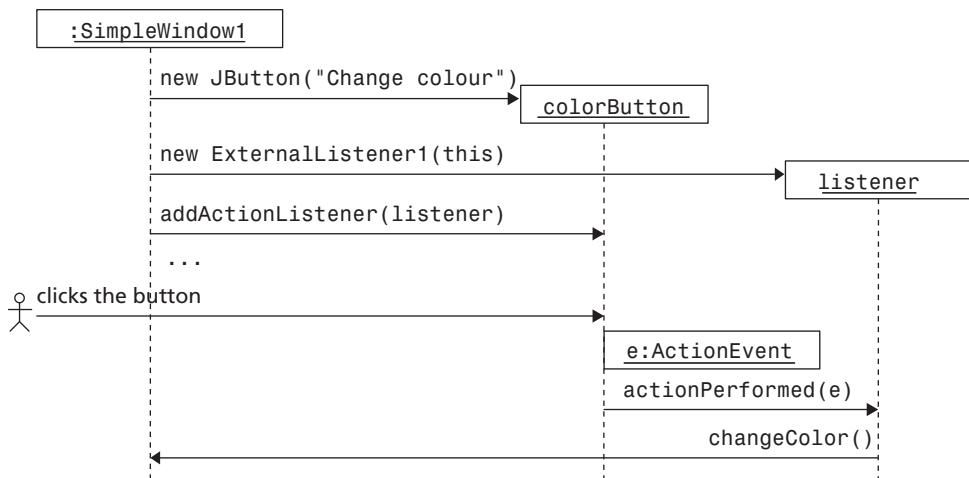


Interaction between the user and the application at runtime is shown in Figure 20.20. The main window first creates objects for the colour button and the listener. When creating the listener, the reference value of the main window (given by the reference `this`) is passed to the listener. The main window then registers the listener with the source (given by the reference `colorButton`) by calling the `addActionListener()` method, so that the



listener will receive `ActionEvents`. When the user clicks on the button, an `ActionEvent` is generated. This event is passed to the listener by calling the `actionPerformed()` method. This call is made as part of the event handling process, and we need not write code for it. The listener responds to the event by sending a message to the main window about changing the colour, by calling the `changeColor()` method.

FIGURE 20.20 Listener registration and event delegation in Program 20.7



Program 20.7 shows the implementation of the application. Since we have already seen several examples of building component hierarchies for GUIs, here we focus on event handling.

The class `SimpleWindow1` has the following fields:

```

// (2) Components
private JButton colorButton;           // the source
private ExternalListener1 listener;     // the listener

// (3) Other fields
private boolean colorToggle;
  
```

Listener registration takes place in the constructor of the class `SimpleWindow1`. It is a good idea to do so here, since the GUI is created when the main window is created.

```

listener = new ExternalListener1(this);      // (4)
colorButton.addActionListener(listener);        // (5)
  
```

Note that we pass the reference value of the main window to the listener at (4), which then has the necessary information from the main window to handle the event. Note also that the source (the colour button) generates an `ActionEvent` when the button is clicked. Therefore we call the `addActionListener()` method in the `JButton` class to register the listener at (5).

The method `changeColor()` at (6) uses the field `colorToggle` in the class `SimpleWindow1` to toggle the colour in the main window. It is the background colour of the *content pane* of the main window that is changed, since the content pane is always drawn over the main



window when the window is shown on the screen. We change the boolean value in the `colorToggle` field (using the negation operator `!`) each time the user clicks the button, so that the colour alternates between two colours.

Event classes and listener interfaces are found in the `java.awt.event` package. An `import` statement can be declared to use a class from this package:

```
import javax.awt.event.ActionEvent;
import javax.awt.event.ActionListener;
```

The class `ExternalListener1` must declare that it implements the `ActionListener` interface:

```
class ExternalListener1 implements ActionListener {           // (7)
```

The class has a field for storing the reference value of the main window. This field is initialised when the listener is created:

```
// Reference to the main window.
private SimpleWindow1 app;                                // (8)
```

```
ExternalListener1(SimpleWindow1 window) {                  // (9)
    app = window;
}
```

The reference `app` to the main window is used in the `actionPerformed()` method at (10) to change the colour in the main window:

```
// Called when the user clicks the colour button.          (10)
public void actionPerformed(ActionEvent event) {
    app.changeColor();
}
```

PROGRAM 20.7 A simple application with external listener (version 1)

```
import java.awt.Color;
import java.awt.Container;
import java.awt.FlowLayout;

import javax.swing.JButton;
import javax.swing.JFrame;

class SimpleWindow1 extends JFrame {

    // (2) Components
    private JButton colorButton;      // the source
    private ExternalListener1 listener; // the listener

    // (3) Other fields
    private boolean colorToggle;

    SimpleWindow1() {
        // Set suitable title for the frame.
```



```
super("Simple window");

// Create a button.
colorButton = new JButton("Change colour");

// Get the content pane.
Container contentPane = getContentPane();

// Create and set a layout manager, and add
// the button to the content pane.
contentPane.setLayout(new FlowLayout(FlowLayout.CENTER));
contentPane.add(colorButton);

// Create a listener and register it with the source (i.e. button)
// to receive ActionEvents.
listener = new ExternalListener1(this);                      // (4)
colorButton.addActionListener(listener);                         // (5)

// Terminate if the frame is closed.
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Show the GUI.
pack();
setVisible(true);
}

// Change background colour for the content pane.
public void changeColor() {                                     // (6)
    Container contentPane = getContentPane();
    if (colorToggle)
        contentPane.setBackground(Color.cyan);
    else
        contentPane.setBackground(Color.yellow);
    colorToggle = !colorToggle;
}
}

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

class ExternalListener1 implements ActionListener {           // (7)

    // Reference to the main window.
    private SimpleWindow1 app;                                // (8)

    public ExternalListener1(SimpleWindow1 window) {          // (9)
        app = window;
    }

    // Called when the user clicks the colour button.          (10)
}
```



```

public void actionPerformed(ActionEvent event) {
    app.changeColor();
}
}

public class SimpleWindowClient1 {
    public static void main(String[] args) {
        SimpleWindow1 gui = new SimpleWindow1();
    }
}

```

Explicitly terminating an application

We called the `setDefaultCloseOperation()` method in the main window, with the argument value `JFrame.EXIT_ON_CLOSE`, to terminate the application when the user clicks on the window close button:

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

We will modify the application in Program 20.7 so that it explicitly handles terminating the program when the main window is closed. The source code for the new application is shown in Program 20.8. The class `SimpleWindow2` uses the class `ExternalListener2` to handle both the changing of the colour and terminating the program. We have intentionally not used inheritance to implement the classes `SimpleWindow2` and `ExternalListener2`, so as to keep the focus on event handling. It is a useful exercise to implement these classes using inheritance.

Program 20.8 shows the changes in the class `SimpleWindow2` in relation to the class `SimpleWindow1` in Program 20.7. It is the `Window` class (and its subclasses, for example, `JFrame`) that generate `WindowEvents` when the user clicks on the window close button. The class `SimpleWindow2` inherits this behaviour, and is a source of `WindowEvents`, so that we can register the listener with it:

```
addWindowListener(listener); // (3)
```

In Program 20.8 the class `ExternalListener2` implements the `WindowListener` interface in order to receive `WindowEvents`, (4). Note that the listener must implement *all* methods in this interface, even if only the method `windowClosing()` is of interest. (The other methods in the `WindowListener` interface are implemented as stubs, so that a call to one of these does nothing, (6).) Implementation of the `windowClosing()` method shows the recommended way to terminate a GUI-based application:

```

public void windowClosing(WindowEvent event) { // (5)
    System.out.println("Terminating the program.");
    app.dispose();
    System.exit(0);
}

```

First, all window resources used in the GUI are freed by calling the `dispose()` method inherited from the `Window` class. In addition to the resources that are explicitly specified in the program, other resources are also used to handle GUI-based applications at runtime. Then, the `System.exit()` method is called. We pass the value 0 to indicate that



the program terminated normally. A call to this method will terminate an application, regardless of where this call is executed in the program. It is a good idea to save any data and undertake any cleaning up that might be necessary, before this method is called.

PROGRAM 20.8 External listener terminates the application (version 2)

```

...
class SimpleWindow2 extends JFrame {
    ...
    ExternalListener2 listener;                                // the listener

    SimpleWindow2() {
        ...
        // Create listener and register it with the source (i.e. the button)
        // to receive ActionEvents.
        listener = new ExternalListener2(this);                // (1)
        colorButton.addActionListener(listener);                 // (2)

        // Register the same listener with the source (i.e. the frame)
        // to receive WindowEvents.
        addWindowListener(listener);                          // (3)
        ...
    }
}

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

class ExternalListener2 implements ActionListener, WindowListener { // (4)

    // Reference to the main window.
    private SimpleWindow2 app;

    public ExternalListener2(SimpleWindow2 mainWindow) {
        app = mainWindow;
    }

    // Called when the user clicks the colour button.
    public void actionPerformed(ActionEvent event) {
        app.changeColor();
    }

    // Called when the user clicks the window close box.
    public void windowClosing(WindowEvent event) {                  // (5)
        System.out.println("Terminating the program.");
        app.dispose();
        System.exit(0);
    }
}

```



```
// Stubs for unused methods from the WindowListener interface.      (6)
public void windowClosed(WindowEvent event) { }
public void windowOpened(WindowEvent event) { }
public void windowIconified(WindowEvent event) { }
public void windowDeiconified(WindowEvent event) { }
public void windowActivated(WindowEvent event) { }
public void windowDeactivated(WindowEvent event) { }

}

public class SimpleWindowClient2 {
    public static void main(String[] args) {
        SimpleWindow2 gui = new SimpleWindow2();
    }
}
```

Listener adapter classes

Program 20.8 shows a typical situation where a listener is interested in handling specific user actions, but must implement all methods in the listener interface for the type of events that correspond to these user actions. The package `java.awt.event` provides an *adapter class* for each listener interface that has more than one method. An adapter class implements stubs for all methods in a given listener interface. A listener can inherit the stubs from the adapter class and override the methods the listener is interested in.

Program 20.9 shows the use of the `WindowAdapter` class. As opposed to the listener in Program 20.8, the new listener in Program 20.9 inherits from the `WindowAdapter` class the stubs for all the methods in the `WindowListener` interface:

```
class ExternalListener3 extends WindowAdapter implements ActionListener{//(4)
```

The listener overrides the `windowClosing()` method at (5) in order to handle `WindowEvents` that indicate the window was closed.

PROGRAM 20.9 A simple application with external listener adapter (version 3)

```
...
class SimpleWindow3 extends JFrame {
    ...
    ExternalListener3 listener;                                // the listener

    SimpleWindow3() {
        ...
        // Create listener and register it with the source (i.e. the button)
        // to receive ActionEvents.
        listener = new ExternalListener3(this);           // (1)
        colorButton.addActionListener(listener);           // (2)

        // Register the same listener with the source (i.e. the frame)
```



```

    // to receive WindowEvents.
    addWindowListener(listener);                                // (3)
    ...
}

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

class ExternalListener3 extends WindowAdapter implements ActionListener { // (4)

    // Reference to the main window.
    private SimpleWindow3 app;

    public ExternalListener3(SimpleWindow3 mainWindow) {
        app = mainWindow;
    }

    // Called when the user clicks the colour button.
    public void actionPerformed(ActionEvent h) {
        app.changeColor();
    }

    // Called when the user clicks the window close box.
    public void windowClosing(WindowEvent h) {                               // (5)
        System.out.println("Terminating the program.");
        app.dispose();
        System.exit(0);
    }
}

public class SimpleWindowClient3 {
    public static void main(String[] args) {
        SimpleWindow3 gui = new SimpleWindow3();
    }
}

```

The main window as listener

Program 20.10 is a variant of Program 20.8, where the main window handles all the events, instead of using an external listener. The class `SimpleWindow4` must declare that it implements both the `ActionListener` and the `WindowListener` interfaces:

```

class SimpleWindow4 extends JFrame
    implements ActionListener, WindowListener { // (1)

```

In the constructor, the frame must first register itself with the colour button in order to receive `ActionEvents`:



```
colorButton.addActionListener(this); // (2)
```

Afterwards the frame must register with itself in order to receive WindowEvents:

```
addWindowListener(this); // (3)
```

The frame is both the source and the listener for WindowEvents, and follows the setup in the event delegation model. Any object can be a listener as long as it implements the right listener interface and registers itself with the source, even if both roles are carried out by the same object.

The class `SimpleWindow4` cannot extend the `WindowAdapter` class in order to implement the `WindowListener` interface, since it already extends the `JFrame` class. Adapter classes must be used with care, since a listener that inherits from an adapter class, cannot inherit from other classes.

PROGRAM 20.10 Main window as listener (version 4)

```
import java.awt.Color;
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

import javax.swing.JButton;
import javax.swing.JFrame;

class SimpleWindow4 extends JFrame
    implements ActionListener, WindowListener { // (1)
    // Components
    private JButton colorButton; // the source

    // Other fields
    private boolean colorToggle;

    SimpleWindow4() {
        // Set suitable title for the frame.
        super("Simple window 4");

        // Create a button.
        colorButton = new JButton("Change colour");

        // Get the content pane.
        Container contentPane = getContentPane();

        // Create a layout manager, and add button to the content pane.
        contentPane.setLayout(new FlowLayout(FlowLayout.CENTER));
        contentPane.add(colorButton);
```



```

// Register the frame as listener with the source (i.e. button)
// to receive ActionEvents.
colorButton.addActionListener(this);                                // (2)

// Register the listener (i.e. the frame) with the source (i.e. the frame)
// to receive WindowEvent events.
addWindowListener(this);                                         // (3)

// Show the GUI.
pack();
setVisible(true);
}

// Change background colour for the content pane.
private void changeColor() {
    if (colorToggle)
        getContentPane().setBackground(Color.cyan);
    else
        getContentPane().setBackground(Color.yellow);
    colorToggle = !colorToggle;
}

// Called when the user clicks on the colour button.
public void actionPerformed(ActionEvent event) {                  // (4)
    changeColor();
}

// Called when the user clicks on the window close box.
public void windowClosing(WindowEvent event) {                  // (5)
    System.out.println("Terminating the program");
    dispose();
    System.exit(0);
}

// Stubs for unused methods from the WindowListener interface.      (6)
public void windowClosed(WindowEvent event) { }
public void windowOpened(WindowEvent event) { }
public void windowIconified(WindowEvent event) { }
public void windowDeiconified(WindowEvent event) { }
public void windowActivated(WindowEvent event) { }
public void windowDeactivated(WindowEvent event) { }

}

public class SimpleWindowClient4 {
    public static void main(String[] args) {
        SimpleWindow4 gui = new SimpleWindow4();
    }
}

```



One listener and several sources with the same type of event

Program 20.9 showed an example of a situation where a listener (the class `ExternalListener2`) was registered with two sources (objects of the classes `JFrame` and `JButton`, respectively) that generated different types of events of the classes `WindowEvent` and `ActionEvent`, respectively). Program 20.11 illustrates a situation where a listener (the class `EchoWithEvents`) is also registered with two sources (objects of the classes `JButton` and `JTextField`), but both sources generate the same type of events (of the class `ActionEvent`).

Program 20.11 is the completion of Program 20.2 that was missing event handling. The GUI for Program 20.11 is shown in Figure 20.21. When the user types the Enter key in a text field, an `ActionEvent` is generated. The same type of event is also generated when the user clicks on a button. The main window in Program 20.11 is interested in `ActionEvents` from the input field, so that it can duplicate the text from this field in the echo field. The main window is also interested in `ActionEvents` from the Done button, so that it can terminate the program. In Program 20.11, the frame (i.e. the main window) is a listener for `ActionEvents` from two sources: the input field and the Done button. In the constructor of the class `EchoWithEvents` we register the frame (the reference `this`) as a listener with both the input field and the Done button:

```
inputField.addActionListener(this);                                // (2)
...
doneButton.addActionListener(this);                               // (3)
```

Since the listener handles `ActionEvents`, the class `EchoWithEvents` must implement the `ActionListener` interface:

```
class EchoWithEvents extends JFrame implements ActionListener { // (1)}
```

and provide an implementation of the method `actionPerformed()` specified in this interface:

```
public void actionPerformed(ActionEvent event) {           // (4)
    if (event.getSource() == inputField) {
        ...
        // the source was the input field
    } else {
        ...
        // the source was the Done button.
    }
}
```

Since both the input field and the Done button will call the method `actionPerformed()` with an `ActionEvent` as argument, the listener must be in a position to identify the source in order to execute the right action. The method `getSource()` returns the reference value of the source that generated the event, so that it is possible to determine if it, for example, equals the reference value of the input field (`event.getSource() == inputField`). If the source is the input field, the text is fetched from the field with the method `getText()`, and the text is written in the echo field with the method `setText()`. If the source is the Done button, the program is terminated.

**FIGURE 20.21** Echo GUI with event handling (Program 20.11)**PROGRAM 20.11** Echo GUI with event handling

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

class EchoWithEvents extends JFrame implements ActionListener { // (1)

    // Components
    private JPanel panel1, panel2, panel3;
    private JButton doneButton;
    private JLabel inputLabel;
    private JLabel echoLabel;
    private JTextField inputField;
    private JTextField echoField;

    EchoWithEvents() {
        // Set suitable title for the frame.
        super("Echo Input (Event handling)");

        // Create GUI components.
        panel1 = new JPanel();
        panel1.setBackground(Color.yellow);
        panel2 = new JPanel();
        panel2.setBackground(Color.pink);
```



```

panel3 = new JPanel();
panel3.setBackground(Color.cyan);

doneButton = new JButton("Done");

InputLabel = new JLabel("Input field:");
echoLabel = new JLabel("Echo field:");

inputField = new JTextField("Type here. " + "End with Enter key.", 10);
inputField.setEditable(true);

echoField = new JTextField("", 10);
echoField.setEditable(false);
echoField.setBackground(Color.lightGray);

// Get the content pane for the frame.
Container contentPane = getContentPane();
contentPane.setBackground(Color.white);

// Set a layout manager for the content pane.
contentPane.setLayout(new GridLayout(3, 1));

// Construct the component hierarchy.
panel1.addInputLabel();
panel1.add(inputField);
panel2.add(echoLabel);
panel2.add(echoField);
panel3.add(doneButton);
contentPane.add(panel1, BorderLayout.NORTH);
contentPane.add(panel2, BorderLayout.CENTER);
contentPane.add(panel3, BorderLayout.SOUTH);

// Register the frame as listener with the text field
// to receive ActionEvents.
inputField.addActionListener(this); // (2)

// Register the frame as listener with the Done button
// to receive ActionEvents.
doneButton.addActionListener(this); // (3)

// Terminate if the frame is closed.
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Show the GUI.
pack();
setVisible(true);
}

public void actionPerformed(ActionEvent event) { // (4)
    if (event.getSource() == inputField) {

```



```

        String inputString = inputField.getText();
        echoField.setText(inputString);
    } else { // the source was the Done button.
        System.out.println("Done.");
        dispose();
        System.exit(0);
    }
}

public class EchoWithEventsClient {
    public static void main(String[] args) {
        EchoWithEvents gui = new EchoWithEvents();
    }
}

```

BEST PRACTICES

Event handling is the key to successful GUIs. Carefully identify the events, sources and listeners in your GUI-based application.

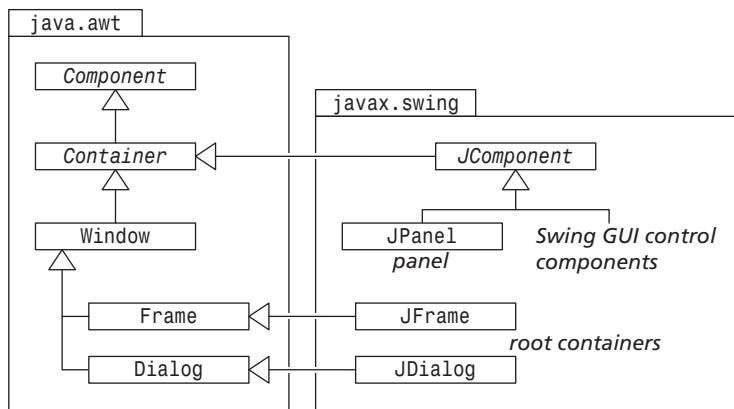
20.6 Dialogue windows

So far, we have only seen GUI examples where one top-level window comprises the whole graphical user interface. Our approach constructs a component hierarchy consisting of GUI control components and panels in a single frame, that is shown on the screen. No other windows are created during the interaction with the user. We will now illustrate a programming paradigm for building a GUI-based application which uses several windows. User actions in a window can result in a new window being created.

The class `JDialog` is useful for designing applications that use several windows. Figure 20.22 shows the inheritance hierarchy for the `JDialog` class. A `JDialog` object is a container (subclass of the `Container` class) that inherits from a top-level window (subclass of the `Window` class). In addition, the class `JDialog` inherits properties and behaviour from the `Dialog` class that makes it well-suited for creating dialogue windows. As the name says, a dialogue window can be used to obtain information from the user or for presenting information to the user.



FIGURE 20.22

Partial inheritance hierarchy showing the **JDialog** class

A dialogue window can have a title, menus and borders. Similar to a frame (the `JFrame` class), a dialogue window is also a root container. It has a content pane that can be used to build a component hierarchy that can be shown on the screen in its own window. Like other root containers, a dialogue window cannot be placed in other containers.

The class `JDialog` has many methods common with the `JFrame` class, since both root containers inherit from the `Window` class. The method `getContentPane()` returns the content pane of the dialogue window, and the method `pack()` packs the components in its component hierarchy. Before a dialogue window is closed, resources bound to it must be freed by calling the `dispose()` method.

The method `setDefaultCloseOperation()` in Table 20.13 can be used to specify the operation to be executed when a dialogue window is closed. Unlike the `JFrame` class, there is no constant defined to exit the application when the dialogue window is closed.

A dialogue window is also *temporary* and usually disappears after the user has finished with it. Such a window can only be created as *property* of an *owner*, that can either be a frame or another dialogue window. Two constructors for creating dialogue windows are shown in Table 20.13. A dialogue window is automatically closed when the owner window is closed, i.e. a dialogue window exists as long as the owner window exists.

A dialogue window can be *modal*, i.e. no other windows in the application can be accessed while this dialogue window is visible, or *non-modal*, i.e. other windows in the application can be accessed while this dialogue window is visible. The method `setModal()` in Table 20.13 can be used to set the behaviour of a dialogue window.

TABLE 20.13

Selected constructor and methods of the root container **JDialog**

javax.swing.JDialog	
<code>JDialog(Dialog owner, String title, boolean modal)</code>	Creates a modal/non-modal dialogue window with the specified <code>title</code> , and the specified <code>owner</code> which is a <i>dialogue window</i> . There are overloaded constructors where either <code>title</code> or <code>modal</code> or both can be omitted.



javax.swing.JDialog	
<code>JDialog(Frame owner, String title, boolean modal)</code>	Creates a modal/non-modal dialogue window with the specified <code>title</code> , and the specified <code>owner</code> which is a <i>frame</i> . There are overloaded constructors where either <code>title</code> or <code>modal</code> or both can be omitted.
<code>void setDefaultCloseOperation(int operation)</code>	Sets the operation that should be executed when the dialogue window is closed, and after all registered <code>WindowListener</code> objects have been notified. The parameter <code>operation</code> can have one of the values defined by the <code>JDialog</code> class: <code>DO NOTHING ON CLOSE</code> (the program itself must undertake proper closing and disposing of the dialogue window), <code>HIDE ON CLOSE</code> (the default operation that hides the dialogue window, but does not dispose of it), <code>DISPOSE ON CLOSE</code> (ensures proper closing and disposing of the dialogue window).
<code>void setModal(boolean b)</code>	Sets the dialogue window to be modal or non-modal, depending on whether the value of the parameter <code>b</code> is <code>true</code> or <code>false</code> , respectively.

Application using dialogue windows

We will write a new version of Program 20.11 that uses a dialogue window for reading an integer from the user, and afterwards shows the number in the owner window. The class `EchoWithDialog` in Program 20.12 implements a frame that is the owner window, and the class `IntegerDialogWindow` in Program 20.13 implements a dialogue window.

The GUI for the application is shown in Figure 20.23. When the user clicks on the "Type an integer ..." button in the owner window (`EchoWithDialog`), a new dialogue window (`IntegerDialogWindow`) is created for reading an integer. The dialogue window is modal, so that the user cannot access the owner window while the dialogue window is visible. The user can close the dialogue window by clicking on the OK button, but this is only possible if a legal integer is typed in the input field. Only a valid value can be sent back to the owner window. The user can close the dialogue window by clicking on its window close button, but then no value is returned to the owner window.

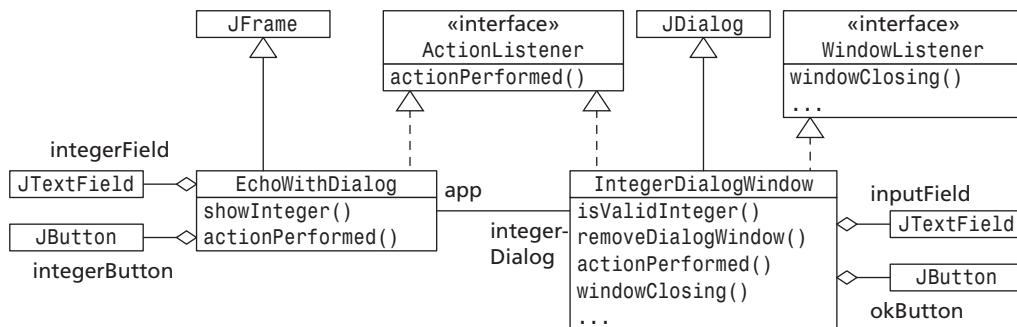
FIGURE 20.23 **GUI using the `JDialog` class (Program 20.12)**





The class diagram for the application is shown in Figure 20.24. The class `IntegerDialogWindow` has a field (`app`) that refers to the owner window, and the class `EchoWithDialog` has a field (`integerDialog`) that refers to the dialogue window. Both classes, `EchoWithDialog` and `IntegerDialogWindow`, implement the `ActionListener` interface and can handle `ActionEvents`. In addition, the class `IntegerDialogWindow` implements the `WindowListener` interface, so that it can be a listener for `WindowEvents`. Unlike the class `EchoWithDialog`, the class `IntegerDialogWindow` handles closing of the dialogue window explicitly, instead of using the `setDefaultCloseOperation()` method.

FIGURE 20.24 Class diagram for GUI in Figure 20.23



Listener registration and event delegation in the owner window `EchoWithDialog` is shown in Figure 20.25. The class `EchoWithDialog` creates a text field (`integerField`) and a button (`integerButton`). It calls the `addActionListener()` method to register itself with the button, in order to receive `ActionEvents`. When the user clicks on the button, an `ActionEvent` is generated. The button calls the method `actionPerformed()` in the `EchoWithDialog` object, sending the event as argument. This results in the creation of a dialogue window (`integerDialog`). The reference `this`, that refers to the owner window, is sent as argument in the constructor call to the `IntegerDialogWindow` class.

FIGURE 20.25 Setup for event delegation in the owner window

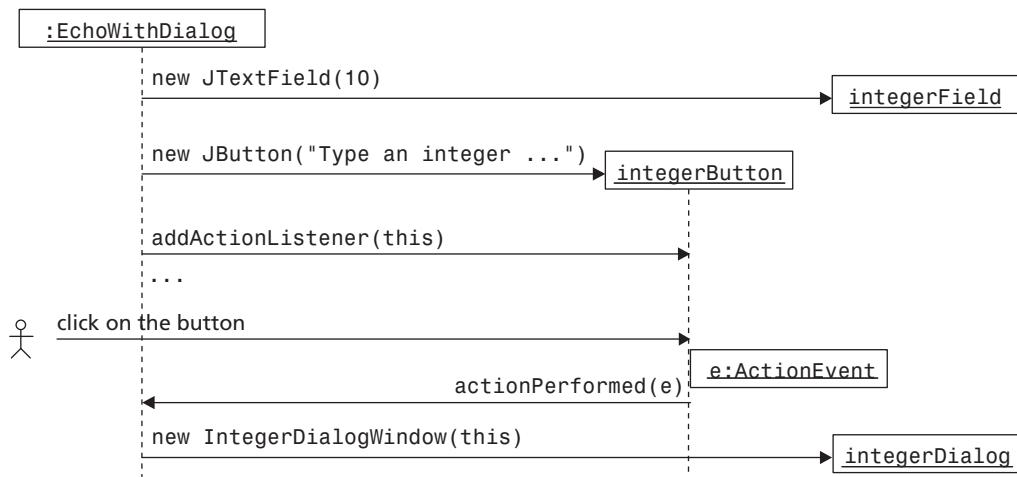
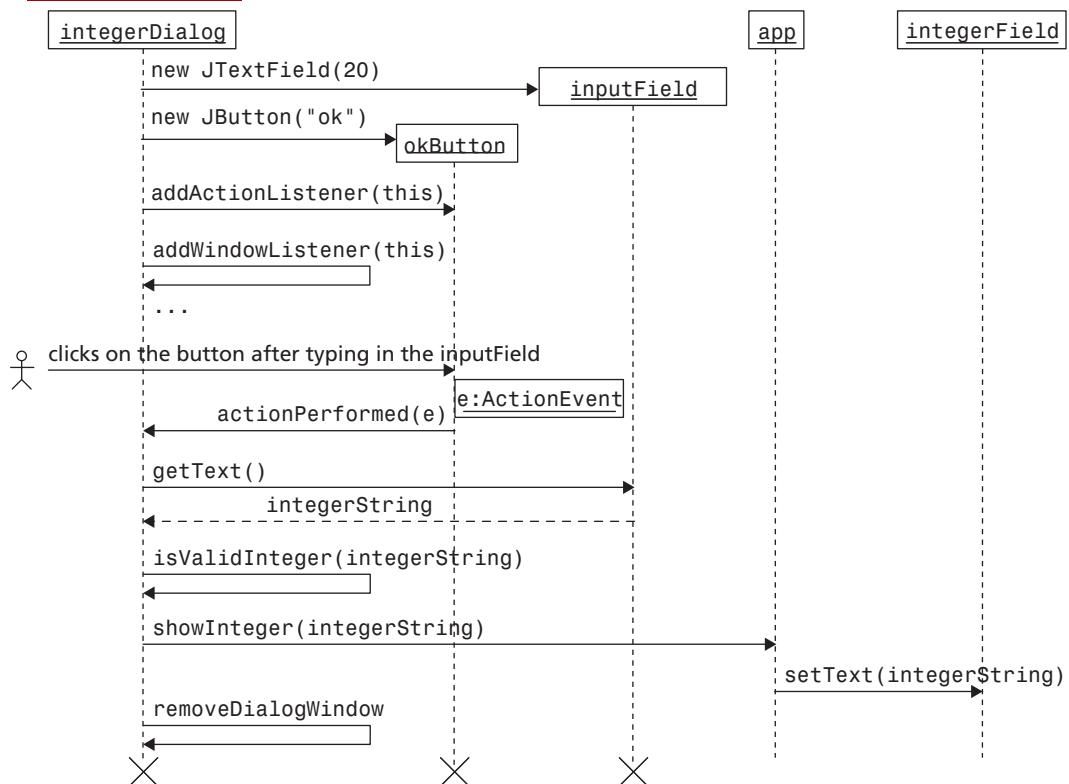




Figure 20.26 shows listener registration and event delegation in the dialogue window. Sequence of actions continues from Figure 20.25 that ended with the creation of a dialogue window (`integerDialog`). As we can see from Figure 20.26, the dialogue window creates a text field (`inputField`) and a button (`okButton`). Thereafter the dialogue window registers itself with the button in order to receive `ActionEvents`, and with itself in order to receive `WindowEvents`. The user can write and edit characters in the text field, even if there are no listeners registered with this component.

Processing of the user action begins when the user clicks on the OK button. An `ActionEvent` is generated by the button, and delegated to the dialogue window (call to the `actionPerformed()` method). The implementation of the `actionPerformed()` method reads the text from the input field (call to the `getText()` method) and determines whether the text contains a valid integer (call to the `isValidInteger()` method). The scenario in Figure 20.26 shows actions executed when the text is a valid integer. Since the dialogue window has a reference to the owner window (via the reference `app`), the dialogue window can send the text to the owner window (call to the `showInteger()` method). The owner window shows the text in its integer field (call to the `setText()` method). Finally, the dialogue window calls the method `removeDialogWindow()` to terminate the dialogue window, after which the dialogue window (and its field) are no longer accessible (marked with a big cross at the bottom of Figure 20.26).

FIGURE 20.26 Setup for event delegation in the dialogue window





Implementation of the class `EchoWithDialog` (Program 20.12) is based on the class diagram in Figure 20.24 and the sequence diagram in Figure 20.25. Implementation of the owner window follows the approach for creating a GUI in a frame.

The method `showInteger()` at (1) writes a valid integer as a string in the `integer` field of the owner window when it is called by the dialogue window.

The method `actionPerformed()` at (2) is called when the user clicks on the "Type an integer ..." button, and results in the creation of a dialogue window, with the `EchoWithDialog` object as the owner.

PROGRAM 20.12 Main window uses a dialogue window

```
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextField;

class EchoWithDialog extends JFrame implements ActionListener {

    // Components
    private JButton integerButton;
    private JTextField integerField;
    private IntegerDialogWindow integerDialog;

    EchoWithDialog() {
        // Set suitable title for the frame.
        super("Echo with Dialog");

        // Create GUI components.
        integerField = new JTextField(10);
        integerField.setEditable(false);
        integerButton = new JButton("Type an integer ...");

        // Set up the GUI.
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        contentPane.add(integerField);
        contentPane.add(integerButton);

        // Register the frame as listener with the button
        // to receive ActionEvents.
        integerButton.addActionListener(this);

        // Terminate if the frame is closed.
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    // Method to show integer value in the text field.
    void showInteger() {
        String str = integerField.getText();
        if (str.matches("\\d+"))
            integerDialog.show(str);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == integerButton)
            showInteger();
    }
}
```



```

    // Show the GUI.
    pack();
    setVisible(true);
}

// Show the integer in the text field. (1)
public void showInteger(String str) {
    integerField.setText(str);
}

// Called when the user clicks on the "Type an integer ..." button. (2)
public void actionPerformed(ActionEvent e) {
    integerDialog = new IntegerDialogWindow(this);
}
}

public class EchoWithDialogClient {
    public static void main(String[] args) {
        EchoWithDialog gui = new EchoWithDialog();
    }
}

```

Implementing a dialogue window

Implementation of the class `IntegerDialogWindow` is shown in Program 20.13. It is based on the class diagram in Figure 20.24 and the sequence diagram in Figure 20.26. The approach for constructing a GUI in a dialogue window is similar to the one for a frame. The components are added to the content pane of the dialogue window. Creation of a dialogue window requires an owner window:

```
super(frame, "Integer Dialog Window", true);      // (owner, title, modal)
```

We have also specified a title for the dialogue window, and that the dialogue window should be modal.

The method `isValidInteger()` at (2) checks that the text in the input field is a valid integer. The text in the input field must be converted to an integer to determine if it contains a valid integer. Wrapper classes for numerical values (for example, the classes `Integer`, `Long`, `Double`, `Float`) have a `parseT()` method for converting a string to a numerical type `T` (for example, to the primitive data types `int`, `long`, `double`, `float`). The static method `parseInt()` in the `Integer` class can be used to interpret the text as an integer. This method returns an integer as a result of the conversion, if conversion is possible. The method `isValidInteger()` will then return the boolean value `true`. Otherwise, the method `parseInt()` throws an unchecked exception of the type `NumberFormatException`. We must explicitly catch this exception with the help of a try-catch block. If we do not do that, any invalid text will result in this exception being thrown, resulting in the execution of the `isValidInteger()` method being terminated. The `isValidInteger()` method will



thereby not be able to determine whether the text was valid. Information about the exception will be written to the standard out unit, and *the application will continue*. It is the GUI monitor that takes over the control, regardless of whether the handling of an event results in an exception. With the setup shown in the `isValidInteger()` method, an exception of the type `NumberFormatException` will be caught in the catch block, so that the method will return the correct value (`false`) if the text cannot be converted to a valid integer.

The method `removeDialogWindow()` at (3) closes the dialogue window. No value is transferred to the owner window. First the window is made invisible, then all window resources are freed.

The method `actionPerformed()` at (4) is called when the user clicks on the OK button. If the text in the input field contains a valid integer, the owner window is asked to show the number in its window, and the dialogue window is closed. Otherwise nothing is done, and the GUI monitor again takes over the control.

PROGRAM 20.13 A dialogue window

```
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JTextField;

class IntegerDialogWindow extends JDialog implements ActionListener,
WindowListener {

    // Reference to the owner window.
    private EchoWithDialog app;

    // Components.
    private JTextField inputField;
    private JButton okButton;

    public IntegerDialogWindow(EchoWithDialog frame) {

        // Call to the superclass constructor.                      (1)
        super(frame, "Integer Dialog Window", true); // (owner, title, modal)
        app = frame;

        // Create GUI components.
        okButton = new JButton("OK");
        inputField = new JTextField(20);
        inputField.setEditable(true);
```



```
// Set up the GUI.  
Container contentPane = getContentPane();  
contentPane.add(inputField, BorderLayout.NORTH);  
contentPane.add(okButton, BorderLayout.SOUTH);  
  
// Register dialogue window with the source (i.e. the okButton button)  
// in order to receive ActionEvents.  
okButton.addActionListener(this);  
  
// Register dialogue window with the source (i.e. with itself)  
// in order to receive WindowEvents.  
addWindowListener(this);  
  
// Show the GUI.  
pack();  
setVisible(true);  
}  
  
// Determine whether a string contains a valid integer. (2)  
private boolean isValidInteger(String tf) {  
    try {  
        Integer.parseInt(tf);  
    } catch (NumberFormatException ex) {  
        return false;  
    }  
    return true;  
}  
  
// Terminate the dialogue window. (3)  
private void removeDialogWindow() {  
    setVisible(false);  
    dispose();  
}  
  
// Methods from the ActionListener interface. (4)  
public void actionPerformed(ActionEvent event) {  
    String integerString = inputField.getText();  
    if (isValidInteger(integerString)) {  
        app.showInteger(integerString);  
        removeDialogWindow();  
    }  
}  
  
// Methods from the WindowListener interface.  
public void windowClosing(WindowEvent event) {  
    removeDialogWindow();  
}  
  
public void windowClosed(WindowEvent event) { }
```



```

public void windowOpened(WindowEvent event)      { }
public void windowIconified(WindowEvent event)   { }
public void windowDeiconified(WindowEvent event) { }
public void windowActivated(WindowEvent event)   { }
public void windowDeactivated(WindowEvent event) { }
}

```

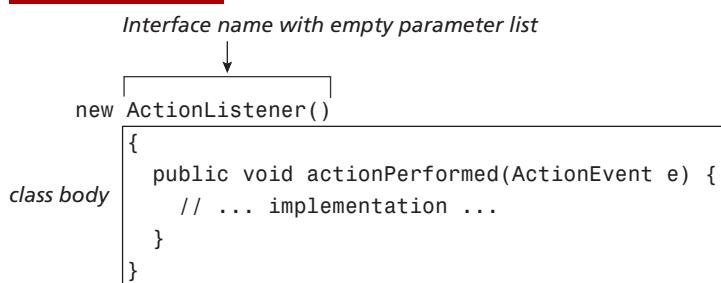
20.7 Anonymous classes as listeners

A *listener* executes actions in response to an event. Very often a listener requires access to information from other objects in the program. A listener must also be registered with a source. External listeners in Program 20.7, Program 20.8 and Program 20.9 had a reference to the main window, where they were also registered with the source. It would be much better to define these listeners in the main window, because then they can both register themselves with the source *and* have access to the information they need in order to carry out actions in response to events. Anonymous classes in Java can be used to implement such listeners.

Creating and registering of anonymous listener objects

Usually an object is created with the help of the new operator together with a constructor call. The constructor call specifies the class whose object we are interested in creating. An anonymous class lets us combine both object creation and class declaration. Figure 20.27 shows how we can define a listener object that can be used to handle events of the type ActionEvent. We require a listener object that implements the listener interface ActionListener. This interface comprises the method actionPerformed(). The expression in Figure 20.27 does exactly that. The new operator creates an object based on a class body that implements the interface specified in the expression. We see that the class used here does not have a name, only the class body is declared. That is why such a class is called an anonymous class.

FIGURE 20.27 Anonymous class implements an interface



The expression in Figure 20.27 returns the reference value of an object that implements the listener interface ActionListener. We can assign this reference value to a variable of the type ActionListener:



```
ActionListener listener = new ActionListener() { /* see Figure 20.27 */ };
```

We can now register this listener with a source:

```
colorButton.addActionListener(listener);
```

where the reference `colorButton` refers to a `JButton` object. When the user clicks on this button, the method `actionPerformed()` in the listener object will be executed.

Using anonymous listener objects

Program 20.14 defines two anonymous classes at (2) and (5). The frame changes color when the user clicks on the colour button, and the program terminates when the user clicks on the window close button of the frame. Since listener objects are used in one place only in this program, we have defined anonymous classes as arguments in the call to the `addActionListener()` method at (2) and (3).

Anonymous classes are always nested in other classes. The anonymous classes in Program 20.14 are nested in the class `SimpleWindow5`. In our example, the anonymous classes can refer to other members that are defined in the enclosing class `SimpleWindow5`. We have utilised this fact in the implementation of the `actionPerformed()` method. The `actionPerformed()` method calls the `changeColor()` method at (4). This method is declared in the enclosing class `SimpleWindow5`, and not in the anonymous class. This is also the case with the call to the `terminateProgram()` method at (7). It is not necessary to pass any references to an anonymous class in order for it to call methods or access fields in the enclosing class.

Program 20.14 shows the two forms of anonymous class declarations. The class at (2) implements the interface `ActionListener`, and must provide an implementation of the `actionPerformed()` method. The class at (5) extends the superclass `WindowAdapter` and overrides the `WindowClosing()` method. The class `SimpleWindow5` in Program 20.14 is *not* a listener, but delegates this responsibility to the anonymous classes.

PROGRAM 20.14 Implementing listener using an anonymous class (version 5)

```
import java.awt.Color;
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

import javax.swing.JButton;
import javax.swing.JFrame;

class SimpleWindow5 extends JFrame { // (1)
    // Components
    private JButton colorButton;

    // Other fields
```



```

private boolean colorToggle;

SimpleWindow5() {
    // Set suitable title for the frame.
    super("Simple window 5");

    // Create a button.
    colorButton = new JButton("Change colour");

    // Get the content pane.
    Container contentPane = getContentPane();

    // Create and set a layout manager, and add a button to the content pane.
    contentPane.setLayout(new FlowLayout(FlowLayout.CENTER));
    contentPane.add(colorButton);

    // Register a listener (that implements a contract) with the button
    // to receive ActionEvents.
    colorButton.addActionListener(new ActionListener() {                      // (2)
        public void actionPerformed(ActionEvent h) {                         // (3)
            changeColor();                                                 // (4)
        }
    });
}

// Register a listener (that extends a superclass) with the frame
// to receive WindowEvents.
addWindowListener(new WindowAdapter() {                                     // (5)
    public void windowClosing(WindowEvent h) {                            // (6)
        terminateProgram();                                              // (7)
    }
});

// Show the GUI.
pack();
setVisible(true);
}

// Change background colour for the content pane.
private void changeColor() {                                               // (8)
    if (colorToggle)
        getContentPane().setBackground(Color.cyan);
    else
        getContentPane().setBackground(Color.yellow);
    colorToggle = !colorToggle;
}

// Terminate the program.
private void terminateProgram() {                                         // (9)
    System.out.println("Terminating the program.");
    dispose();
}

```



```

        System.exit(0);
    }

public class SimpleWindowClient5 {
    public static void main(String[] args) {
        SimpleWindow5 gui = new SimpleWindow5();
    }
}

```

Some remarks on anonymous classes

An anonymous class is an expression that can be used where references can be used. For an anonymous class that extends a superclass, arguments can be specified. Then the superclass constructor corresponding to the argument list will be executed. For an anonymous class that implements an interface, the argument list must be empty, because an interface cannot have constructors. Since an anonymous class does not have a name, there are a few things we cannot do with an anonymous class:

- We cannot declare static members in an anonymous class.
- We cannot specify constructors for an anonymous class.
- An anonymous class in a method cannot refer to local variables in the method, unless they are declared with the keyword `final`.
- We cannot declare references that have the same type as the anonymous class. References of the anonymous class's supertypes must be used to manipulate objects of an anonymous class.

BEST PRACTICES

Use anonymous classes as listeners judiciously: such classes allow listeners to be defined and registered where events are handled, but can also make the code difficult to read.

20.8 Programming model for GUI-based applications

The programming model for GUI-based applications consists of the following steps:

- 1** Construction of the graphical user interface, i.e. the component hierarchy with layout.
- 2** Identification and registration of listeners with sources.
- 3** Implementation of listener interfaces, i.e actions to be executed when events occur.

The approach for implementing the programming model for GUI-based applications can be summarised as follows:



- Draw the component hierarchy that corresponds to the GUI.
Group the components in panels, depending on the spatial relations between the components.
Create the required components as the component hierarchy is being built.
- Choose a layout manager for each panel.
Call the method `setLayout(anotherLayoutManager)` to associate a layout manager with a panel.
- Add the components to each panel.
Call the method `add(guiComponent)` or `add(guiComponent, additionalInfo)` to place components in a panel.
Build the component hierarchy by nesting panels by calling the `add()` method.
- Choose a layout manager for the content pane of the root container.
Call the `getContentPane()` method to obtain the content pane of the root container.
Add components to the content pane of the root container with the `add()` method.
- Ensure that all the necessary events are handled properly.
Identify events, sources and listeners.
Register listeners with sources using the `addListener(listener)` method to handle an `XEvent`.
Listeners implement the relevant `XListener` interfaces and ensure proper handling of events.
Identify and handle an event that ensures proper termination of the application, usually a `WindowEvent`. This event is handled with the `windowClosing()` method in the `WindowListener` interface. Implementation of the `windowClosing()` method uses the call `System.exit(0)` to terminate the application.
- Make the root container visible by first packing the component hierarchy.
Call the `pack()` method to pack the components in the component hierarchy.
Call the `setVisible(true)` method to show the GUI.

BEST PRACTICES

Use the guidelines in Section 20.8 as a checklist to ensure that you have covered all the angles when you build your GUI.

20.9 GUI for the four-in-a-row game

In Chapter 14 we created a four-in-a-row game that was played at the terminal window. We will now develop a graphical user interface for this game, as shown in Figure 20.28. This is done by extending the game framework with 3 new classes:

- A new subclass `game.swing.GameSwingUI` of the class `game.AbstractUserInterface` that presents the game in a graphical user interface.

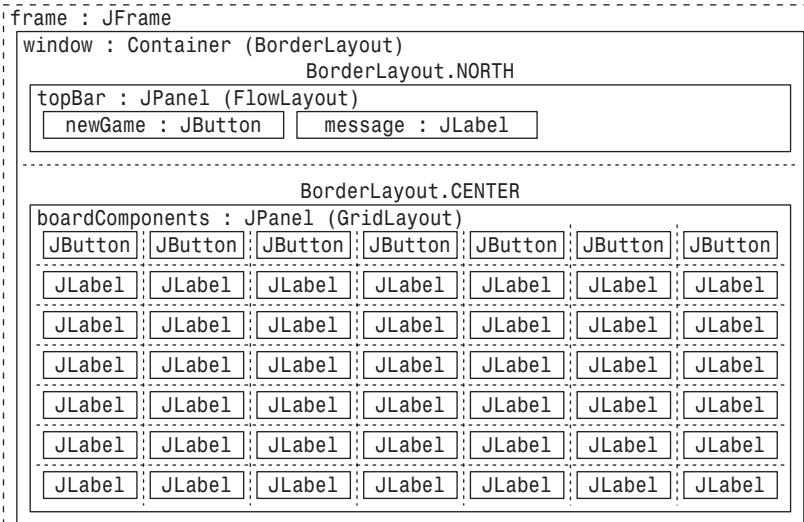


- A new subclass `game.swing.ComponentColorIdentity` of the class `game.ParticipantIdentity` that extends the player identity with a colour that can be applied to graphical components.
- A new class `game.swing.SwingPlayer` that implements the `game.IPlayer` interface, that will allow the user to choose a column via a graphical user interface.

FIGURE 20.28 GUI for the four-in-a-row game



FIGURE 20.29 Component hierarchy for the four-in-a-row game



The class `GameSwingUI` in Program 20.15 builds the component hierarchy shown in Figure 20.29. The class provides a graphical representation of the game by overriding the abstract methods `showGameBoard()` and `showMessage()` from the abstract class `game.AbstractUserInterface`. The row of `JButton` components at the top of the game board will drop pieces in the corresponding column when they are clicked. A player may only make a move in a column that is active.

**PROGRAM 20.15 GUI implementation for the four-in-a-row game**

```
package game.swing; // File: game/swing/GameSwingUI.java
import game.AbstractUserInterface;
import game.GameBoard;
import game.IPieceIdentity;

import java.awt.BorderLayout;
import java.awt.Component;
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class GameSwingUI extends AbstractUserInterface {
    Container boardComponents;
    Container topBar;
    JButton newGame;
    JLabel message;
    JFrame frame;

    GameSwingUI() {
        frame = new JFrame("Four-in-a-row");

        Container window = frame.getContentPane();
        window.setLayout(new BorderLayout());

        topBar = new JPanel();
        topBar.setLayout(new FlowLayout(FlowLayout.LEFT));
        message = new JLabel("New game started");
        newGame = new JButton("New Game");
        ActionListener newGameAction = new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                board.reset();
                runGameLoop();
            }
        };
        newGame.addActionListener(newGameAction);

        topBar.add(newGame);
        topBar.add(message);
    }
}
```



```
        window.add(topBar, BorderLayout.NORTH);

        boardComponents = new JPanel();
        window.add(boardComponents, BorderLayout.CENTER);
    }

    public void setBoard(GameBoard board) {
        super.setBoard(board);

        int columns = GameBoard.COLUMN_COUNT;
        int rows = GameBoard.ROW_COUNT;
        boardComponents.setLayout(new GridLayout(columns, rows + 1));

        for (int column = 0; column < columns; ++column) {
            boardComponents.add(makeDropButton(column));
        }

        for (int row = 0; row < GameBoard.ROW_COUNT; ++row) {
            for (int column = 0; column < columns; ++column) {
                JComponent cell = new JLabel("");
                cell.setBorder(BorderFactory.createEtchedBorder());
                cell.setOpaque(true);
                boardComponents.add(cell);
            }
        }
    }

    Component makeDropButton(final int column) {
        JButton dropButton = new JButton("Drop");
        ActionListener dropAction = new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                board.dropPieceDownColumn(board.nextPiece(), column);
            }
        };
        dropButton.addActionListener(dropAction);
        return dropButton;
    }

    protected void showGameBoard() {
        for (int column = 0; column < GameBoard.COLUMN_COUNT; ++column) {
            setDropPermissionForColumn(column, false);

            for (int row = 0; row < GameBoard.ROW_COUNT; ++row) {
                IPieceIdentity piece = board.pieceAt(column, row);
                ComponentColorIdentity identity = (ComponentColorIdentity) piece;
                Component component = componentForCell(column, row);
                identity.applyComponentColor(component);
            }
        }
    }
}
```



```

void setDropPermissionForColumn(int column, boolean dropIsAllowed) {
    Component button = boardComponents.getComponent(column);
    button.setEnabled(dropIsAllowed);
}

Component componentForCell(int column, int row) {
    return boardComponents.getComponent((GameBoard.ROW_COUNT - row) *
        GameBoard.COLUMN_COUNT + column);
}
protected void showMessage(String text) {
    message.setText(text);
}

public void startUserInterface() {
    runGameLoop();
    frame.pack();
    frame.setVisible(true);
}
}

```

In the text interface the pieces were represented as characters. In the graphical user interface we will represent the pieces by colouring the cells in the game board. We also wish to use more descriptive names like "Player 1" and "Machine Player" in messages reported to the user. The class `ComponentColorIdentity` in Program 20.16 therefore extends the state from the class `ParticipantIdentity` with a colour and a textual description. The method `applyComponentColor()` is used by the method `showGameBoard()` in the class `GameSwingUI` to colour each of the `JLabel` components that comprise the cells in the game board.

PROGRAM 20.16 Color coding GUI components to show player identity

```

package game.swing; // File: game/swing/ComponentColorIdentity.java

import game.IPlayer;
import game.ParticipantIdentity;

import java.awt.Color;
import java.awt.Component;

/**
 * The identity of a participant in a game that is played in a graphical
 * user interface where each cell of the game board is presented
 * as a java.awt.Component with a given colour.
 */
public class ComponentColorIdentity extends ParticipantIdentity {
    Color color;

```



```

public ComponentColorIdentity(IPlayer player, Color color,
                               String descriptiveName) {
    super(player, descriptiveName);
    this.color = color;
}

public void applyComponentColor(Component component) {
    component.setBackground(color);
}
}

```

Given the classes `GameSwingUI` and `ComponentColorIdentity`, it is now possible to create a game with a graphical representation of the game board. However, if we use the existing class `game.terminal.TerminalPlayer` in the game, the user will still be forced to type the move at the terminal window. The class `SwingPlayer` in Program 20.17 therefore defines a new implementation of the `IPlayer` interface. It allows the user to choose a column by clicking on one of the buttons at the top of the columns in the game board built by the `GameSwingUI` class. The new implementation of the `performMove()` method does not actually do the dropping of the piece, but tells the user interface which columns the user can drop a piece in, by calling the method `setDropPermissionForColumn()` that is defined in the `GameSwingUI` class. The listener objects that the `makeDropButton()` method in the `GameSwingUI` class associates with the buttons, will themselves execute the dropping of a piece when the user clicks on one of the buttons.

PROGRAM 20.17 GUI-based player move

```

package game.swing; // File: game/swing/SwingPlayer.java
import game.GameBoard;
import game.IPieceIdentity;
import game.IPlayer;

public class SwingPlayer implements IPlayer {
    GameSwingUI ui;

    SwingPlayer(GameSwingUI ui) {
        this.ui = ui;
    }

    public void performMove(IPieceIdentity piece, GameBoard board) {
        assert !board.isGameOver();
        for (int column = 0; column < GameBoard.COLUMN_COUNT; ++column) {
            ui.setDropPermissionForColumn(column,
                board.isValidColumnSelection(column));
        }
    }
}

```



Program 20.18 creates a game for two players that play against each other, based on the three new classes that extend the functionality of the framework in the `game` package. The program constructs the objects that make up the game in the same way as in Program 13.21 ([Please fix Xref.](#)), but uses the new user interface, the new piece identity and the new player implementation. As before, we can easily replace the player implementation that is used, in order to create a game where the player can play against the program, or where the program can play against itself.

PROGRAM 20.18 Game for two players playing against each other

```
package game.swing; // File: game/swing/EasySwingGame.java
import game.GameBoard;
import game.IPieceIdentity;

import java.awt.Color;

/**
 * An easy game where one player plays against another player.
 */
public class EasySwingGame {
    public static void main(String args[]) {
        GameSwingUI ui = new GameSwingUI();
        IPieceIdentity[] participants = new IPieceIdentity[] {
            new ComponentColorIdentity(new SwingPlayer(ui), Color.BLUE,
                "Player 1"),
            new ComponentColorIdentity(new SwingPlayer(ui), Color.YELLOW,
                "Player 2")
        };
        IPieceIdentity emptyCell = new ComponentColorIdentity(null, Color.GRAY,
            "");
        GameBoard board = new GameBoard(participants, emptyCell, ui);
        ui.startUserInterface();
    }
}
```

20.10 Review questions

- What is a container? Name three classes in the `javax.swing` package that are containers.

By a container we mean an object of any subclass of the abstract class `Container`. By a component we mean an object of any subclass of the abstract class `Component`. A container can have several components. A container is also a component, since the class `Container` is a subclass of the class `Component`. A container can therefore have both containers and components. The classes `JFrame`, `JDialog` and `JPanel` define containers, since they are subclasses of the class `Container`.



2. What is a component hierarchy?

A component hierarchy is a hierarchical structure that shows how a graphical user interface, consisting of containers and components, is constructed. The structure shows how containers are nested and which components are placed in them.

3. What is the content pane of a root container? Why is it important?

A root container forms the starting point for building a GUI. Each root container has a container associated with it, called the content pane. The content pane contains the component hierarchy that defines the GUI.

4. Finish writing the following statements:

A _____ is used to arrange components in a container.

The method _____ is used to associate a _____ with a container.

The method _____ is used to place components in a container.

The method _____ can be used to make a component visible or invisible on the screen.

A *layout manager* is used to arrange components in a container.

The method `setLayout()` is used to associate a *layout manager* with a container.

The method `add()` is used to place components in a container.

The method `setVisible()` can be used to make a component visible or invisible on the screen.

5. Which statements are true about containers?

- a** A `JPanel` is a container.
- b** A `JButton` is a container.
- c** The content pane of a root container is a container.
- d** A `JFrame` is a root container.
- e** A container can have at the most one layout manager at any given time.
- f** A GUI-based application must have a root container.
- g** A root container cannot be placed in another container.

All statements are true.

The class `JButton` is a subclass of the class `Container`, and is also a container. Whether there is any point in placing components in it is a different issue.

6. Which statements are true about layout managers?

- a** A layout manager arranges the size and the placement of components in a container.
- b** A `FlowLayout` manager is the default layout manager for a `JPanel`.



- c A BorderLayout manager is the default layout manager for a JPanel.
- d In order to use a BorderLayout manager, a component must be placed in each region (NORTH, SOUTH, WEST, EAST, CENTER) that the manager defines.
- e The sequence in which the components are placed in each region specified by a BorderLayout manager, is irrelevant.
- f A FlowLayout manager can stretch components placed in a container.

(a), (b), (e)

It is not necessary that all regions defined by the BorderLayout manager, should have a component. A FlowLayout manager cannot stretch components placed in a container.

7. Which statements are true about the GridLayout manager?

- a The expression new GridLayout(3, 4) will create a grid that has 3 rows and 4 columns.
- b The expression new GridLayout(3, 4) will create a grid that has 4 rows and 3 columns.
- c The number of components that can be placed in a container using a grid can be more or less than the size of the grid.
- d The sequence in which the components are placed determines the location of a component in the grid.
- e The normal behaviour is to fill the cells row-wise from left to right, and top to bottom.

(a), (c), (d), (e)

In the constructor `GridLayout(int n, int m)`, the first parameter specifies the number of rows and the second the number of columns.

8. Given the following program:

```

import java.awt.Container;
import java.awt.GridLayout;

import javax.swing.JButton;
import javax.swing.JFrame;

class GridLayoutPE extends JFrame {

    GridLayoutPE(int x, int y, int n) {
        super("GridLayoutPE");

        Container contentPane = getContentPane();
        contentPane.setLayout(new GridLayout(x, y));

        for (int i = 0; i < n; ++i)
            contentPane.add(new JButton("" + i));

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```



```

        pack();
        setVisible(true);
    }
}
public class GridLayoutPEClient {
    public static void main(String[] args) {
        for (int i = 0; i <= 3; i++)
            new GridLayoutPE(2, 1, i);
    }
}

```

- a** How many windows are created?
b How many buttons are there in each window?
c What happens when you click on the close button of one of the windows?

(a) Four windows are created:



(b) 0, 1, 2 and 3 buttons in the four windows, respectively. Note the numbering of the buttons.

(c) If you click on the close button of one of the windows, all windows are closed and the program is terminated. The following code line is responsible for this behaviour:

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

- 9.** Which statements are true about designing component hierarchies?
- a** A JPanel cannot be placed in another JPanel.
b A JFrame can be placed in a JPanel.
c Components can be placed in a JFrame.
d A JFrame has a content pane.
e A JPanel has a content pane.
f The whole component hierarchy can be placed in the content pane of a root container.
g All containers in a component hierarchy must use the same layout manager.
- (d), (f)

A container can contain other components. A JPanel is a container, since the class JPanel is a subclass of the class Container. A JPanel is also a component, since the class JPanel is a subclass of the class Component. Therefore a JPanel can be placed in another JPanel. A JFrame defines a top-level window that cannot be placed in other containers (for example, in a JPanel). Components cannot be placed directly in a JFrame, only its content pane. A JPanel is not a top-level window, and therefore has



no content pane. Containers in a component hierarchy can use different layout managers.

10. What is the difference between a source and a listener when it comes to events?

A source generates events in response to user actions, and informs listeners about these events. A listener can receive events from a source if it registers itself with the source. Registration requires that the listener implements the right listener interface. A listener can register itself with several sources, and a source can inform several listeners about events.

11. What is the purpose of a listener interface?

A listener interface defines methods that a source can call in a listener to inform about events. The compiler checks that only listeners that implement the right listener interface can be registered with a source. At runtime a source is guaranteed that the listener implements the methods in the listener interface.

12. Which statements are true about event handling?

- a** A listener can register itself with one source only.
- b** A listener can be informed about one type of events only.
- c** A source can generate several types of events.
- d** A source must have at least one listener registered.

(c)

A listener can register itself with several sources, if it implements the right listener interfaces. This means that it can be informed about different types of events, depending on which listener interfaces it implements. A source need not have a listener registered. The source will function anyway.

13. The following questions concern events of the type `ActionEvent`.

- a** Which user action in a `JTextField` will generate an `ActionEvent`?
- b** Which user action on a `JButton` will generate an `ActionEvent`?
- c** Which method is provided by an `ActionEvent` source to register listeners?
- d** Which listener interface must a listener implement in order to receive an `ActionEvent`?
- e** Which method specifies the listener interface for events of the type `ActionEvent`?
 - (a) When the Enter key is typed in the text field.
 - (b) When the button is clicked with the mouse.
 - (c) The method `addActionListener()` registers an `ActionEvent` listener with an `ActionEvent` source.
 - (d) The interface `ActionListener` must be implemented by an `ActionEvent` listener.
 - (e) The method `actionPerformed()` is specified in the interface `ActionListener`.



- 14.** Given the following program:

```
import javax.swing.JButton;
import javax.swing.JPanel;

public class TestPanelPE {
    public static void main(String[] args) {
        JPanel panel = new JPanel();
        JButton button = new JButton("Kawabanga!");
        panel.add(button);
        panel.setVisible(true);
    }
}
```

What is the result of compiling and running the program? Choose the one correct answer.

- a** The program will not compile, because a `JPanel` object is not a top-level window.
- b** The program will compile, but at runtime only the panel is visible and not the button, since we have not defined any layout manager for components in the panel.
- c** The program will compile, but at runtime no GUI is shown, and the program will terminate.

(c)

The program will compile. Since a `JPanel` object is not a top-level window, no GUI is shown. A `JPanel` object has a `FlowLayout` manager as default layout manager, if no layout manager is specified.

- 15.** Given the following program:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;

class GUICounterV1 extends JFrame {

    private JButton buttonPlus;
    private JButton buttonMinus;
    private JTextField integerField;

    private int counter;

    GUICounterV1() {
        super("GUICounterV1");
```



```

integerField = new JTextField("0", 10);
integerField.setEditable(false);
buttonPlus = new JButton("Plus");
buttonMinus = new JButton("Minus");

JPanel panel = new JPanel();
panel.add(integerField);
panel.add(buttonPlus);
panel.add(buttonMinus);
getContentPane().add(panel); // (1)

ActionListener listener = new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        if (buttonPlus == event.getSource()) // (2)
            ++counter;
        else
            --counter;
        integerField.setText("" + counter);
    }
};
buttonPlus.addActionListener(listener); // (3)
buttonMinus.addActionListener(listener);

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
pack();
setVisible(true);
}
}

```

- a** What does the program do?
 - b** How many sources for ActionEvent are there in the program?
 - c** How many listeners for ActionEvent are there in the program?
 - d** What does the call `getContentPane()` at (1) do:
- `getContentPane().add(panel);`
- e** What will the call `event.getSource()` at (2) return?
 - f** What does the following call at (3) do:
- `buttonPlus.addActionListener(listener);`
- g** Write a new version of the program (without anonymous classes) so that the main window is the only listener for all ActionEvents.



- (a) The program allows the user to increment and decrement a counter with the help of buttons:



- (b) There are 2 sources for `ActionEvent`, both are `JButton` objects.
- (c) There is one listener for `ActionEvent`, and the listener is implemented using an anonymous class.
- (d) The method `getContentPane()` is called on the current object, which is an object of the `GUICounterV2` class. This class is a subclass of the class `JFrame`. An object of the class `GUICounterV2` inherits this method and the content pane from the class `JFrame`. The method returns the reference value of this content pane.
- (e) The call `event.getSource()` will return the reference value of the source that generated the event, referenced by the variable `event`.
- (f) The call `buttonPlus.addActionListener(listener)` registers the listener referenced by the variable `listener` with the button referenced by the variable `buttonPlus`.
- 16.** Given the following program:

```

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;

public class TestFramePE {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        JButton button = new JButton("Kawabanga!");
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                System.exit(0);
            }
        });
        frame.getContentPane().add(button); // (1)
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // (2)
        frame.pack();
        frame.setVisible(true);
    }
}

```

- a** Will the program compile, run and terminate normally?



- b** If we replace (1) with the following statement:

```
frame.add(button);
```

will the program compile, run and terminate normally? (Please fix indentation.)

- c** If we delete (2), will the program compile, run and terminate normally?

(a) The program will compile and run. It will terminate normally when either the button or window close button is clicked.

(b) The program will not compile. The compiler checks that the content pane of the frame is used for adding components.

(c) The program will compile and run. It will terminate normally when the button is clicked. If the window close button is clicked, the window will disappear, but the GUI thread will continue to run. One way to check this is to see the processes running on the machine.

- 17.** Which statements are true about dialogue windows?

- a** The class `JOptionPane` creates modal dialogue windows, while the class `JDialog` can create both modal and non-modal dialogue windows.
- b** A non-modal dialogue window must be closed before the program can continue.
- c** A modal dialogue window must be closed before the program can continue.

(a), (c)

A modal dialogue window is usually used when the program needs information from the user before the program can continue, or for simplifying the use of the program so that unnecessary windows do not cause confusion to the user.

- 18.** Which statements are true about anonymous classes?

- a** An anonymous class can be used to create one object only.
- b** In an anonymous class declaration that implements an interface, we cannot specify parameters in the parameter list after the interface name.
- c** In an anonymous class declaration that extends a superclass, we can specify parameters in the parameter list after the superclass name.
- d** In an anonymous class declaration we can refer to members in the enclosing class, even if they are private.

(b), (c), (d)

Each time an expression with an anonymous class declaration is executed, a new object of the anonymous class is created.

(d) is true only if the anonymous class declaration occurs either in a constructor- or a method body.

- 19.** Given the following program:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```



```

import javax.swing.JButton;
import javax.swing.JFrame;

class AnonPE extends JFrame {

    public int i = 10;
    private int j = 20;

    public void methodA(int m) {
        System.out.println(m);
    }

    private void methodB(int n) {
        System.out.println(n);
    }

    AnonPE() {
        JButton button = new JButton("Click on me!");
        getContentPane().add(button);
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                methodA(j); // (1)
                methodB(i); // (2)
            }
        });
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }
}

public class AnonPEClient {
    public static void main(String[] args) {
        new AnonPE();
    }
}

```

Which statements are true about the program?

- a** The program will not compile, because (1) refers to a private field *j* in the enclosing class *AnonPE*.
- b** The program will not compile, because the *methodB()* is a private method in the enclosing class *AnonPE* and cannot be called in the anonymous class.
- c** The program will compile and, when run, will show the following GUI:





- d** The program will compile and, when run, will show the GUI in (c). The program will terminate after the button is clicked, with the following output in the terminal window:

20
10

- e** The program will compile and, when run, will show the GUI in (c). The program will print the following output in the terminal window each time the button is clicked:

20
10

- f** The program will compile and run, but will only terminate if the window close button is clicked.

(c), (e), (f)

In the given program, the anonymous class can refer to all members in the enclosing class. Clicking on the button does not terminate the program.

- 20.** The GUI below is for a program that reads text from a text field and converts it to uppercase when the user clicks on the "Change to uppercase" button. The result of the conversion is shown in the window. Characters in the text field are deleted each time this conversion is done.



Finish writing the program given below for the GUI shown above.

```

import java._____.*; // (1)
import java._____.event.*; // (2)
import javax._____.*; // (3)

class GUIPE extends _____ { // (4)

    private _____ label; // (5)
    private _____ inputField; // (6)
    private _____ button; // (7)

    public GUIPE() {
        super("GUI exercise");

        Container contentPane = _____(); // (8)

        label = new _____("", _____.CENTER); // (9)
        contentPane._____(label, BorderLayout.CENTER); // (10)
    }
}

```



```

_____ panel = new _____(); // (11)
inputField = new _____(20); // (12)
panel._____ (inputField); // (13)

button = new JButton("Change to uppercase");
button._____ ( // (14)
    new _____() { // (15)
        public void _____(____ event) { // (16)
            label.setText(inputField.
                getText().toUpperCase());
            inputField._____ (""); // (17)
        }
    }
);
panel._____ (button); // (18)
contentPane._____ (panel, BorderLayout.SOUTH); // (19)

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

pack();
_____ (true); // (20)
}
}

public class GUIPEClient {
    public static void main(String[] args) {
        GUIPE gui = new GUIPE();
    }
}

```

The complete source code is given below.

```

import java.awt.*; // (1)
import java.awt.event.*; // (2)
import javax.swing.*; // (3)

class GUIPE extends JFrame { // (4)

    private JLabel label; // (5)
    private JTextField inputField; // (6)
    private JButton button; // (7)

    public GUIPE() {
        super("GUI excercise");

        Container contentPane = getContentPane(); // (8)

        label = new JLabel("", JLabel.CENTER); // (9)
        contentPane.add(label, BorderLayout.CENTER); // (10)

        JPanel panel = new JPanel(); // (11)
        inputField = new JTextField(20); // (12)
    }
}

```



```

        panel.add(inputField); // (13)

        button = new JButton("Change to uppercase");
        button.addActionListener( // (14)
            new ActionListener() { // (15)
                public void actionPerformed(ActionEvent event) { // (16)
                    label.setText(inputField.getText().toUpperCase());
                    inputField.setText(""); // (17)
                }
            });
        panel.add(button); // (18)
        contentPane.add(panel, BorderLayout.SOUTH); // (19)

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        pack();
        setVisible(true); // (20)
    }
}

```

20.1 Programming exercises

1. Create a new version of Program 20.1 (`GUIFrame`), so that the information is presented using the class `JOptionPane`.
2. Add an OK button and a Cancel button to the main window in Program 20.6 (`GridLayoutDemoV2`). The program reads and checks the coordinates (`int` values) when the OK button is clicked. Determine the quadrant in which a point with these coordinates is located. Use the class `JOptionPane` to give suitable messages to the user.
The quadrants are defined depending on the values of the coordinates:
 - Quadrant 1: $x \geq 0, y \geq 0$
 - Quadrant 2: $x < 0, y \geq 0$
 - Quadrant 3: $x < 0, y < 0$
 - Quadrant 4: $x \geq 0, y < 0$
3. Write a new version of Program 20.11 (`EchoWithEvents`), so that it uses an anonymous class to implement the listener.
4. Write a new version (without anonymous classes) of the program in Question 20.15, so that the main window is the only listener for all `ActionEvents`.
5. Write a new version (without anonymous classes) of the program in Question 20.15, so that a new class defines the only external listener for all `ActionEvents`.
6. Write a currency converter that converts between two currencies. Given the exchange rate (for example, NOK 12.00 == GBP 1) and one of the currencies (for



example, NOK 48), the currency converter calculates the amount in the other currency (which is GBP 3 in this case).

7. Write a simple calculator for the common arithmetic operations (+, -, *, /). The user specifies two values and clicks on an operator button to execute the designated operator on the specified values. The result is shown in a text field in the same window.
8. Write a graphical version of the class `TextUserInterface` (Program 8.3 on page 213) to read and write information about an employee. The new class `GraphicalInterface` uses the class `GUIDialog` from Program E.3. Do testing on a modified version of the class `NewCompany` (Program 8.7 on page 222). Use the version of the class `PersonnelRegister` from Exercise 15.10 on page 508.
9. Write a new version of Exercise 13.3 on page 387, so that the class `GUIDialog` from Program E.3 is used to read information about an employee from the terminal window.
10. Extend Program 20.3 so that when the user clicks on the Order button, information about which pizza size and which pizza toppings were chosen is shown to the user for confirmation.
11. Write a program for the Hot and Cold game where a player tries to guess a number between 0 and 1000. The program selects an integer (see Section 6.8 on page 149 [Please fix Xref format.] how this can be done) and informs the user that an integer has been chosen. The background colour of a text field used for reading guesses, can be used to give feedback to the user about how good the guess was. The background colour is changed to red to indicate that the trail is getting hotter, and to blue to indicate that the trail is getting colder. The trail gets *hotter* if the current guess is closer to the answer than the previous guess. Otherwise the trail gets *colder*.

Design a suitable GUI for the game. The program should report how many tries it took to guess the correct answer. The user should be able to start a new game at any time.

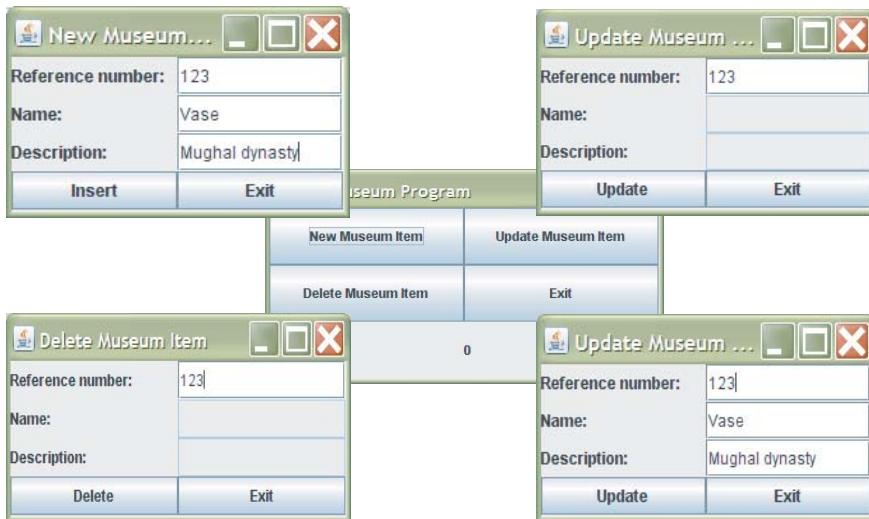
12. Modify Program 20.18 on page 697 so that the program asks which type of player will participate in the game. Allow the user to choose between `TerminalPlayer`, `RandomPlayer` and `SwingPlayer` for each of the players. The game should start when both players have been chosen.
13. Implement a graphical user interface for museum administration (see Exercise 19.12 on page 632 and Exercise 11.18 on page 324). Instead of a text-based interface, the program now provides a graphical user interface, where all interaction with the user takes place through windows and dialogue boxes. The graphical interface should provide the same functionality as the text-based interface did.

The program should have a main window where it is possible to choose the following operations:

- add a new item
- change the information about an item
- remove an item



- terminate the program



When the user chooses an operation in the main window, a new window should come up that is customised for executing that operation. Through these dialogue windows the user is informed about registered information and can provide new information via a form. The forms that are used for adding an item, changing information about and removing items are very alike (see the figures above). Use inheritance in a suitable way to extract the similarities of these forms into a common superclass.





P A R T F O U R

Appendices

Answers to review questions

A.1 Getting started

1. Program 1.1 has seven comments.
2. A computer has a *central processing unit (CPU)* that executes low-level platform-specific instructions and makes the computer work.
3. *Source code* is a high-level description of the tasks the computer should perform, which are written in a high-level *language* and stored in text files.
4. (a), (c), (d).
Compilers and virtual machines are off-the-shelf programs.
5. We specify the set of common properties we want a group of objects to share by defining a *class*.
6. All Java programs have a *method* called *main*, where the execution of the program starts.
7. (b).
8. Program 1.1 has a class called *SimpleProgram* and a method called *main*.
9. The command “javac TestProgram.java” can be executed on the command line to compile a source code file called *TestProgram.java*.
10. The command “java -ea TestProgram” can be executed on the command line to run a Java program consisting of a primary class called *TestProgram*.
11. (a) Source code. The source code can be translated to other forms.
12. (c) Only *Dog.java* is valid.

A.2 Basic programming elements

1. The code prints:

```
10+10 is 20  
10+10 is 1010  
40
```

2. A variable identifies a memory location that can hold a value of a specific data type. A local variable is created during the execution of a method, and will only exist in memory while the method is being executed.

3. `int numberOfPoints = 35;`

A value written directly in the source code is called a literal.

4. A constant is a variable whose value cannot be changed after initialization. We define a constant using the modifier `final`, as shown in the following line of code:

```
final int NUMBER_OF_POINTS = 35;
```

5. (a) `minimum-Price`: The name is invalid because it contains a hyphen, (-).

(b) `minimumPrice`: The name is valid because all letters are allowed in variable names. The name conveys its purpose.

(c) `XYZ`: The name is valid, but it does not convey its purpose.

(d) `xCoordinate`: This is a valid name, as it only contains letters. The name conveys what the variable stores: an *x*-coordinate to a point or another geometric primitive.

(e) `y2k`: Both letters and digits are allowed in variable names. The name is valid, but not very meaningful.

(f) `isDone`: The name is valid and easy to read. It indicates that the variable will hold a Boolean value that conveys whether a specific condition is satisfied. The name may be too general.

(g) `numberOfDaysInALeapYear`: This is a very long variable name, and hard to read, as it contains many words, but it is a valid name.

(h) `JDK_1_6_0`: This is also a valid name, since underscore (_) is allowed in a variable name. On the other hand, the purpose of the name is not clear. It may also be mistaken for a constant, as we have defined its name in all uppercase letters.

6. A data type is defined by a set of valid values and the operations that can be performed on these values.

A primitive data type is a data type defined by the programming language, which we can use directly by specifying its type name in the source code. Many languages offer primitive data types, to provide support for integers and floating-point values, for example.



- 7.** (a) An arithmetic expression can consist of *operators* and operands.
(b) Multiplication (*) requires *two* operands, and is thus a *binary operator*.
(c) The operator - in the expression -4 is a *unary* operator, while the operator - in the expression 5 - 4 is a *binary* operator.
- 8.** The operands in an expression in Java are always evaluated from *left* to *right*.
If two operators with different *precedence* are next to each other in an expression, the operator with the *highest precedence* will be evaluated first.
- 9.** If two operators with the same precedence are next to each other in a expression, *associativity* rules are used to determine which operator will be evaluated first.
The operator - (unary minus) is left-associative and groups from *left* to *right*, while the - (subtraction) is right-associative and groups from *right* to *left*.
- 10.** (a) $3 + 2 - 1 \rightarrow (3 + 2) - 1 \rightarrow 5 - 1 \rightarrow 4$. Addition and subtraction have the same precedence, and are left-associative.
(b) $2 + 6 * 7 \rightarrow 2 + (6 * 7) \rightarrow 2 + 42 \rightarrow 44$. Multiplication has higher precedence than addition, and is thus evaluated first. Both operators are left-associative.
(c) $-5 + 7 - -6 \rightarrow (-5) + 7 - (-6) \rightarrow ((-5) + -7) - (-6) \rightarrow 2 - (-6) \rightarrow 8$. Unary operators have higher precedence than binary, and are right-associative.
(d) $2 + 4 / 5 \rightarrow 2 + (4 / 5) \rightarrow 2 + 0 \rightarrow 2$. Division has higher precedence than addition. Integer division where the denominator is less than the numerator yields 0 as the result.
(e) $5 * 2 - -3 * 4 \rightarrow 5 * 2 - (-3) * 4 \rightarrow (5 * 2) - (-3) * 4 \rightarrow 10 - ((-3) * 4) \rightarrow 10 - (-12) \rightarrow 2$. Unary operators have higher precedence than binary. Multiplication has higher precedence than subtraction. Both multiplication and subtraction are left-associative.
(f) $10 / 0 \rightarrow$ An exception will be thrown and the program will terminate.
(g) $2 + 4.0 / 5 \rightarrow 2 + (4.0 / 5.0) \rightarrow 2 + 0.8 \rightarrow 2.0 + 0.8 \rightarrow 2.8$. Division has higher precedence than addition. When one of the operands of a division operation is a floating-point value, the other operand is automatically converted to a floating-point value before the division is performed.
(h) $10 / 0.0 \rightarrow 10.0 / 0.0 \rightarrow \text{Infinity}$. The integer 10 is converted to the floating-point value 10.0 because the denominator is a floating-point value. Floating-point division is performed, giving the result **Infinity**, since the denominator is 0.0.
(i) $4.0 / 0.0 \rightarrow \text{Infinity}$. Floating-point division where the denominator is 0.0 always gives the result **Infinity**.
(j) $2 * 4 \% 2 \rightarrow (2 * 4) \% 2 \rightarrow 8 \% 2 \rightarrow 0$. Multiplication and modulus operators have the same precedence. Both operators are left-associative.

A



- 11.** Possible format strings and parameter values to the `printf()` method are shown below. See also Appendix C.

a `System.out.printf("%+6d%n", 123456); // positive integer`
`System.out.printf("%+6d%n", -654321); // negative integer`

b `System.out.printf("%e%n", 123456789.3837);`
`System.out.printf("%g%n", 123456789.3837);`

c `System.out.printf("We are 100% motivated to learn Java!%n");`

d `System.out.printf("%08d%n", 1024);`

A.3 Program control flow

- 1.** A Boolean expression can only evaluate to one of two values, `true` or `false`. The value of such an expression can be assigned to a `boolean` variable.
- 2.** A relational operator compares the value of two operands. For example, the `==` operator compares its operands for equality.

A logical operator negates or combines the values of Boolean expression. For example, the `!` operator negates the value of a Boolean expression.

- 3.** (a) `2 < 3` is a Boolean expression. When evaluated, the resulting value (`true`) can be assigned to a Boolean variable.
- (b) `2 + 3 < 1 - - 5` is a Boolean expression. When evaluated, the resulting value (`true`) can be assigned to a Boolean variable.
- (c) This is not a valid Boolean expression. The string "`true`" cannot be converted to a Boolean value.
- (d) The Boolean literal `false` can be assigned to a Boolean variable.
- 4.** The expression is evaluated to `true && !(false) || !(true) == true`, then to `true && true || !(true) == true`, and finally to `true || (!(true) == true)`.
- 5.** Selection and loop statements offer two mechanisms for controlling the *flow of execution*. Such statements enable us to *select between* alternative actions, or to *repeat* an action a given number of times.
- 6.** Line (1): `false`.

Line (2): The expression `numMen = numWomen` is not a valid Boolean expression, since the operator used, `=`, is the assignment operator, and not the equality operator, `==`. If we replace the `=` operator with the `==` operator, the expression will be valid, and it will return the value `false`.

Line (3): This is a valid Boolean expression. Since the multiplication operator `*` has higher precedence than the relational operators, the value of the expression is `false`.

Line (4): This is also a valid Boolean expression, whose value is `false`.

A



- 7.** The Java operators `!` (negation), `&&` (conditional And), and `||` (conditional Or) expect operands of type `boolean`.
- 8.** (a) true. (b) false. (c) true. (d) true. (e) true. (f) false.
- 9.** Rewrite the Boolean expressions using De Morgan's laws:
- (a) $!(b1 \mid\mid b2) \rightarrow !b1 \And !b2$
 - (b) $!(b1 == b2 \And b2) \rightarrow !(b1 == b2) \mid\mid !b2 \rightarrow b1 != b2 \mid\mid !b2$
 - (c) $!(!b2 \mid\mid b1 == \text{true}) \rightarrow !!b2 \And !(b1 == \text{true}) \rightarrow b2 \And b1 == \text{false}$
 - (d) $(!b1 == b2 \mid\mid b2) \rightarrow !(b1 == b2 \And !b2)$
 - (e) $!(b1 == b2) \mid\mid !(b2 == b1) \rightarrow !(b1 == b2) \And (b2 == b1) \rightarrow !(b1 == b2) \And (b1 == b2) \rightarrow !(b1 == b2)$ (The second to last simplification follows from the fact that the order of operands to the `==` operator has no impact on the evaluation, i.e. `b1 == b2` is identical to `b2 == b1`. The last simplification follows from the fact that the two operands of the conditional And have the same value.)
- 10.** A selection statement and the remainder operator can be used to determine whether the variable `numPoints` has a value that is an even or an odd number:
- ```
if (numPoints % 2 == 0)
 System.out.println("Number of points (" + numPoints + ") is even.");
else
 System.out.println("Number of points (" + numPoints + ") is odd.");
```
- 11.** A repetition statement is often called a *loop*. The condition for repeating the *loop body* is specified as a *Boolean expression*.
- 12.** The difference between the two types of loops are as follows: in a `while` loop, the condition is tested *before* the loop body is executed. This means that if the condition is not satisfied when control flow enters the loop, the loop body will not be executed at all. In a `do-while` loop, the condition is not tested until *after* the loop body has been executed. This means that the loop body is executed at least once in a `do-while` loop.
- 13.** An `assert` statement specifies a *Boolean expression*, and (optionally) a *string*. The *string* is printed to the terminal window if the expression evaluates to `false`, and the execution is *aborted*.
- 14.** (a) False. Assertions are only executed if the program is run with the “`-ea`” (or the equivalent “`-enableassertions`”) option. (b) False. (c) True.

A



## A.4 Using objects

1. A class specifies the *properties* and *behaviour* of objects that can be created from the class.
2. Instance variables represent *properties*, and instance methods represent the *behaviour* of objects that can be created from a class.
3. (a), (c), (f), (g).
4. False. The code line is a declaration. It only declares a reference variable called `myCD` of reference type `CD`. No object is created by this declaration.
5. A declaration requires the type and the variable name, and object creation requires the use of the `new` operator with a constructor call:

```
CD cd1 = new CD();
CD cd2 = new CD();
CD cd3 = new CD();
CD cd4 = new CD();
CD cd5 = new CD();
```

6. Only one, and `cd1` and `cd2` are aliases to this object.
7. We can use the dot notation to call a method or access a field in an object:

```
reference.methodName();
value = reference.fieldName;
```

8. (a), (d).  
(b) and (e) are not valid, because the class name cannot be used to access instance members. (c) is not valid because a method call requires the specification of a parameter list even if there are no parameters.
9. '`z`' is a character literal, while '`"z"`' is a string literal. A character literal is represented by its Unicode value, whereas a string is represented by a `String` object that stores the characters in the string.
10. If `(str1==str2)` is true, it means that `str1` and `str2` are aliases and refer to the same `String` object. The method call `str1.equals(str2)` will always return `true`, because the `String` object is compared with itself. If two `String` objects have the same state, i.e. the character sequence in both strings are identical, the expression `(str1==str2)` will return `true`. But the reference values in `str1` and `str2` are different, because the references refer to two different `String` objects.



- 11.** The statements will print:

2002  
2050  
20002

(1) calculates  $(2000 + 2)$ , (2) calculates  $(2000 + 50)$ , where the character '2' has the code number 50. (3) concatenates the strings "2000" and "2".

- 12.** (b).

- 13.** (f).

In (f), the value 12 is auto-boxed into an `Integer` object. However, the reference value of an `Integer` object cannot be assigned to a reference of type `Double`.

## A.5 More on control structures

- 1.** We can use any of the following statements:

```
i = i + 1;
i += 1;
i++;
++i;
```

A

- 2.** Output:

```
4
6
6
4
4
```



- 3.** One would choose a `for(;;)` loop, since it is appropriate for implementing a counter-controlled loop when the number of iterations is known beforehand.

- 4.** All parts (initialization, loop condition, updating) in the `for(;;)` loop header can be omitted. No loop condition implies that the loop condition is true. No initialization or updating corresponds to the empty statement, which does nothing.

- 5.** (a), (b), (d).

The loop body is never executed if the loop condition is `false` when control enters the loop for the first time.

- 6.** Here is one possible `for(;;)` loop that is equivalent to the code in the question:

```
int sum = 0;
for (int i = 10; i >= 5; i -= 2) {
 sum += i;
}
```

Here we will also mention that the initialization part can be a list of declarations, separated by a comma. The loop above can also be written as follows:

```
for (int i = 10, sum = 0; i >= 5; i -= 2) {
 sum += i;
}
```

The `for(;;)` loop is executed three times.

- 7.** In (a) the loop is executed five times, and the variable `i` has the value 10 after the loop has completed. In (b) the loop is executed four times, and the variable `i` has the value 12 on termination of the loop, but the variable is not accessible outside the loop.

- 8.** (b).

In (a) the loop executes three times but it does not print anything because an empty statement `()` comprises its loop body. The string "Move it!" is printed only once after the loop has completed.

- 9.** (b), (c).

- 10.** (b).

A `default` label is optional in a `switch` statement. A string literal cannot be used as a case label. A `break` statement is not part of a `switch` statement. Boolean values and floating-point values cannot be specified as case labels.

- 11.** Rewriting the code using a `switch` statement:

```
switch(i) {
 case 10: case 20:
 System.out.println("10 or 20");
 break;
 case 15:
 System.out.println("15");
 break;
 default:
 System.out.println("Not valid");
}
```

- 12.** (d), (f), (g).

Boolean `switch` expressions and case labels are not permitted, as in (d). The value of the `switch` expression cannot be `double`, as in (f). The `switch` expression cannot be `boolean`, as in (g). The case label values must be unique, which is not the case in (g).



## A.6 Arrays

1. An array has a public field called `length`, whose value is the number of elements stored in the array. Given that the reference `row` refers to an array, the number of elements in the array is returned by the expression `row.length`.

2. The `[]` notation can be used to declare an array reference:

```
String[] arrayRef;
```

The `[]` notation can be used to create an array:

```
arrayRef = new String[10]; // Array can store 10 elements.
```

The `[]` notation can be used to access an array element:

```
arrayRef[2] = "Name: " + arrayRef[0];
```

3. (f).

(a) declares `age` as a reference to an array of integers; no array is created. (b) is missing the `new` operator for creating an array. (c) declares `age` as a variable of type `int`, and not as a reference to an array. Both (d) and (e) are missing the correct specification of the length of the array. Only (f) is correct.

4. (b), (d).

There is no method called `length()` for arrays. The first element in the array is given by `arrayRef[0]`.

5. (a), (d), (e), (f).

Block notation, `{}`, is used to create and initialize an array in a declaration. It cannot be used in an assignment statement as in (b) and (c).

6. (b).

The index value must be a non-negative integer value that satisfies the relation  $0 \leq \text{index value} < \text{array length}$ .

7. `twoDimArrayName` has the type `String[][]`, i.e. a reference to an array of array of `String` objects, which is a two-dimensional array of `String` objects.

`twoDimArrayName[1]` has the type `String[]`, i.e. a reference to an array of `String` objects.

`twoDimArrayName[1][2]` has the type `String`, i.e. a reference to a `String` object.

8. (a), (d), (e), (f)

Block notation, `{}`, is used to create and initialize an array in a declaration. It cannot be used as an assignment statement, as in (b) and (c). The delimiter in block notation is a comma `,` and not a semicolon `;`, as in (g).

A



**9.** (a), (b)

The loop body is not executed if the collection is empty, and changing the value of the element variable does not change the values in the collection.

**10.** (b), (c), (d).

The element variable must be declared in the header of the `for(:)` loop. Changing the value of the element variable in the loop body is allowed, but that does not change any values in the collection.

**11.** (d).

The value stored in each element is its index value plus 1.

**12.** (c).

The values in the elements are not affected by the first loop. They remain initialized to the default value for `int` type (0).

**13.** (a), (b), (c).

In (d) the two hotels will share the same floors and rooms. In (b) enough storage is allocated for the rooms. In all cases we have to be consistent with what the different indices represent. In (a), (b) and (c) all the rooms will be initialized to the default value for type `int` (0).

A



**14.** (c).

Arrays have a public field named `length`, and `String` objects have a method named `length`. Not the other way around, as in the code.

**15.** (b), (d), (e).

The method call `nextInt(n)` will always return an integer in the interval  $[0, n-1]$ , inclusive, where integer  $n$  is the upper bound on the random number returned.

## A.7 Defining classes

**1.** Static variables in the class `Counter`:

`MAX_VALUE, description`

Static methods in the class `Counter`:

`getDescription`

Instance variables in the class `Counter`:

`value`

Instance methods in the class `Counter`:

`getCounter, setCounter, incrementCounter, decrementCounter, resetCounter`

- 2.** *Instance members* belong to objects, while *static members* belong to the class.
- 3.** Values of all the field variables in an object comprise its *state*.
- 4.** The initial state of an object is the state immediately after it is created using the `new` operator.
- 5.** The default value of any reference variable is always the `null` literal.
- 6.** Field variables in the class `RectangleV2`, which are initialized with default values:

`length, breadth`

Field variable in the class `RectangleV2`, which is initialized with an initial value:

`area`

When an object of the class `RectangleV2` is created, the fields `length` and `breadth` are initialized to `0.0`, and afterwards the field `area` is initialized to `0.0` (`length * breadth`).

- 7.** (a).
- 8.** Class names in the class `ClientA`:

`Counter, System`

Local variables in the method `main()`:

`numOfCars, carCounter`

Names of methods called in the method `main()`:

`incrementCounter, println, getCounter`

Formal parameters to the method `main()`:

`args`

An actual parameter is specified in the call to the `println()` method. The value of this actual parameter is the value returned by the method call `carCounter.getCounter()`.

- 9.** Printout:

```
k + d is equal to 2
k + d is equal to 3
k + d is equal to 12
k + d = 12
```

Note that the formal parameter `d` in method `doIt()` shadows the field `d`. The variable `k` at (1) refers to the local variable `k` that is declared in the `for` loop. The variable `k` at (2) and (3) refers to the field `k`.

- 10.** (b), (c).

A



The method signature does not include the return type. A `void` method cannot return a value. The return value from a method need not be assigned to a variable.

- 11.** (a) Yes. This is called method overloading.  
(b) Yes. The actual parameters are evaluated in the method call and their values are assigned to formal parameters that are local variables in the method declaration.  
(c) Yes. But a `return` statement in a `void` method cannot return a value.
- 12.** There are many ways to implement methods for the situations in this question. Here are some suggestions:
- (a) A non-`void` method that returns a Boolean value. The return value (`true` or `false`) indicates whether a person has reached retirement age or not.
  - (b) A `void` method, if the method does not need to return a value.
  - (c) A non-`void` method that returns an `int` value.
  - (d) A non-`void` method that returns the `int` value entered by the user.
  - (e) A `void` method that updates the field with the value passed as parameter.
  - (f) A non-`void` method that returns the reference value of the array containing `Counter` objects. The return type is `Counter[]`.
  - (g) A non-`void` method that returns the reference value of the two-dimensional array containing `String` objects. The return type is `String[][]`.
  - (h) A non-`void` method that returns the reference value of the two-dimensional array containing `int` values, representing the sales of newspapers. The return type is `int[][]`. Note that the array length is not specified in the return type. The first index from the left represents a week in a four-week period, and the second index represents a day in the week.

A



- 13.** Printout:

```
Before method call: counter value is equal to 1 and number of times is 5
After method call: counter value is equal to 6 and number of times is 5
```

Actual parameter values are not changed after the method call. For actual parameters that are references to objects, the object state can have changed, as we can see from the printout above.

- 14.** Printout:

```
Before swapping: counter1 is 10 and counter2 is 20
After swapping: counter1 is 10 and counter2 is 20
```

Actual parameter values are not changed after the method call. This also applies to actual parameters that are references. They refer to the same objects as they did before the call.

**15.** (b), (c), (d), (e), (f).

The type list of the formal parameters is (`Counter`, `Counter[]`), i.e. the actual parameter list must comprise a reference to a `Counter` object and a reference to an array of `Counter` objects. All alternatives, except (a) and (g), have a type list that is compatible with the type list of the formal parameters.

(a) is not valid because the actual parameter `counterArrayA[]` is not an array reference. It is a reference, but to a `Counter` object.

(g) is not valid because the type list of the actual parameters is (`Counter[]`, `Counter[][][]`).

**16.** Using the `this` reference in the class:

```
class Counter {
 final static int MAX_VALUE = 100;
 static String description = "This class creates counters.";
 int value;

 Counter() { this.value = 1; }
 Counter(int initialValue) { this.value = initialValue; }

 int getCounter() { return this.value; }
 void setCounter(int newValue) { this.value = newValue; }
 void incrementCounter() { ++this.value; }
 void decrementCounter() { --this.value; }
 void resetCounter() { this.value = 0; }

 static String getDescription() { return description; } // No this
}
```

A



**17.** (a), (c), (d), (e), (g), (h).

(b) and (f) are not valid, because instance members cannot be accessed using the class name.

**18.** (c), (d).

Overloaded constructors have different formal parameter lists. A constructor is not a method, and it cannot have a return type. A constructor is invoked on the object created by the `new` operator, and can therefore refer to this object using the `this` reference in the constructor body.

**19.** Printout if we use (a): 0

Printout if we use (b): 600

Printout if we use (c): 50

In (c), the constructor overwrites the initial values that the field variables were assigned in their declarations.

**20.** (a): (1) is valid, since the implicit default constructor is called. (2) is not valid, since there is no non-default constructor declared.

(b): (1) is valid, since the explicit default constructor is called. (2) is not valid, since there is no non-default constructor declared.

(c): (1) is not valid, since the explicit default constructor is not declared. (2) is valid, since the non-default constructor is called.

(d): (1) is valid, since the explicit default constructor is called. (2) is valid, since the non-default constructor is called.

**21.** The following lines will result in a compile time error: (3), (10), (15), (16), (18), (20).

(3), (10) and (18) are not valid, because instance members cannot be accessed by using the class name. (15) and (16) are not valid, because we cannot access instance members inside a static method, with or without the `this` reference. (20) is not valid, because the `this` reference is not available in a static method.

**22.** (f). It is not possible to create objects of an enumerated type using the `new` operator, but we can declare variables of an enumerated type.

**23.** (a), (b), (c), (d), (e).

(a): The name of the enumerated type must be used together with the name of the enum constant.

```
colour = LightColour.GREEN;
```

(b): We must create an array containing the enum constants by calling the `values()` method before we can iterate over the constants.

```
for (LightColour colour : LightColour.values()) {
 System.out.println(colour);
}
```

(c): The name of the enumerated type should not be specified in the `case` label.

```
switch(colour)
 case RED: System.out.println("STOP!"); break;
 case GREEN: System.out.println("GO!"); break;
 case YELLOW: System.out.println("CAREFUL!"); break;
 default: assert false: "UNKNOWN COLOUR!";
}
```

(d): The name of the enumerated type must be used together with the name of the enum constant.

```
boolean b = colour.equals(LightColour.GREEN);
```

(e): Cannot assign to an enum constant as it is `final`.



## A.8 Object communication

1. The properties and behaviour defined for a class determine what *responsibilities* the class has, and what *roles* objects of this class can fulfil.

2. (c), (d).

Only reference values of objects are passed, and the called method may change the object state.

3. In Java an object can ask another object to do something by *calling a method* of the other object.

4. An association between two classes can be established by one or both of the classes declaring a *field variable* in order to store a *reference value* of an object of the other class.

5. An association in which a `Car` object owns four `Tire` objects is called a *one-to-many* association.

6. (a).

The compiler selects the method body that will be executed during compilation of the code by examining the signatures of the method declarations.

A



## A.9 Sorting and searching arrays

1. Values of all numerical data types in Java, such as `int` and `double`, have a *natural order*. The Java programming language also defines a set of *relational* operators that can be used to compare numerical values.

2. The only primitive data type in Java whose values cannot be compared is `boolean`. Values of this data type can only be compared for *equality*.

3. There will be no output from statements (1) to (5), as their assertions are all valid. The output from statement (6) will be a message from the Java Virtual Machine stating that an `AssertionError` occurred during execution. The file name, as well as the line number that caused the error, will be reported:

```
Exception in thread "main" java.lang.AssertionError
at ComparePrimitiveValues.main(ComparePrimitiveValues.java:10)
```

4. Some possible ways to verify that the letter 'a' precedes the letter 'z' are:

```
assert 'a' < 'z';
assert 'a' <= 'z';
assert 'z' > 'a';
assert 'z' >= 'a';
assert ('z' - 'a') > 0;
```

- 5.** The following assert statement verifies that the Unicode standard defines as many lowercase as uppercase letters in the English alphabet:

```
assert ('z' - 'a') == ('Z' - 'A');
```

- 6.** One suggestion for an assertion:

```
assert testScore >= MIN_SCORE && testScore <= MAX_SCORE :
 "Reported test score " + testScore + " is outside valid range " +
 MIN_SCORE + " to " + MAX_SCORE;
```

- 7.** To enable comparison of two objects, their class has to implement the `Comparable` interface, which defines the `compareTo()` method used to compare objects.

- 8.** (c) provides the correct implementation of the `compareTo()` method. For (a), the method signature is incorrect, as the parameter to `compareTo()` must be of type `Object`. Method (a) also compares the two title fields for reference equality, instead of object equality. In method (b) the signature is correct, but the input parameter is not cast to type `CD`, which is required to access the `title` and `noOfTracks` fields. The `Object` class has no such fields.

The class declaration must be modified as follows: (modification in italics)

```
class CD implements Comparable {
```

- 9.** The CDSorter program produces the following output:

```
CD list:
The CD entitled 'Java Jive' has 5 tracks.
The CD entitled 'T Cup Blues' has 7 tracks.
The CD entitled 'Another cup of Joe' has 6 tracks.
The CD entitled 'T Cup Blues' has 8 tracks.
Sorted CD list:
The CD entitled 'Another cup of Joe' has 6 tracks.
The CD entitled 'Java Jive' has 5 tracks.
The CD entitled 'T Cup Blues' has 7 tracks.
The CD entitled 'T Cup Blues' has 8 tracks.
```

- 10.** (c).

Objects cannot be ordered using relational operators. These operators are only defined for values of the primitive data types. That every individual field of an object has a natural order is not a sufficient criteria for sorting using the `sort()` method of the `java.util.Arrays` class. The class of the object must implement the `Comparable` interface, defining the natural order of its objects. The `sort()` method of the `java.util.Arrays` class is also capable of sorting arrays containing values of primitive data types such as `int` and `double`.

- 11.** (d).

Neither the selection sort nor insertion sort algorithms are able to detect a partially-sorted array. The algorithms will compare the current element in each pass with *every* element in the remaining unsorted part and sorted subarray respectively. Both algorithms perform the same number of comparisons per pass. Thus, statements (a),



(b) and (c) are all false. However, selection sort only performs three assignments per pass, for swapping the smallest (or largest) value in the unsorted part with the first element of the unsorted part. In contrast, insertion sort shifts up to  $n-2$  element values per pass, each shift requiring one assignment operation.

## A.10 Text file I/O and simple GUI dialogs

1. (b), (e).

Only (b) and (e) are text files. The others are binary files.

2. (a), (b).

3. (b).

(a) and (c) are equivalent.

4. (a), (b), (c).

5. (a), (c).

6. (a).

7. (b).

Alternative (b) will avoid printing "null" when the end of file is reached.

8. (c).

(c) will ensure that all other connected resources are freed.

9. A method can specify exceptions it will rethrow in a `throws` clause in the method header.

10. (a), (c).

11. (a).

The statements in the `writeToFile()` and `closeWriteFile()` methods do not throw an `IOException`. A call to the `FileWriter` constructor in the `openFileForWrite()` method can throw an `IOException`. The `openFileForWrite()` method must specify an `IOException` in a `throws` clause, and so must the `main()` method, as it calls the `openFileForWrite()` method.

12. (c).

The `main()` method calls the other three methods, which all contain code that can throw an `IOException`. All *four* methods must specify an `IOException` in a `throws` clause.

13. (a), (b).

A



The `showConfirmDialog()` method can only show combinations specified by the following constants in the `JOptionPane` class: `DEFAULT_OPTION` (i.e. “OK” button), `YES_NO_OPTION`, `YES_NO_CANCEL_OPTION`, `OK_CANCEL_OPTION`.

**14.** (a), (b), (c).

**15.** (a), (b).

However, (a) will not return any value to indicate which button the user clicked.

A



# Language reference

---

## B.1 Reserved words

Keywords are reserved words in the Java programming language. These words have a predefined meaning and therefore cannot be used for other purposes in the source code than that which is defined in the Java language specification. Incorrect use will be reported by the compiler.

Table B.1 shows all the keywords that are defined in Java. In addition, three reserved words are used as literals (Table B.2). Table B.3 shows reserved words that are not used in the current version of Java (version 6.0), but may be used in future versions. All reserved words in Java are written in lowercase letters.

**TABLE B.1**    **Keywords in Java**

|          |         |            |              |           |
|----------|---------|------------|--------------|-----------|
| abstract | default | if         | private      | this      |
| assert   | do      | implements | protected    | throw     |
| boolean  | double  | import     | public       | throws    |
| break    | else    | instanceof | return       | transient |
| byte     | enum    | int        | short        | try       |
| case     | extends | interface  | static       | void      |
| catch    | final   | long       | strictfp     | volatile  |
| char     | finally | native     | super        | while     |
| class    | float   | new        | switch       |           |
| continue | for     | package    | synchronized |           |

**TABLE B.2** Reserved words for literals

|      |      |       |
|------|------|-------|
| null | true | false |
|------|------|-------|

**TABLE B.3** Reserved words for future use

|       |      |
|-------|------|
| const | goto |
|-------|------|

## B.2 Operators

How an expression is evaluated depends on the precedence of the operators it uses. In the case of adjacent operators with different precedence, the operator with highest precedence is applied first. Adjacent operators with the same precedence are evaluated according to associativity rules: either operators are grouped with operands from left to right (for left-associative operators), or from right to left (for right-associative operators). Parentheses, “()”, can always be used to overrule operator precedence and associativity rules. It is a good idea to use parentheses to emphasise how an expression is evaluated.

The index can be used to find where the different operators are explained and used in program examples.

Table B.4 gives a overview of the operators defined in Java. Note the following about the operators shown in Table B.4:

- The operators are arranged in decreasing order of precedence from the start of the table. Thus the assignment operators have the lowest precedence.
- Operators in the same row have the same precedence.
- All operators that are not listed either as *unary* (i.e. demanding one operand) or *ternary* (i.e. requiring three operands), are *binary* (i.e. demanding two operands).
- All binary operators except for assignment operators associate with operands *from left to right*.
- All unary operators, except for unary post-decrement and post-increment operators, all assignment operators, and the ternary conditional operator, associate *from right to left*.

**TABLE B.4** Operators in Java

| Operator     | Operation                         | Example              |
|--------------|-----------------------------------|----------------------|
| []           | Indexing in arrays                | args[0]              |
| .            | Accessing class members           | System.out.println() |
| (parameters) | Actual parameters in method calls | str.indexOf('%', 10) |
| ++           | Unary post-increment              | i++                  |
| --           | Unary post-decrement              | j--                  |

| Operator                    | Operation                                               | Example                                       |
|-----------------------------|---------------------------------------------------------|-----------------------------------------------|
| <code>++</code>             | Unary pre-increment                                     | <code>++i</code>                              |
| <code>--</code>             | Unary pre-decrement                                     | <code>--j</code>                              |
| <code>+</code>              | Unary plus                                              | <code>+x</code>                               |
| <code>-</code>              | Unary minus                                             | <code>-y</code>                               |
| <code>~</code>              | Unary bitwise complement                                | <code>~flag</code>                            |
| <code>!</code>              | Unary Boolean negation                                  | <code>!done</code>                            |
| <code>new<br/>(type)</code> | Unary object creation<br>Unary type conversion operator | <code>new Integer(2007)<br/>(int) 3.14</code> |
| <code>*</code>              | Multiplication                                          | <code>9 * 2</code>                            |
| <code>/</code>              | Integer division                                        | <code>9 / 2</code>                            |
|                             | Floating-point division                                 | <code>9.0 / 2.0</code>                        |
| <code>%</code>              | Modulus (remainder of division)                         | <code>9 % 2</code>                            |
| <code>+</code>              | Addition                                                | <code>9 + 2</code>                            |
|                             | String concatenation                                    | <code>"July" + 2006</code>                    |
| <code>-</code>              | Subtraction                                             | <code>9 - 2</code>                            |
| <code>&lt;&lt;</code>       | Bitwise left shift                                      | <code>i &lt;&lt; 4</code>                     |
| <code>&gt;&gt;</code>       | Bitwise right shift with sign extension                 | <code>j &gt;&gt; 4</code>                     |
| <code>&gt;&gt;&gt;</code>   | Bitwise right shift with 0 extension                    | <code>k &gt;&gt;&gt; 4</code>                 |
| <code>&lt;</code>           | Less than                                               | <code>a &lt; b</code>                         |
| <code>&lt;=</code>          | Less than or equal to                                   | <code>a &lt;= b</code>                        |
| <code>&gt;</code>           | Greater than                                            | <code>a &gt; c</code>                         |
| <code>&gt;=</code>          | Greater than or equal to                                | <code>a &gt;= c</code>                        |
| <code>instanceof</code>     | Type comparison for objects                             | <code>str instanceof String</code>            |
| <code>==</code>             | Equal to: primitive values                              | <code>i == j</code>                           |
|                             | Equal to: reference values                              | <code>str1 == str2</code>                     |
| <code>!=</code>             | Not equal to: primitive values                          | <code>i != j</code>                           |
|                             | Not equal to: reference values                          | <code>str1 != str2</code>                     |
| <code>&amp;</code>          | Logical AND                                             | <code>(i &gt; 0) &amp; (i &lt;= 10)</code>    |
|                             | Bitwise AND                                             | <code>i &amp; 0277</code>                     |
| <code>^</code>              | Logical exclusive OR                                    | <code>overLimit ^ belowLimit</code>           |
|                             | Bitwise exclusive OR                                    | <code>i ^ 0277</code>                         |
| <code> </code>              | Logical OR                                              | <code>found   finished</code>                 |
|                             | Bitwise OR                                              | <code>i   0277</code>                         |
| <code>&amp;&amp;</code>     | Conditional AND                                         | <code>washed &amp;&amp; ironed</code>         |
| <code>  </code>             | Conditional OR                                          | <code>washed    cleaned</code>                |
| <code>? :</code>            | Ternary conditional expression                          | <code>(i &lt; 0) ? -i : i</code>              |



| Operator | Operation                                            | Example              |
|----------|------------------------------------------------------|----------------------|
| =        | Assignment of:<br>primitive value<br>reference value | i = 2<br>str1 = str2 |
|          | Assignment <i>after</i> :                            |                      |
| +=       | addition                                             | i += 2               |
|          | string concatenation                                 | str1 += "2002"       |
| -=       | subtraction                                          | i -= 2               |
| *=       | multiplication                                       | i *= 2               |
| /=       | integer division                                     | i /= 2               |
|          | floating-point division                              | x /= 2.0             |
| %=       | calculation of remainder                             | i %= 2               |
| <<=      | bitwise left shift                                   | i <<= 2              |
| >>=      | bitwise right shift with sign extension              | i >>= 2              |
| >>>=     | bitwise right shift with 0 extension                 | i >>>= 2             |
| &=       | logical AND                                          | flag &= found        |
|          | bitwise AND                                          | i &= 0277            |
| ^=       | logical exclusive OR                                 | flag ^= found        |
|          | bitwise exclusive OR                                 | i ^= 0277            |
| =        | logical OR                                           | flag  = found        |
|          | bitwise OR                                           | i  = 0277            |

B



### B.3 Primitive data types

Table B.5 shows the range for the predefined primitive data types in Java. The minimum and the maximum values for each primitive data type are defined by the constants *Wrapper.MIN\_VALUE* and *Wrapper.MAX\_VALUE* of the corresponding wrapper class respectively.

TABLE B.5 Primitive data types in Java

| Data type | Width (bits)   | Minimum value, maximum value                          | Wrapper class |
|-----------|----------------|-------------------------------------------------------|---------------|
| boolean   | not applicable | true, false (no ordering implied)                     | Boolean       |
| byte      | 8              | $-2^7$ (-128), $2^7 - 1$ (+127)                       | Byte          |
| char      | 16             | 0x0 (Unicode value \u0000), 0xffff (\uffff)           | Character     |
| double    | 64             | 4.94065645841246544e-324,<br>1.79769313486231570e+308 | Double        |

| Data type | Width (bits) | Minimum value, maximum value                                                            | Wrapper class |
|-----------|--------------|-----------------------------------------------------------------------------------------|---------------|
| float     | 32           | 1.401298464324817e-45f,<br>3.402823476638528860e+38f                                    | Float         |
| int       | 32           | -2 <sup>31</sup> (-2147483648), 2 <sup>31</sup> -1 (+2147483647)                        | Integer       |
| long      | 64           | -2 <sup>63</sup> (-9223372036854775808L), 2 <sup>63</sup> -1<br>(+9223372036854775807L) | Long          |
| short     | 16           | -2 <sup>15</sup> (-32768), 2 <sup>15</sup> -1 (+32767)                                  | Short         |

## B.4 Java modifiers

Table B.6 summarises the accessibility of classes and interfaces when placed inside a Java package. Table B.7 shows additional modifiers for classes and interfaces, while Table B.8 shows accessibility modifiers for *members* of a class, i.e. its fields and methods.

TABLE B.6 Accessibility modifiers for classes and interfaces

| Modifier                        | Top-level classes and interfaces                |
|---------------------------------|-------------------------------------------------|
| default (no modifier specified) | Accessible in the package (package accessible). |
| public                          | Accessible from everywhere.                     |

TABLE B.7 Other modifiers for classes and interfaces

| Modifier | Classes                                                                              | Interfaces                                                              |
|----------|--------------------------------------------------------------------------------------|-------------------------------------------------------------------------|
| abstract | The class can contain <b>abstract</b> methods, and therefore cannot be instantiated. | Implied. It is not possible to create an instance of an interface type. |
| final    | The class cannot be extended (i.e. it cannot be subclassed).                         | Not possible.                                                           |

TABLE B.8 Accessability modifiers for class members

| Modifier  | Members                                                                                                                  |
|-----------|--------------------------------------------------------------------------------------------------------------------------|
| public    | Accessible from everywhere.                                                                                              |
| protected | Accessible by all classes in the same package as its class, and accessible by subclasses of its class in other packages. |



| Modifier              | Members                                                                                                      |
|-----------------------|--------------------------------------------------------------------------------------------------------------|
| default (no modifier) | Only accessible by classes, including subclasses, in the same package as its class (package access ability). |
| <b>private</b>        | Only accessible in its own class.                                                                            |

B



# Formatted values

## INTRODUCTION

The Java programming language offers the ability to format values. This appendix gives an overview of some commonly used methods from the Java standard library.

### C.1 Syntax for format strings

Table C.1 gives an overview of the methods for the Java standard library that can be used for formatting values. All methods accept a *format string* as their first parameter, followed by zero or more parameters (given by the syntax `Object... args`).

TABLE C.1 Methods for formatting values

| Method                                                                       | Description                                                                                                                                                                                                                                                                                                                                                                        |
|------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>String format(     String formatString,     Object... args )</pre>      | Found in the <code>String</code> , <code>PrintStream</code> and <code>PrintWriter</code> classes.<br>Returns the resulting string with formatted values.                                                                                                                                                                                                                           |
| <pre>PrintStream printf(     String formatString,     Object... args )</pre> | Found in the <code>PrintStream</code> class. Generates a string, with formatted values, and prints the string to the current <code>PrintStream</code> . Returns the reference value of the current <code>PrintStream</code> .<br>In particular, the <code>PrintStream</code> object referred to by the reference <code>System.out</code> prints to the terminal window by default. |
| <pre>PrintWriter printf(     String formatString,     Object... args )</pre> | Found in the <code>PrintWriter</code> class. Generates a string with formatted values and prints the string to the current <code>PrintWriter</code> .<br>Returns the reference value of the current <code>PrintWriter</code> .                                                                                                                                                     |

A format string can contain one or more *format specifications*. A format specification comprises a set of *conversion flags* and a *conversion code*. The methods in Table C.1

construct a string from the format string by replacing each format specification with the string representation of the corresponding parameter.

A format specification has the general form:

"%[*argumentIndex\$*][*flags*][*width*][.*precision*]*code*"

All the elements in square brackets are optional. A format specification is always initiated by the character '%'. The optional *argumentIndex* is used to indicate which of the parameters that follow the format string will be formatted. The first parameter after the format string *format* has index 1 and is identified with "1\$", the second parameter with "2\$", and so on (see Table C.9). If no argument index is given, the parameters after the format string are used in the sequence they are listed in the call to the *format()* or *printf()* method. Thus the first parameter after the format string is converted using the first format specification, the second parameter is converted using the second specification, and so on.

The optional conversion flags, *flags*, are used to control the conversion for different types of values. Which flags are allowed is determined by the mandatory conversion code, *code*, always placed at the end of the format specification. The optional width specifies the minimum number of characters the printout must have, while the optional precision specifies the maximum number of characters to print. For floating-point values, the precision indicates how many decimals will be printed (for conversion code "%f") or the total number of significant digits (for conversion code "%g"). Note that the dot character '.' separates the width and precision flags. The last element in the format specification is the mandatory conversion code, which determines the type of conversion to be applied to a value.

## C.2 Conversion codes and types

Table C.2 describes how the value of a parameter *p* is converted to a string by different conversion codes, e.g. through the call *printf("%code", p)*. If the parameter *p* cannot be converted to a value of the type defined by the code, an exception is thrown and the program terminated.

Uppercase conversion codes will generate a printout containing only uppercase letters.

Java also offers a set of conversion flags to control the conversion of values to strings. A subset of these flags is listed in Table C.3.

Table C.4 shows which conversion flags and codes can be combined for each parameter type.



**TABLE C.2** Formatting parameter **p** with different conversion codes

| Conversion code | Description                                                                                                                                                                                                                                                                                                                                                       |
|-----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| %               | Results in the character literal '%' being inserted into the resulting string.                                                                                                                                                                                                                                                                                    |
| a, A            | The value of p is converted to a hexadecimal floating-point number, formatted in exponential form.                                                                                                                                                                                                                                                                |
| b, B            | If p is a Boolean variable, a string representation of its value is generated by calling the method <code>String.valueOf()</code> . This results in the string "true" for the value true, and the string "false" for the value false. For reference types, this conversion results in the string "false" if p has the value null and the string "true" otherwise. |
| c, C            | Results in a string containing a single Unicode character.                                                                                                                                                                                                                                                                                                        |
| d               | The value of p is converted to an integer, which is formatted as a string where the integer is represented in the decimal numeral system (see Appendix F).                                                                                                                                                                                                        |
| e, E            | The value of p is converted to a floating-point number, formatted in scientific notation.                                                                                                                                                                                                                                                                         |
| f               | The value of p is converted to a floating-point number with base 10 and without an exponent.                                                                                                                                                                                                                                                                      |
| g, G            | The value of p is converted to a floating-point number, given either in scientific notation or decimal notation depending on the size of the exponent.                                                                                                                                                                                                            |
| h, H            | If p is equal to null, the result is the string "null". Otherwise, the hash code is generated and converted to a string as a hexadecimal number.                                                                                                                                                                                                                  |
| n               | Results in the platform-specific line terminator string being inserted into the resulting string.                                                                                                                                                                                                                                                                 |
| o               | The value of p is converted to an octal integer (see Appendix F).                                                                                                                                                                                                                                                                                                 |
| s, S            | If p is equal to null, the result is the string "null". If the class of p implements the <code>java.util.Formattable</code> interface, the value of p is converted to a string through the method call <code>p.formatTo()</code> . Otherwise, the result of the conversion is given by the method call <code>p.toString()</code> .                                |
| x, X            | The value of p is converted to a hexadecimal integer (see Appendix F).                                                                                                                                                                                                                                                                                            |

C



**TABLE C.3** Conversion flags

| Conversion flag  | Description                                                                                                                                                                                          |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -                | The result is left-justified, positioned within the width defined in the format specification. If this flag is not specified, the converted value is right-justified within the defined field width. |
| #                | The result is prefixed with "0", "0x" or "0X" for integer values. For all other types of values, the formatting depends on the definition of the <code>java.util.Formattable</code> interface.       |
| +                | The result will always contain a sign, either '+' for positive values or '-+' for negative values.                                                                                                   |
| space (i.e. ' ') | The result will include a space character for positive values.                                                                                                                                       |
| 0                | The result will be left-filled with the digit 0 to fill the defined width.                                                                                                                           |
| ,                | The result will include a separator character that is determined by the locale that is set when the program is run.                                                                                  |
| (                | Negative values will be enclosed in parentheses.                                                                                                                                                     |

**TABLE C.4** Combinations of conversion flags and codes

| Desired formatting                                                  | Conversion flag  | Conversion code | Parameter type                    |
|---------------------------------------------------------------------|------------------|-----------------|-----------------------------------|
| Decimal number                                                      | -, +, 0, ,, (    | d               | Integer.                          |
| Octal number                                                        | -, #, 0          | o               | Integer.                          |
| Hexadecimal number                                                  | -, #, 0          | x or X          | Integer.                          |
| Floating-point number                                               | -, #, +, 0, ,, ( | d               | Integer or floating-point number. |
| Floating-point number with scientific notation or as decimal number | -, #, +, 0, ,, ( | g or G          | Integer or floating-point number. |
| Floating-point number with scientific notation                      | -, #, +, 0, ,, ( | e or E          | Integer or floating-point number. |
| Percent character                                                   | none             | %               |                                   |
| Newline character                                                   | none             | n               |                                   |



## C.3 Examples

The tables below illustrate how different types of values are formatted with different combinations of conversion codes and flags.

**TABLE C.5** Formatting of integer values

| Value   | "%d"      | "%+d"      | "%8d"     | "%-8d"    | "%08d"     | "%,d"       |
|---------|-----------|------------|-----------|-----------|------------|-------------|
| 0       | "0"       | "+0"       | "0"       | "0"       | "00000000" | "0"         |
| 1024    | "1024"    | "+1024"    | "1024"    | "1024"    | "00001024" | "1,024"     |
| -999    | "-999"    | "-999"     | "-999"    | "-999"    | "-0000999" | "-999"      |
| 1650237 | "1650237" | "+1650237" | "1650237" | "1650237" | "01650237" | "1,650,237" |

**TABLE C.6** Formatting of integer values (cont.)

| Value | "%(d"   | "%+- ,11d" | "%x"       | "%#x"        | "%0"          | "%,#0"         |
|-------|---------|------------|------------|--------------|---------------|----------------|
| 0     | "0"     | "+0"       | "0"        | "0x0"        | "0"           | "00"           |
| 1024  | "1024"  | "+1,024"   | "400"      | "0x400"      | "2000"        | "02000"        |
| -999  | "(999)" | "-999"     | "fffffc19" | "0xfffffc19" | "37777776031" | "037777776031" |

**TABLE C.7** Formatting of floating-point values

| Value    | "%f"          | "%.2f"    | "%7.2f"   | "%07.2f"  | "%4g"    | "%.4e"        |
|----------|---------------|-----------|-----------|-----------|----------|---------------|
| 0.0      | "0.000000"    | "0.00"    | "0.00"    | "0000.00" | "0.000"  | "0.0000e+00"  |
| 0.3      | "0.300000"    | "0.30"    | "0.30"    | "0000.30" | "0.3000" | "3.0000e-01"  |
| 1.0      | "1.000000"    | "1.00"    | "1.00"    | "0001.00" | "1.000"  | "1.0000e+00"  |
| 1.5      | "1.500000"    | "1.50"    | "1.50"    | "0001.50" | "1.500"  | "1.5000e+00"  |
| -1.5     | "-1.500000"   | "-1.50"   | "-1.50"   | "-001.50" | "-1.500" | "-1.5000e+00" |
| 678.9    | "678.900000"  | "+678.90" | "678.90"  | "0678.90" | "678.9"  | "6.7890e+02"  |
| 1234.567 | "1234.567000" | "1234.57" | "1234.57" | "1234.57" | "1235"   | "1.2346e+03"  |



**TABLE C.8**    **Formatting of strings**

| Value    | "%s"     | "%10s"   | "%-10s"  | "%.2s" "%%7.2s" | "%7.0s" | "%.3s" |
|----------|----------|----------|----------|-----------------|---------|--------|
| "Hello!" | "Hello!" | "Hello!" | "Hello!" | "He" "He"       | "He"    | "Hel"  |

**TABLE C.9**    **Using the argument index**

| Argument list | "%1\$d - %2\$d - %3\$d" | "%3\$d - %2\$d - %1\$d" | "%2\$d - %3\$d - %1\$d" |
|---------------|-------------------------|-------------------------|-------------------------|
| 2006, 12, 23  | "2006-12-23"            | "23-12-2006"            | "12-23-2006"            |



# The Unicode character set

## INTRODUCTION

This appendix presents a selection of Unicode characters and their code values.

### D.1 Excerpt from the Unicode character set

Table D.1 shows some characters from the Unicode character set, along with their decimal value and their char literal representation. The characters in Table D.1 are also a part of the ASCII character set, except for the '€' character.

The character codes for the characters in the groups consisting of uppercase letters, lowercase letters and digits occur consecutively in the character set. For example, if we know that the character code for the character A is 'A', i.e. 65, then the character code for character B is 'A' + 1, i.e. 66 or 'B'.

TABLE D.1 Selected values from the Unicode character set

| Decimal value | Char literal | Character | Decimal value | Char literal | Character | Decimal value | Char literal | Character |
|---------------|--------------|-----------|---------------|--------------|-----------|---------------|--------------|-----------|
| 32            | \u0020       | space     | 64            | \u0040       | @         | 96            | \u0060       | '         |
| 33            | \u0021       | !         | 65            | \u0041       | A         | 97            | \u0061       | a         |
| 34            | \u0022       | "         | 66            | \u0042       | B         | 98            | \u0062       | b         |
| 35            | \u0023       | #         | 67            | \u0043       | C         | 99            | \u0063       | c         |
| 36            | \u0024       | \$        | 68            | \u0044       | D         | 100           | \u0064       | d         |
| 37            | \u0025       | %         | 69            | \u0045       | E         | 101           | \u0065       | e         |
| 38            | \u0026       | &         | 70            | \u0046       | F         | 102           | \u0066       | f         |
| 39            | \u0027       | '         | 71            | \u0047       | G         | 103           | \u0067       | g         |



| Decimal value | Char literal | Character | Decimal value | Char literal | Character | Decimal value | Char literal | Character |
|---------------|--------------|-----------|---------------|--------------|-----------|---------------|--------------|-----------|
| 40            | \u0028       | (         | 72            | \u0048       | H         | 104           | \u0068       | h         |
| 41            | \u0029       | )         | 73            | \u0049       | I         | 105           | \u0069       | i         |
| 42            | \u002a       | *         | 74            | \u004a       | J         | 106           | \u006a       | j         |
| 43            | \u002b       | +         | 75            | \u004b       | K         | 107           | \u006b       | k         |
| 44            | \u002c       | ,         | 76            | \u004c       | L         | 108           | \u006c       | l         |
| 45            | \u002d       | -         | 77            | \u004d       | M         | 109           | \u006d       | m         |
| 46            | \u002e       | .         | 78            | \u004e       | N         | 110           | \u006e       | n         |
| 47            | \u002f       | /         | 79            | \u004f       | O         | 111           | \u006f       | o         |
| 48            | \u0030       | 0         | 80            | \u0050       | P         | 112           | \u0070       | p         |
| 49            | \u0031       | 1         | 81            | \u0051       | Q         | 113           | \u0071       | q         |
| 50            | \u0032       | 2         | 82            | \u0052       | R         | 114           | \u0072       | r         |
| 51            | \u0033       | 3         | 83            | \u0053       | S         | 115           | \u0073       | s         |
| 52            | \u0034       | 4         | 84            | \u0054       | T         | 116           | \u0074       | t         |
| 53            | \u0035       | 5         | 85            | \u0055       | U         | 117           | \u0075       | u         |
| 54            | \u0036       | 6         | 86            | \u0056       | V         | 118           | \u0076       | v         |
| 55            | \u0037       | 7         | 87            | \u0057       | W         | 119           | \u0077       | w         |
| 56            | \u0038       | 8         | 88            | \u0058       | X         | 120           | \u0078       | x         |
| 57            | \u0039       | 9         | 89            | \u0059       | Y         | 121           | \u0079       | y         |
| 58            | \u003a       | :         | 90            | \u005a       | Z         | 122           | \u007a       | z         |
| 59            | \u003b       | ;         | 91            | \u005b       | [         | 123           | \u007b       | {         |
| 60            | \u003c       | <         | 92            | \u005c       | \         | 124           | \u007c       |           |
| 61            | \u003d       | =         | 93            | \u005d       | ]         | 125           | \u007d       | }         |
| 62            | \u003e       | >         | 94            | \u005e       | ^         | 126           | \u007e       | ~         |
| 63            | \u003f       | ?         | 95            | \u005f       | _         | 8364          | \u20ac       | €         |

## D.2 Lexicographical order and alphabetical order

The `compareTo()` method of the `String` class compares two strings lexicographically according to the Unicode values of the characters in the strings. In many languages characters are not typically ordered according to their Unicode values. Table D.2 shows some character used in the Norwegian language ordered according to their character codes: Å, Æ, Ø, å, æ, ø. However, when sorting Norwegian text, the lexicographical order should be: Æ, Ø, Å, æ, ø, å. A `java.text.Collator` and a `java.util.Locale` provide the solution for sorting text in Norwegian order:

```
Collator norwayCollator = Collator.getInstance(new Locale("no"));
System.out.println(norwayCollator.compare("øl", "ål") < 0); // true
```

TABLE D.2 Characters used in Norwegian

| Decimal value | Char literal | Character | Decimal value | Char literal | Character | Decimal value | Char literal | Character |
|---------------|--------------|-----------|---------------|--------------|-----------|---------------|--------------|-----------|
| 197           | \u00c5       | Å         | 216           | \u00d8       | Ø         | 230           | \u00e6       | æ         |
| 198           | \u00c6       | Æ         | 229           | \u00e5       | å         | 248           | \u00f8       | ø         |



D



# Console I/O and simple GUI dialog boxes

## INTRODUCTION

This appendix presents two classes for reading values from the terminal window and for creating simple GUI dialog boxes.

### E.1 Support for console I/O

The `Console` class in Program E.1 provides `readType()` methods for reading integers, floating-point numbers and strings entered via the keyboard, where `Type` can be `Int`, `Double` and `String` respectively. In addition, it provides the method `readToEOL()` to read the remaining input on the current line, i.e. effectively emptying the current line of any input. The class allows multiple values to be entered on a line.

To use the `Console` class with your program, copy the file `Console.java` and compile it in the source code directory of your program.

Program E.2 uses the `Console` class. The program finds the largest integer entered on the keyboard. Note how the user is prompted to re-enter a value in case of error.

#### PROGRAM E.1 Console I/O

```
import java.util.InputMismatchException;
import java.util.Scanner;

/**
 * Class reads values from the console.
 */
public final class Console {
 private Console() {} // Cannot create objects of this class.

 /** Scanner object connected to System.in. */
 private static Scanner keyboard = new Scanner(System.in);

 /**
 * Reads an integer from the console.
 * @return the integer value
 */
 public static int readInt() {
 try {
 return keyboard.nextInt();
 } catch (InputMismatchException e) {
 System.out.println("Please enter an integer value.");
 return readInt();
 }
 }

 /**
 * Reads a double from the console.
 * @return the double value
 */
 public static double readDouble() {
 try {
 return keyboard.nextDouble();
 } catch (InputMismatchException e) {
 System.out.println("Please enter a double value.");
 return readDouble();
 }
 }

 /**
 * Reads a string from the console.
 * @return the string value
 */
 public static String readString() {
 try {
 return keyboard.nextLine();
 } catch (InputMismatchException e) {
 System.out.println("Please enter a string value.");
 return readString();
 }
 }

 /**
 * Reads all remaining input on the current line from the console.
 */
 public static void readToEOL() {
 keyboard.nextLine();
 }
}
```

```

* Reads an int value from the keyboard.
* @return Next value as int
*/
public static int readInt() {
 while (true)
 try {
 return keyboard.nextInt();
 } catch (InputMismatchException ime) {
 reportError();
 }
}

/**
 * Reads a double value from the keyboard.
 * @return Next value as double
*/
public static double readDouble() {
 while (true)
 try {
 return keyboard.nextDouble();
 } catch (InputMismatchException ime) {
 reportError();
 }
}

/**
 * Reads a string from the keyboard.
 * The returned string will not contain any white space.
 * @return Next value as string.
*/
public static String readString() {
 while (true)
 try {
 return keyboard.next();
 } catch (InputMismatchException ime) {
 reportError();
 }
}

/**
 * Reads to the end of line (EOL).
 * @return Remaining input as string.
*/
public static String readToEOL() {
 while (true)
 try {
 return keyboard.nextLine();
 } catch (InputMismatchException ime) {
 reportError();
 }
}

```



```

}

/**
 * Empties the current line, and prints an error message
 * to the terminal window.
 */
private static void reportError() {
 keyboard.nextLine(); // Empty the line first.
 System.out.println("Error in input. Try again!");
}

```

---

## PROGRAM E.2 Using console I/O

```

// Using Console class
public class ConsoleDemo {

 public static void main(String[] args) {
 System.out.println(// (1)
 "Program finds the largest number in a sequence of numbers" +
 " entered at the console.\n" +
 "A negative number signals end of the sequence.");
 int maxValue = 0;
 while (true) {
 System.out.print("Enter an integer: ");
 int n = Console.readInt(); // (2)
 if (n < 0) {
 break;
 }
 if (n > maxValue) {
 maxValue = n;
 }
 }
 System.out.println("Largest number: " + maxValue);
 }
}

```

Compiling and running the program:

```

> javac ConsoleDemo.java Console.java
> java ConsoleDemo

```

Program finds the largest number in a sequence of numbers entered at the console.  
A negative number signals end of the sequence.

```

Enter an integer: 123
Enter an integer: 2006
Enter an integer: zero
Error in input. Try again!

```

9



```
Enter an integer: 2007
Enter an integer: -1
Largest number: 2007
```

---

## E.2 Support for simple GUI dialog boxes

The `GUIDialog` class in Program E.3 provides `requestType()` methods for reading integers, floating-point numbers and strings using a GUI dialog box, where `Type` can be `Int`, `Double` and `String` respectively. Each method allows a prompt to be passed as parameter, which is shown in the input dialog box. In addition, it provides the method `confirmInfo()` to confirm information passed as parameter, as well as the method `prompt()` to display a message to the user.

To use the `GUIDialog` class with your program, copy the file `GUIDialog.java` and compile it in the source code directory of your program. Remember to end your program with the following statement to stop all execution:

```
System.exit(0);
```

Program E.4 uses the `GUIDialog` class. The program finds the largest integer in a sequence of integers entered via dialog boxes. Figure E.1 shows the interaction with the user. Note how the user is prompted to re-enter a value in case of error.

### PROGRAM E.3 GUI dialog boxes

```
import javax.swing.JOptionPane;

/**
 * This class provides methods for reading integers, floating-point
 * numbers and strings via simple GUI dialogue boxes.
 * Remember to stop all execution by using this statement in your program:
 * System.exit(0);
 */
public final class GUIDialog {
 private GUIDialog() { } // Cannot create objects of this class.

 /**
 * Reads an int value.
 * @return An int value
 */
 public static int requestInt(Object prompt) { // (1)
 String input = "";
 while (true) // (2)
 try {
 input = JOptionPane.showInputDialog(null, prompt, "Input", // (3)
 JOptionPane.PLAIN_MESSAGE);
 return Integer.parseInt(input); // (4)
 }
}
E
```

```

 } catch (NullPointerException exception) { // (5)
 prompt("'" + input + "' is not a valid integer.");
 } catch (NumberFormatException exception) { // (6)
 prompt("'" + input + "' is not a valid integer.");
 }
 }

/***
 * Reads a double value.
 * @return A double value
 */
public static double requestDouble(Object prompt) { // (7)
 String input = "";
 while (true)
 try {
 input = JOptionPane.showInputDialog(null, prompt, "Input",
 JOptionPane.PLAIN_MESSAGE);
 return Double.parseDouble(input);
 } catch (NullPointerException exception) {
 prompt("'" + input + "' +
 " is not a valid floating-point number.");
 } catch (NumberFormatException exception) {
 prompt("'" + input + "' +
 " is not a valid floating-point number.");
 }
 }

/***
 * Reads a non-empty string.
 * @return A string
 */
public static String requestString(Object prompt) { // (8)
 String input = "";
 while (true)
 try {
 input = JOptionPane.showInputDialog(null, prompt, "Input",
 JOptionPane.PLAIN_MESSAGE);
 if (input == null || input.length() == 0) {
 throw new IllegalArgumentException();
 }
 return input;
 } catch (IllegalArgumentException exception) {
 prompt("'" + input + "' + " is not a valid string.");
 }
 }

/***
 * Prints a message.
 */
public static void prompt(Object message) { // (9)
}

```



```

 JOptionPane.showMessageDialog(null, message, "Message",
 JOptionPane.WARNING_MESSAGE);
 }

 // Return values from the method confirm()
 public static final int YES = JOptionPane.YES_OPTION;
 public static final int NO = JOptionPane.NO_OPTION;
 public static final int CANCEL = JOptionPane.CANCEL_OPTION;

 /**
 * Confirms information.
 * @return YES, if YES button was clicked.
 * @return NO, if NO button was clicked.
 * @return CANCEL, if CANCEL button was clicked.
 */
 public static int confirmInfo(Object information) { // (10)
 return JOptionPane.showConfirmDialog(null, information, "Confirm",
 JOptionPane.YES_NO_CANCEL_OPTION);
 }
}

```

---

#### PROGRAM E.4 Using GUI dialog boxes

```

// Using GUIDialog class
public class GUIDialogDemo {

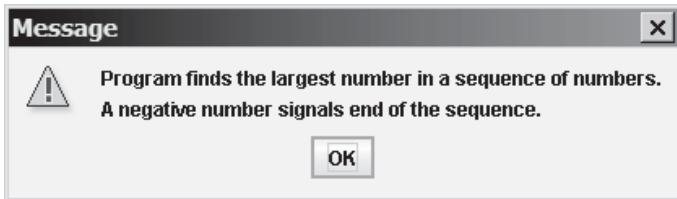
 public static void main(String[] args) {
 GUIDialog.prompt(// (1)
 "Program finds the largest number in a sequence of numbers.\n" +
 "A negative number signals end of the sequence.");
 int maxValue = 0;
 while (true) {
 int n = GUIDialog.requestInt("Enter an integer: "); // (2)
 if (n < 0) {
 break;
 }
 if (n > maxValue) {
 maxValue = n;
 }
 }
 GUIDialog.prompt("Largest number: " + maxValue); // (3)
 System.exit(0);
 }
}

```

---



**FIGURE E.1** Example GUI dialogs



(a)

An input dialog box titled "Input". It has a label "Type an integer:" and a text input field containing "123". Below the input field are "OK" and "Cancel" buttons.

(b)

An input dialog box titled "Input". It has a label "Type an integer:" and a text input field containing "2006". Below the input field are "OK" and "Cancel" buttons.

(c)

An input dialog box titled "Input". It has a label "Type an integer:" and a text input field containing "zero". Below the input field are "OK" and "Cancel" buttons.

(d)



(e)

An input dialog box titled "Input". It has a label "Type an integer:" and a text input field containing "-1". Below the input field are "OK" and "Cancel" buttons.

(f)



(g)



E



# Numeral systems and representation

## INTRODUCTION

This appendix gives an overview of different numeral systems and how we can specify numbers in them. The appendix also gives an introduction to integer representation using 2's compliment, which is the way integers are represented in Java. In addition, an example illustrates how methods from the Java standard library can be used to print a string representation of integers in the different numeral systems.

### F.1 Numeral systems

#### Decimal numeral system

We are used to the *decimal numeral system*. This numeral system uses the digits from 0 to 9 to specify numbers (see Table F.1). This numeral system has ten digits and is also called the *base 10 numeral system*. Each digit that occurs in a number contributes to the value of the number depending on its position in the number, starting with position 0 for the digit that is right-most in the number. Each digit has a *positional value* in the number. For example, the value of the number 123 can be rewritten as follows:

$$123 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 = 1 \cdot 100 + 2 \cdot 10 + 3 \cdot 1 = 100 + 20 + 3$$

It is the position and the base value that determines the weight of each digit in the number. The left-most digit contributes the most to the value of the number, and is therefore called the *most significant digit*. Conversely, the right-most digit is called the *least significant digit*.

There are numeral systems that are not based on this principle. For example, in the Roman numeral system, the digit V has the value 5, regardless of its position in a Roman numeral.


**TABLE F.1** Numeral systems

| Binary (base 2) | Octal (base 8) | Decimal (base 10) | Hexadecimal (base 16) |
|-----------------|----------------|-------------------|-----------------------|
| 0               | 0              | 0                 | 0                     |
| 1               | 1              | 1                 | 1                     |
| 10              | 2              | 2                 | 2                     |
| 11              | 3              | 3                 | 3                     |
| 100             | 4              | 4                 | 4                     |
| 101             | 5              | 5                 | 5                     |
| 110             | 6              | 6                 | 6                     |
| 111             | 7              | 7                 | 7                     |
| 1000            | 10             | 8                 | 8                     |
| 1001            | 11             | 9                 | 9                     |
| 1010            | 12             | 10                | a                     |
| 1011            | 13             | 11                | b                     |
| 1100            | 14             | 12                | c                     |
| 1101            | 15             | 13                | d                     |
| 1110            | 16             | 14                | e                     |
| 1111            | 17             | 15                | f                     |
| 10000           | 20             | 16                | 10                    |

### Binary numeral system

Computers use the *binary numeral system* (also known as the *base 2 numeral system*) to store and handle data. This numeral system uses the digits 0 and 1 only (see Table F.1). This numeral system thus has two different digits, and therefore has the *base 2*. A number in this numeral system consists of a sequence of zeros and ones in which each 0 and 1 is called a *bit*. The number  $1111011_2$  is a binary number. Since it can also be a number in the decimal numeral system, we use the subscript 2 to indicate that it is a number in the binary numeral system.



We can convert a binary number to its corresponding number in the decimal numeral system by calculating the positional value of the digits. The number  $1111011_2$  corresponds to  $123_{10}$  in the decimal numeral system:

$$\begin{aligned}1111011_2 &= 1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 \\&= 1*64 + 1*32 + 1*16 + 1*8 + 0*4 + 1*2 + 1*1 \\&= 64 + 32 + 16 + 8 + 0 + 2 + 1 \\&= 123_{10}\end{aligned}$$

We have used the base value 2 to calculate the positional value of each digit in the binary number. It is important to note that both  $1111011_2$  and  $123_{10}$  represent the same value – we have just specified the value in two different numeral systems.

## Octal numeral system

The bit pattern of a binary number can be long. Therefore we often use the *octal numeral system* (the *base 8 numeral system*). This uses the digits from 0 to 7 (see Table F.1). The octal numeral system has eight different digits, and thus has the *base 8*. The number  $173_8$  is a valid octal number. The number 180 is not, because it uses the digit 8, which is not in the octal numeral system.

We can convert an octal number to the corresponding number in the decimal numeral system by calculating the positional values of the digits, as shown for the binary number above, but using the base value 8. The number  $173_8$  corresponds to the number  $123_{10}$  in the decimal numeral system:

$$\begin{aligned}173_8 &= 1*8^2 + 7*8^1 + 3*8^0 \\&= 1*64 + 7*8 + 3*1 \\&= 64 + 56 + 3 \\&= 123_{10}\end{aligned}$$

The relationship between the binary numeral system and the octal numeral system is explained in Section F.2.

## Hexadecimal numeral system

Numbers can be specified even more compactly by using the *hexadecimal numeral system* (the *base 16 numeral system*). This numeral system has sixteen different digits, and thus has the *base 16*. The first nine digits are the same as those in the decimal numeral system, i.e. from 0 to 9. The remaining seven digits consist of the letters from *a* through *f*, which represent the values from  $10_{10}$  to  $15_{10}$ . For example, the number  $7b$  is a hexadecimal number. A digit between *a* and *f* can be replaced by the corresponding uppercase letter between *A* and *F*. The number  $7B_{16}$  is equivalent to the number  $7b_{16}$ .

We can convert a hexadecimal number to the corresponding number in the decimal system by using the base value 16. The number  $7b_{16}$  is equivalent to the number  $123_{10}$ :

$$\begin{aligned}7b_{16} &= 7*16^1 + b*16^0 \\&= 7*16 + 11*1 \\&= 112 + 11 \\&= 123_{10}\end{aligned}$$



The relationship between the binary numeral system, the octal numeral system and the hexadecimal numeral system is explained in Section F.2.

The procedure above can be generalised for converting any number to the decimal numeral system from a numeral system in the base  $R$ . Given a number  $PQ\dots VW_R$  with  $k+1$  digits, it can be converted to the corresponding number in the decimal numeral system as follows:

$$PQ\dots VW_R = P \cdot R^k + Q \cdot R^{k-1} + \dots + V \cdot R^1 + W \cdot R^0$$

## F.2 Conversions between numeral systems

Table F.1 shows that it requires at the most three bits to represent a digit in the octal numeral system. We also see that it requires at the most four bits to represent a digit in the hexadecimal numeral system. We can use this observation to convert between the binary, octal and hexadecimal systems (see Figure F.1).

The procedure for converting from the octal to the binary numeral system is shown by the arrow (a) in Figure F.1. The following calculation shows that the procedure of replacing each octal digit with a 3-bit binary number is correct:

$$\begin{aligned} 173_8 &= 1 \cdot 8^2 + 7 \cdot 8^1 + 3 \cdot 8^0 \\ &= 1 \cdot (2^3)^2 + 7 \cdot (2^3)^1 + 3 \cdot (2^3)^0 \\ &= 1 \cdot 2^6 + 7 \cdot 2^3 + 3 \\ &= (001_2) \cdot 2^6 + (111_2) \cdot 2^3 + (011_2) \\ &= (0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0) \cdot 2^6 + (1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0) \cdot 2^3 + (0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0) \\ &= 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= 1111011_2 \end{aligned} \quad (1)$$

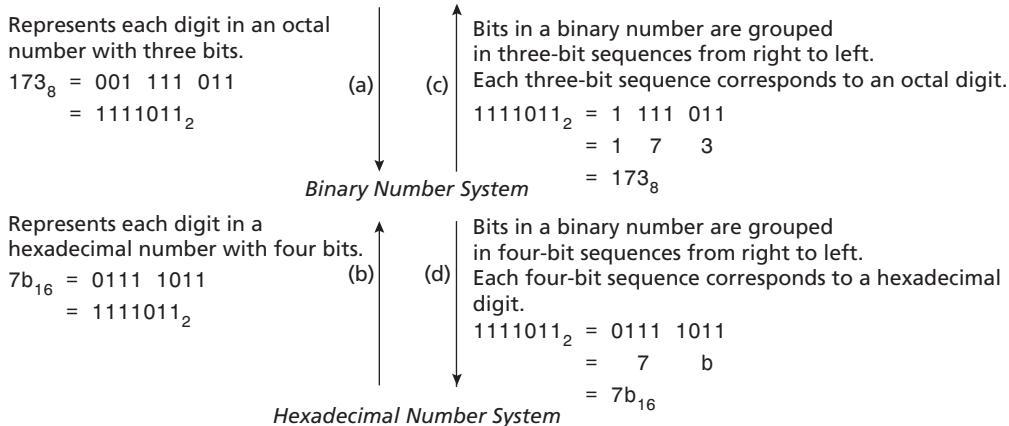
The result in (2) above is the same as the one we get by replacing each octal digit with its equivalent 3-bit sequence in (1) above. Analogously, we can convert from the hexadecimal to the binary numeral system by replacing each hexadecimal digit with the equivalent 4-bit sequence (arrow (b) in Figure F.1).

The procedure for converting from the binary to the octal numeral system is to reverse the procedure shown by the arrow (a) in Figure F.1. The arrow (c) in Figure F.1 shows this procedure. It groups bits in 3-bit sequences from right to left and replaces each 3-bit sequence with its equivalent octal digit. This is equivalent to doing the calculation at (1) in reverse. Analogously, the arrow (d) in Figure F.1 shows how we can convert from the binary to the hexadecimal numeral system.

Figure F.1 also shows how we can convert an octal number to its representation in the hexadecimal numeral system via the binary numeral system: arrows (a) and (d). Analogously, we can convert a hexadecimal number to its representation in the octal numeral system: arrows (b) and (c).



**FIGURE F.1** Conversion between decimal, octal and hexadecimal numbers



### F.3 Conversions from decimal numeral system

We can convert a decimal number to a binary number by reversing the procedure for converting from a binary number to a decimal number (see *Decimal numeral system* on page 757). The procedure involves dividing the quotient repeatedly with the base value 2, until the quotient is equal to 0. Division by the base value is implicit in the following steps:

$$\begin{aligned} 123_{10} &= 61 \cdot 2 + 1 \\ 61_{10} &= 30 \cdot 2 + 1 \\ 30_{10} &= 15 \cdot 2 + 0 \\ 15_{10} &= 7 \cdot 2 + 1 \\ 7_{10} &= 3 \cdot 2 + 1 \\ 3_{10} &= 1 \cdot 2 + 1 \\ 1_{10} &= 0 \cdot 2 + 1 \\ 123_{10} &= 1111011_2 \end{aligned}$$

The last step above assembles the remainders with the last remainder first. We get the binary number  $1111011_2$ , which is equivalent to  $123_{10}$ . We can prove this by substituting the quotients into the equations above:

$$\begin{aligned} 123_{10} &= 61 \cdot 2 + 1 = (30 \cdot 2 + 1) \cdot 2 + 1 = ((15 \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 1 \\ &= (((7 \cdot 2 + 1) \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 1 \\ &= ((((3 \cdot 2 + 1) \cdot 2 + 1) \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 1 \\ &= (((((1 \cdot 2 + 1) \cdot 2 + 1) \cdot 2 + 1) \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 1 \\ &= (((((0 \cdot 2 + 1) \cdot 2 + 1) \cdot 2 + 1) \cdot 2 + 1) \cdot 2 + 0) \cdot 2 + 1 \\ &= 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= 1111011_2 \end{aligned}$$



Analogously, we can convert a decimal number to an octal number by repeatedly dividing the quotient with the base value 8:

$$\begin{aligned}123_{10} &= 15 \cdot 8 + 3 \\15_{10} &= 1 \cdot 8 + 7 \\1_{10} &= 0 \cdot 8 + 1 \\123_{10} &= 173_8\end{aligned}$$

Finally, we show how we can convert a decimal number to a hexadecimal number by repeatedly dividing the quotient with the base value 16:

$$\begin{aligned}123_{10} &= 7 \cdot 16 + 15 = 7 \cdot 16 + b \\7_{10} &= 0 \cdot 16 + 7 \\123_{10} &= 7b_{16}\end{aligned}$$

## F.4 Integer representation

In this section we look at how integer values are represented in memory. We will not discuss floating-point numbers here: details can be found in the comprehensive IEEE-754 standard that Java uses to represent floating-point numbers.

TABLE F.2 Representation of **byte** value with 2's compliment

| Decimal value | Binary representation<br>(8 bits) | Octal value with<br>prefix 0 | Hexadecimal value with<br>prefix 0x |
|---------------|-----------------------------------|------------------------------|-------------------------------------|
| 127           | 01111111                          | 0177                         | 0x7f                                |
| 128           | 01111110                          | 0176                         | 0x7e                                |
| ...           | ...                               | ...                          | ...                                 |
| 123           | 01111011                          | 0173                         | 0x7b                                |
| ...           | ...                               | ...                          | ...                                 |
| 2             | 00000010                          | 02                           | 0x02                                |
| 1             | 00000001                          | 01                           | 0x01                                |
| 0             | 00000000                          | 00                           | 0x0                                 |
| -1            | 11111111                          | 0377                         | 0xff                                |
| -2            | 11111110                          | 0376                         | 0xfe                                |
| ...           | ...                               | ...                          | ...                                 |
| -123          | 10000101                          | 0205                         | 0x85                                |
| ...           | ...                               | ...                          | ...                                 |



| Decimal value | Binary representation (8 bits) | Octal value with prefix 0 | Hexadecimal value with prefix 0x |
|---------------|--------------------------------|---------------------------|----------------------------------|
| -127          | 10000001                       | 0201                      | 0x81                             |
| -128          | 10000000                       | 0200                      | 0x80                             |

Table F.2 shows the representation of values in the byte primitive data type. Values of type byte are represented by eight bits in memory. With eight bits we can represent  $2^8$  or 256 integer values. Integer representation in Java uses 2's complement, which allows both positive and negative integers to be represented in memory. For the byte type, it means we can represent values from -128 (i.e.  $2^7$ ) to +127 (i.e.  $2^7-1$ ).

Before we can understand 2's compliment, we need to understand 1's compliment. The 1's compliment of a binary number is obtained by changing the value of each bit in the number. For example, 1's compliment of  $1111011_2$  is  $0000100_2$ . We use the notation  $\sim N_2$  to denote the 1's compliment of a binary number  $N_2$ . 2's compliment is defined in terms of 1's compliment:

$$-N_2 = \sim N_2 + 1 \quad (1)$$

In the equation above, we calculate 2's compliment,  $-N_2$ , of a binary number  $N_2$  by adding 1 to 1's compliment,  $\sim N_2$ . If binary number  $N_2$  is a positive number, 2's compliment will be the corresponding negative number, and vice versa. For example, given the positive number  $1111011_2$ , we can calculate 2's compliment as follows:

$$\begin{aligned} N_2 &= 01111011 = 123_{10} \\ \sim N_2 &= 10000100 \\ +1 &= 1 \\ -N_2 &= 10000101 = -123_{10} \end{aligned}$$

Analogously, we can calculate 2's compliment of a negative number:

$$\begin{aligned} N_2 &= 10000101 = -123_{10} \\ \sim N_2 &= 01111010 \\ +1 &= 1 \\ -N_2 &= 01111011 = 123_{10} \end{aligned}$$

Adding a number and its 2's compliment always gives the result 0 (zero):

$$\begin{array}{r} 01111011 = 123_{10} \\ +10000101 = +(-123_{10}) = -123_{10} \\ \hline 00000000 = 0_{10} \end{array}$$

The *carry bit* from the left-most bit is ignored. The example above also shows that subtraction is calculated as addition with 2's compliment:

$$N_2 - M_2 = N_2 + (-M_2)$$



For example, the expression  $(123_{10} - 2_{10})$  is calculated as follows, giving the correct result  $121_{10}$ :

$$\begin{array}{r}
 123_{10} = 01111011 \\
 + (-2_{10}) = +11111110 \\
 \hline
 121_{10} = 01111001
 \end{array}$$

In the binary representation, the left-most bit is called the *most significant bit*, and the right-most bit is called the *least significant bit*. Table F.2 shows that the most significant bit has the value 0 for positive integers and the value 1 for negative numbers.

Values of the primitive data types `short`, `int` and `long` are also represented by 2's complement with the number of bits 16, 32 and 64, respectively.

## F.5 String representation of integers

In Java it is possible to specify integer values in the decimal, octal and hexadecimal numeral systems:

```

int i = 123; // Decimal: no prefix.
int j = 0173; // Octal: prefix 0 (zero) required.
int k = 0x7b; // Hexadecimal: prefix 0x required.

```

It is *not* possible to specify integer values in the binary numeral system.

The wrapper classes `Integer` and `Long` provide methods that return a string representation of integers in the different numeral systems. A selection of these methods from the `Integer` class for `int` values (32 bits) is shown in Table F.3. The class `Long` provides analogous methods for `long` values (64 bits).

Program F.1 uses all the methods from Table F.3 to print the string representation of `int` values in the different numeral systems. Note that the prefix 0 and the prefix 0x for octal and hexadecimal notations are *not* a part of the string representation. The prefix can be printed by specifying the appropriate format conversion code when calling the `printf()` method provided by the `PrintStream` class (see Appendix C).

TABLE F.3 Methods for different string representations of integers

| <code>java.lang.Integer</code>                                                                                       |                                                                                                                                                                                                                                                                            |
|----------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> static String toBinaryString(int i) static String toOctalString(int i) static String toHexString(int i) </pre> | Returns the string representation of the integer value in the parameter <code>i</code> depending on the method called, either in the binary, octal or hexadecimal numeral system respectively. The string representation is for an unsigned integer without leading zeros. |



## java.lang.Integer

---

|                                            |                                                                                                                                                                                                                         |
|--------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>static String toString(int i)</code> | Returns the string representation of the integer value in the parameter <code>i</code> as defined in the decimal numeral system, with the sign ' <code>-</code> ' as the first character if the integer is less than 0. |
|--------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

---

### PROGRAM F.1 String representation of integers

```
// Writes different string representations of integers.
public class IntegerRepresentation {
 public static void main(String[] args) {
 int i = Integer.parseInt(args[0]);
 System.out.println("String representation for integers: " + i);
 System.out.println("Decimal: " + Integer.toString(i));
 System.out.println("Binary: " + Integer.toBinaryString(i));
 System.out.println("Octal: " + Integer.toOctalString(i));
 System.out.println("Hexadecimal: " + Integer.toHexString(i));
 }
}
```

Running the program:

```
> java IntegerRepresentation 123
String representation for integers: 123
Decimal: 123
Binary: 1111011
Octal: 173
Hexadecimal: 7b
> java IntegerRepresentation -123
String representation for integers: -123
Decimal: -123
Binary: 1111111111111111111111110000101
Octal: 3777777605
Hexadecimal: ffffff85
```

---



# Programming tools in the JDK

## INTRODUCTION

This appendix describes some commonly used tools for developing and running Java programs.

### G.1 Programming tools

Sun Microsystems, Inc. offers a toolkit for Java called the Java Development Kit (JDK). This toolkit contains all necessary tools for writing and running Java programs. The latest version of the JDK can be freely downloaded from <http://java.sun.com/downloads/>. The most important tools in the JDK are:

- **javac**: the Java compiler. Translates source code to byte code.
- **java**: the Java Virtual Machine (JVM). Provides the runtime environment for executing Java byte code.
- **javadoc**: a documentation tool. Generates HTML documentation from source code.

### G.2 Commands

The tools in the Java JDK are stand-alone applications that can be run from the command line. All the tools accept a command line option **-help** that provides a summary of how to use the tool.

#### Compiling source code: **javac**

The **javac** compiler reads class and interface declarations written in the Java programming language and compiles these to Java byte code class files.

```
> javac PersonnelRegister.java
```

This command will compile the source code in the file named “**PersonnelRegister.java**”, and create byte code for all the classes and interfaces defined in this file. If the source



code contains the declaration of a class called `PersonnelRegister`, a byte code file named “`PersonnelRegister.class`” will be generated. If the given source code file uses other classes and/or interfaces that have not yet been compiled, the compiler will try to compile these as well. For example, if the source code in the file named “`PersonnelRegister.java`” uses a class called `Employee`, and this class has not yet been compiled, the `javac` compiler will attempt to compile a file named “`Employee.java`”.

Table G.1 shows some of the parameters accepted by the `javac` compiler.

**TABLE G.1 A subset of options for `javac` in JDK**

| Command line options        | Description                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|-----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-d directory</code>   | Places all generated class files in <i>directory</i> .                                                                                                                                                                                                                                                                                                                                                                                       |
| <code>-Xlint</code>         | Turns on all warnings that are recommended by Sun Microsystems, Inc. There are several similar options that only provide specific types of warnings during compilation. Use option <code>-X</code> to get a complete list of <code>-Xlint</code> options.                                                                                                                                                                                    |
| <code>-encoding utf8</code> | Source code files are usually interpreted using the default character encoding for the operating system. This may cause problems if your source code files are created using another encoding scheme. The UTF-8 encoding scheme is commonly used to support non-English characters in source code. If UTF-8 is not the default character encoding for the operating system you use, you can specify the character encoding with this option. |

The following command will compile the files named “`PersonnelRegister.java`” and “`Company.java`” using UTF-8 character encoding, and place the generated byte code files in the `.../.../my_classes` directory:

```
> javac -Xlint -encoding utf8 -d .../.../my_classes
 PersonnelRegister.java Company.java
```

The command must be executed as a single command, even though it is shown on two lines above.

### Running the program: `java`

The `java` interpreter is used to run Java applications. It starts the runtime environment of Java and loads the class specified on the command line:

```
> java -ea Company
```

This command loads the class `Company` from the byte code file named “`Company.class`”, and executes the `main()` method of the class. The interpreter will print an error message if a file named “`Company.class`” is not available, or if the `Company` class does not have a valid `main()` method.



Execution of assertion statements must be explicitly turned on using the option given in Table G.2.

**TABLE G.2** Option for turning on assertion checking

| Command line options     | Description                                                                                                                                                              |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -enableassertions or -ea | Turns on execution of <code>assert</code> statements in the program. This will cause the program to terminate if any of the assertions evaluates to <code>false</code> . |

### Generating documentation: `javadoc`

The `javadoc` tool reads declarations and documentation comments in the source code and generates HTML pages describing classes, interfaces and members:

```
> javadoc PersonnelRegister.java
```

This command will generate a file named “`PersonnelRegister.html`” with documentation for the class `PersonnelRegister` that is defined in the file named “`PersonnelRegister.java`”.

**TABLE G.3** A subset of options for `javadoc` in the JDK

| Command line options | Description                                                 |
|----------------------|-------------------------------------------------------------|
| -encoding utf8       | Allows non-English characters (as for <code>javac</code> ). |
| -d <i>directory</i>  | Places the generated documentation in <i>directory</i> .    |

The following command generates documentation for all source code files in the current working directory and places the documentation in the subdirectory `doc`. The source code files are assumed to be stored in the UTF-8 character encoding.

```
> javadoc -encoding utf8 -d doc *.java
```

Section 8.5 on page 224 shows an example of the use of the `javadoc` tool.

## G.3 Configuring the CLASSPATH

The tools in a standard installation of the JDK know where to find the byte code files for all classes in the Java standard library. The tools will also look for byte code files in the current working directory. In most cases this set-up will be sufficient to run the program.

However, sometimes it is necessary to instruct the JDK tools to look for byte code files in other directories as well. For examples, when we want to:

- Install additional libraries that must be made available to all Java programs.



- Start a compiled program that resides in a directory different from the current working directory.

By specifying a *class path* it is possible to augment the list of directories that are searched for byte code files. A class path can be specified by the `-classpath` option of the `java` tool, or by defining the environment variable `CLASSPATH`. Both techniques will work for all tools described in this appendix.

A class path consists of the paths to one or more directories where byte code files reside. Multiple paths are separated by the character ":" (colon) under Posix/Linux, and with ";" (semicolon) under Microsoft Windows.

Under Windows, the following commands will first compile the class `Company` in the file named "Company.java", placing the byte code files in the `C:\my_classes\` directory, and then run the program:

```
> javac -classpath C:\my_classes -d C:\my_classes Company.java
> java -ea -classpath C:\my_classes Company
```

The `-classpath` option specifies the path of the classes used by the `Company` class.

Libraries can be made available as Java archive (JAR) files. Such archives can contain a large number of byte code files. To get the Java JDK tools to look for classes in these archives, the full path to the archive must be specified in the class path.

A system administrator for a Unix system who wants to make the library `console.jar` available for all users of the system, for example, can do the following:

- 1 Place the archive file, "console.jar", in a shared directory, e.g. `/usr/share/java/`.
- 2 Set the environment variable `CLASSPATH` for all users to `/usr/share/java/console.jar`.

Environment variables are set in different ways under different operating systems. Check the documentation for the operating system you are using to find out how this is done. Further information on how classes are found, and how to configure the environment variable `CLASSPATH`, is available at:

<http://java.sun.com/j2se/1.5.0/docs/tooldocs/findingclasses.html>

# Introduction to UML

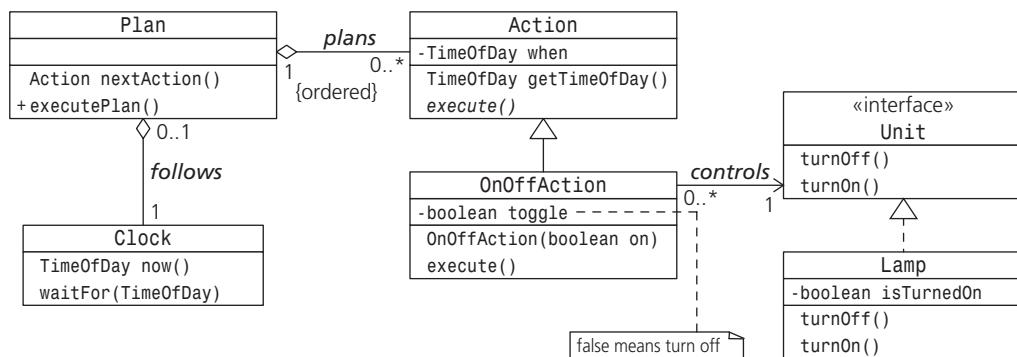
## INTRODUCTION

The *Unified Modelling Language* (UML) is a standard modelling language for object-oriented analysis and design. UML has a graphical notation that can be used to make diagrams that specify, visualise and document various aspects of object-oriented systems. This appendix gives a short introduction to basic elements in UML diagrams.

### H.1 Class diagram

A *class diagram* shows static structures in program design. Such diagrams can be used to illustrate the relationships between classes and interfaces. Figure H.1 shows a class diagram for a program that controls street lights in a city.

**FIGURE H.1** Class diagram



The rectangles describe *types* that are either classes or interfaces. A rectangle can consist of several compartments that describe different aspects of the type:

- 1 *Name*. The name of the type is always placed in the top compartment. In Figure H.1, the type **Unit** has also been tagged with the *stereotype* «interface», which specifies that **Unit** is an interface. The other types shown in the diagram are classes.



**2 Attributes.** The second compartment describes the attributes of a type. This corresponds to field variables in Java. This compartment can be left empty if you don't need to show the attributes of a class. The attributes of the `Plan` class are not shown in Figure H.1.

**3 Operations.** The third compartment describes the operations of a type. In Java, this corresponds to methods.

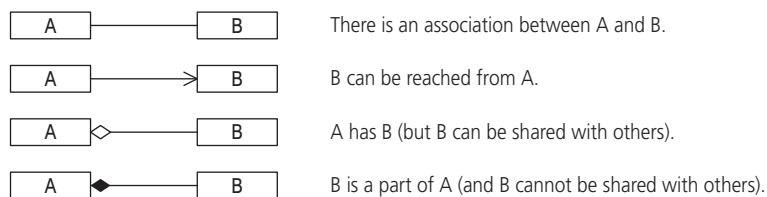
Attributes and operations can be marked with symbols that specify their visibility. The most important symbols are + (plus) and - (minus), which denote the `public` access modifier and the `private` access modifier, respectively.

The lines and arrows between the rectangles describe associations between the types. A triangular arrow with a solid line represents *inheritance*. In Figure H.1, the `OnOffAction` class inherits from the `Action` class. A triangular arrow with a dashed line represents implementation of an interface. In Figure H.1, the `Lamp` class implements the `Unit` interface.

Figure H.2 shows how different types of associations are depicted in UML. Associations can specify a *multiplicity* defining the number of objects at each end of the association. Table H.1 shows some typical multiplicities between types in associations. For example, the association between `Plan` and `Clock` in Figure H.1 shows that every `Plan` object has exactly one `Clock` object associated with it (multiplicity 1 at the association end connected to the `Clock` class), but that a `Clock` object need not be associated with a `Plan` object (multiplicity 0..1 at the association end connected to the `Plan` class).

A *note* can be included in a UML diagram, and can be attached to any element. Figure H.1 includes a note attached to the `toggle` field of the `OnOffAction` class, explaining how its value should be interpreted. *Constraints* on associations can be given in curly brackets, for example, the constraint `{ordered}` shown in Figure H.1.

**FIGURE H.2**      **Associations**



**TABLE H.1**      **Multiplicity**

| Multiplicity | Description                                                                                                                           |
|--------------|---------------------------------------------------------------------------------------------------------------------------------------|
| 1            | Only one object can fulfil the association.<br>A <code>Plan</code> object has exactly one <code>Clock</code> object.                  |
| 0..1         | Zero or one object can fulfil the association.<br>A <code>Clock</code> object can be associated with 0 or 1 <code>Plan</code> object. |

| Multiplicity | Description                                                                                         |
|--------------|-----------------------------------------------------------------------------------------------------|
| 0..*         | Any number of objects can fulfil the association.<br>A Plan object has zero or more Action objects. |
| $n$          | Only $n$ objects can fulfil the association, where $n > 1$ .                                        |
| 0.. $n$      | Up to and including $n$ objects can fulfil the association, where $n > 1$ .                         |

## H.2 Object diagrams

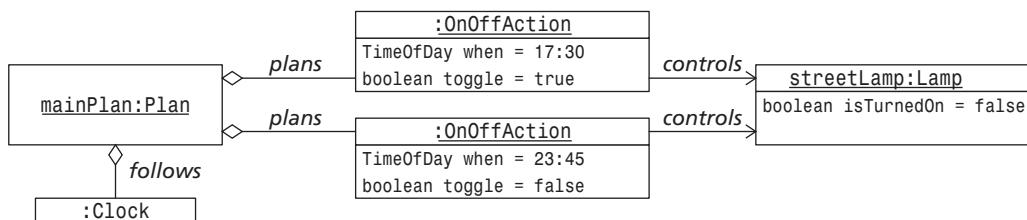
H



An *object diagram* shows the relationship between objects and their state. These diagrams can be used to show data structures and the state of a program at any given time during execution.

The notation for objects is similar to that used for classes, but the name of an object has the form *objectName:className* and can be underlined. The compartment for the attributes shows the state of the object, while the compartment for operations is seldom used in object diagrams. Figure H.3 shows a plan for turning street lights on at 17:30 and off at 23:45.

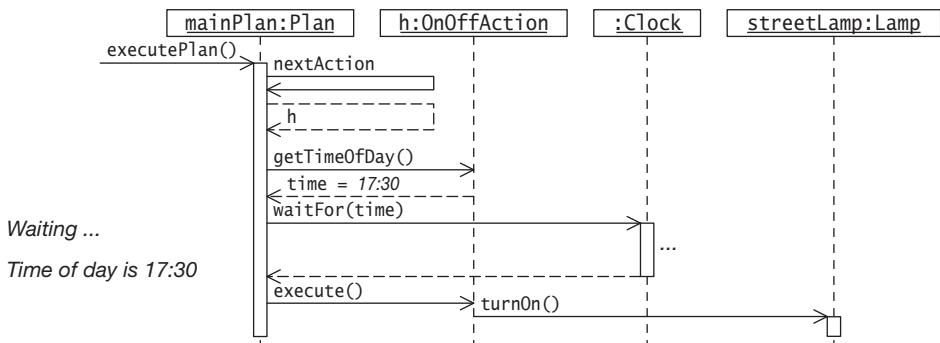
**FIGURE H.3** Object diagram



## H.3 Sequence diagrams

A *sequence diagram* shows the interaction between objects, i.e. how the objects communicate during program execution. These diagrams show the *lifelines* of a set of objects, and the method calls between them. Time flows downwards in sequence diagrams.

**FIGURE H.4** Sequence diagram



Method calls are shown by solid arrows, and return values as dashed arrows. The first method called in Figure H.4 is `executePlan()` in the `Plan` object referred to by the reference `mainPlan`. The method calls are shown in sequence from top to bottom. The objects at the top of the diagram have lifelines that extend downwards along the time axis. The narrow rectangles on the lifeline show time intervals when an object is *active*, i.e. executing a method. Supplementary notes are given to the left of the diagram. It is a good idea to provide such additional information in sequence diagrams.

## H.4 Activity diagrams

An *activity diagram* shows how an operation can be broken down into smaller steps, the order in which these steps (or *activities*) can be carried out, and alternative paths that can be followed to complete the operation. Figure H.5 shows an example of an activity diagram for managing street lights. This type of diagram can be seen as the object-oriented equivalent of the data flow diagrams that are often used in structured programming.

Each step is shown as a rectangle with rounded corners, and the action to be carried out is described using natural language. The transition from one step to the next is shown as a solid line arrow (or *activity edge*), which can be marked with a condition (or *guard*) that must be satisfied to allow the transition to take place. Such conditions are written in square brackets, `[condition]`, again using natural language (see Figure H.5).

At some point the execution can reach a *decision point* where different courses of action can be selected. A decision point is shown as a diamond in the UML diagram. Any number of activity edges can exit from a decision point, each edge marked with the condition that must be satisfied for the edge to be traversed. (An edge without a condition is always traversed once the activity it originates from has completed.)

The *starting point* of the operation is marked with a filled circle, while the *endpoint* is modelled by a filled circle enclosed in a border. All activity diagrams need a starting point, and it is common to place this at the top of the diagram in Western cultures, reflecting the normal reading direction from top to bottom. A diagram may have more than one endpoint. For continuous operations it may also make sense not to have an endpoint at all.

**FIGURE H.5** Activity diagram





## *Symbols*

^ bitwise exclusive OR 735  
 ^ logical exclusive OR 735  
 ^= assignment after bitwise exclusive OR 736  
 ^= assignment after logical exclusive OR 736  
 - subtraction 26, 735  
 -- decrement 734, 735  
 -= assignment after subtraction 736  
 ; statement terminator 20  
 ! negation 48, 735  
 != not equal to 735  
 ?: ternary conditional operator 735  
 . accessing class member 734  
 . dot character 740  
 ' single quote 82  
 " double quote 82  
 (*type name*) 342  
 [ ] 125, 734  
 { ... } block notation 52  
 @Test 407  
 \* asterisk 225  
 \* multiplication 26, 735  
 \* wildcard import 437  
 \* wildcard static import 438  
 \*= assignment after multiplication 736  
 / division 26, 735  
 // source code comment 21  
 /= assignment after division 736  
 \ 593  
 \ backslash 82  
 \n newline 82  
 \t tab 82  
 & bitwise AND 735  
 & logical AND 735  
 && conditional AND 48, 735  
 &= assignment after bitwise AND 736  
 &= assignment after logical AND 736  
 % modulus 26, 735  
 % percent sign 740, 742  
 %= assignment after modulus 736  
 + addition 19, 26, 735  
 + string concatenation 19, 84, 735  
 ++ increment 734, 735  
 += assignment after addition 736  
 < less than 245, 246, 735  
 <? extends T> 495  
 <? super T> 496  
 <?> 496  
 << bitwise left shift 735  
 <<= assignment after bitwise left shift 736

<= less than or equal to 236, 735  
 <?> 471  
 = assignment 21, 22, 46, 736  
 == equal to 46, 236, 237, 245, 246, 735  
 > greater than 245, 246, 735  
 >= greater than or equal to 236, 735  
 >> bitwise right shift with sign extension 735  
 >>= assignment after bitwise right shift with sign extension 736  
 >>> bitwise right shift with 0 extension 735  
 >>>= assignment after bitwise right shift with 0 extension 736  
 | bitwise OR 735  
 | logical OR 735  
 |= assignment after bitwise OR 736  
 |= assignment after logical OR 736  
 || conditional OR 48, 735  
 ~ bitwise complement 735

## *Numerics*

1's compliment 763

2's compliment 762, 763

## A

abstract 371, 374, 376, 733, 737  
 abstract class 370, 449  
 abstract data type (ADT) 464  
 abstract method 376, 737  
 Abstract Window Toolkit (AWT) 635  
 abstraction 76, 217  
 access modifier 332  
 access modifiers 442  
     class members 443  
     package members 442  
 accessibility modifier 737  
 accessing array elements 127  
 accurate execution 3  
 ActionEvent 661  
     source JButton 663  
     source JCheckBox 664  
     source JRadioButton 664  
     source JTextField 663  
 ActionListener 662  
     actionPerformed() 662  
 active methods 273  
     *see also* method execution  
 active object 774  
 activity diagram 774  
     activity 774  
     activity edge 774

alternative paths 774  
 decision point 774  
 endpoint 774  
 guard 774  
 starting point 774  
 transition between activities 774  
 actual parameter 167  
     array 171  
     reference value 171  
 actual result 409  
 actual type parameters  
     *see* parameterized types  
 adapter class 671  
 adapter classes 671  
 adding a series of integers 61  
 addition 26  
 additional parentheses 20  
 ADT  
     *see* abstract data type  
 aggregation 374  
 algorithm 224  
     analysing amount of work 241, 244,  
         249, 251, 254  
     binary search 253  
     insertion sort 242  
     linear search 251  
     selection sort 238  
     simulate by hand 264, 266  
 alias 89, 90, 91  
     arrays 128  
 alphabetical order 747  
 and 48  
 annotation 407  
 annotation type 407  
 annual interest rate 44, 71  
 anonymous array 129  
 anonymous classes 687, 690  
     extending a superclass 688  
     interface implementation 687  
 API 447  
 Application Programming Interface 447  
 area of a circle 42  
 argument 167  
 arithmetic expression 26  
     associativity rules 29  
     evaluation rules 28  
     left associativity 29  
     operand 26  
     operator 26  
     operator precedence 28  
     precedence rules 28  
     promoting operands 27

right-associativity 29  
**ArithmeticeException** 275  
**array** 124  
 [ ] 125, 734  
 accessing elements 127, 251  
 alias 128  
 anonymous 129  
 as actual parameter value 171  
 bounds 128  
 comparison 146  
 copying 145, 146  
 creating 125, 126, 136  
 declaration 126, 128  
 default initialization 125, 126  
 default value 125  
 dimension 136  
 element 124, 172  
 element type 124, 125  
 explicit initialization 129  
 index 124, 251, 550  
 iterating 132, 135, 144  
 iterating multidimensional 139  
 length 124, 125, 127  
 multidimensional 136, 143  
 new 125  
 of arrays 124, 138  
 of objects 124, 126  
 one-dimensional 136  
 out of bounds 128, 143  
 partially sorted 244  
 partially-filled 147  
 pass 238, 249  
 passing array 177  
 ragged 141  
 reference declaration 129  
 reference variable 125  
 returning array 177  
 segment 556  
 shifting values 242, 244  
 sorted 238  
 sorting 238  
 subarray 242  
 type 125, 172  
 unsorted part 238, 240, 249  
**ArrayIndexOutOfBoundsException** 128, 143, 573  
**ArrayList** 464  
**Arrays**  
 selected methods 499  
**Arrays** class 259  
 ASCII 745  
**assert** 64, 236, 252, 733, 769  
 assertion 64, 230, 769  
     controlling parameter values 332, 362  
     testing relational operators 236  
     validating user input 65  
**AssertionFailedError** 408  
 assigning to variable 12, 21, 46  
 association 220, 221, 772  
 constraining 772  
 one-to-many 221, 222  
 one-to-one 221  
 associativity rules 29, 734  
 asterisk 225  
 audio tape 9  
 auto-boxing 92  
 auto-unboxing 92  
 automated testing 397  
 automatic garbage collection 179  
**AWTEvent** 661

**B**

background colour 636  
 backslash 82, 83  
 backslash ( ) 593  
 bank account balance 44, 71  
 base 758, 759  
     *see also* numeral system  
 base 10 numeral system  
     *see* decimal numeral system  
 base 2 numeral system  
     *see* binary numeral system  
 base 8 numeral system  
     *see* octal numeral system  
 base case 546  
 base class 439, 449  
 basic class  
     *see* superclass  
 basic instruction 13  
 behaviour 207, 209, 330, 331  
     combining 208  
 binary file 294  
 binary files 592  
     end of file 600  
     reading of binary values 598, 599  
     writing of binary values 593  
 binary numeral system 758  
     converting to decimal 759  
     converting to hexadecimal 760  
     converting to octal 760  
 binary operator 26, 734  
 binary representation 586  
 binary search 253, 548  
     amount of work 254  
     implementation 254  
 binary search algorithm 548  
**binarySearch()** 259  
 bits 294  
 Blackjack 391  
 block 240  
 block statement 52, 57  
     { ... } 52  
     declaring new variable 53  
 body of method 13  
 book 9  
**Boolean**  
     literal 46  
     value 46, 236, 741  
**boolean** 46, 236, 733, 736

**Boolean** class 736  
**Boolean** expression 46  
     evaluation rules 49  
     short-circuit evaluation 49  
**BorderLayout** 639, 653  
     compass directions 653  
 borders 639  
 bottle of milk 9  
 bounds of array 128  
**break** statement 117, 733  
     in **for(;;)** loop 108  
     in **switch** statement 112  
 breaking down problem 8  
 broader data type 27  
 bubble sort 266  
 buffer 304, 465, 588  
**BufferedReader** 588  
**BufferedWriter** 588  
 building Java programs 4  
 button 645, 646, 649  
     checkbox 645  
     disabled 645  
     enabled 645  
     push button 645  
     radio button 645  
     selected 645  
     unselected 645  
**ButtonGroup** 645  
**byte** 236, 733, 736  
**Byte** class 736  
 byte code 6, 13, 14, 437, 454, 767  
     execution 13  
 byte input streams 586  
 byte output streams 586  
 bytes 586  
     byte input streams 586  
     byte output streams 586  
     selected 587  
 bytes 294

**C**

calculate weekly salary 52  
 call statement 13  
 call-by-value 169  
     consequences 170  
 calling methods 12, 213  
 carry bit 763  
 Cartesian coordinate system 69  
**case** label 733  
     common actions 116  
     constant expression 113  
     falling through 115  
**case sensitive, names**  
     case sensitive 25  
 casting  
     *see* type conversion  
**catch** 733  
**catch block**  
     *see try-catch statement*  
 cause of error 6

celestial body 565  
 cell 400, 656  
 Central Processing Unit 13  
 chaining *if-else* statement 57  
 change object state 211  
**char** 25, 82, 236, 733, 736  
 character 7  
     code number 81  
     comparing 83  
     count 13  
     literal 81  
     Unicode 81  
**Character** class 736  
 character encoding 294  
     UTF-8 768, 769  
 character streams 587  
     readers 587  
     selected 588  
     writers 587  
 checkbox 634, 645  
 checked exception 283, 571  
 checking if value is in interval 68  
 checking input 223  
 child  
     *see* subclass  
 choosing method declaration 224  
 circular list 528  
 circumference of  
     circle 42  
     Earth 44  
 class 5, 9, 76  
     . accessing class member 734  
     abstract class 370  
     abstract method 737  
     accessibility modifier 737  
     behaviour 76  
     concrete class 371  
     constructor 162  
     declaration 9, 76  
     diagram 78  
     enumerated type as member 196  
     field declaration 163  
     field variable 162  
     file 6  
     final class 349  
     fully qualified name 437  
     geometrical object 357  
     global constant 180  
     global variable 180  
     helper 442  
     implement a contract 378  
     implement an interface 378  
     instance member 162  
     instance method 162  
     member access 737  
     member accessibility 737  
     member declaration 162  
     modifier 737  
     name 5, 8, 9  
     overloaded method name 350  
     overview 162  
     predefined 161  
     properties 76  
     static member 162  
     static method 162  
     static variable 162  
     **String** 13  
     user-defined 161  
     visible member 332  
**class** 733  
 class declaration  
     *see* class  
 class diagram 771, 772  
     association 772  
     compartment 771  
     constraining association 772  
     multiplicity 772  
     note 772  
     stereotype 771  
     visibility 772  
 class file 437  
 class files 454  
 class libraries 436  
 class path 770  
**ClassNotFoundException** 610  
**CLASSPATH** 407, 770  
 clear purpose 229  
 clear responsibilities 217  
 close a file 298, 300  
 code  
     assumptions 230  
     collecting 213  
     deficiencies 208  
     duplication 209, 212  
     example 10  
     maintenance 224  
     number 81  
     point 81  
     restructuring 209  
     source - *see* source code  
 code re-use 436  
 code value 745  
**collator** class 747  
 collecting  
     code 213  
     related operations 213  
**collection** 475  
     basic operations 476  
         deletion 476  
         insertion 476  
     bulk operations 475  
         difference 477  
         intersection 477  
         subset 477  
         union 477  
 collection of classes 9  
**collections**  
     selected methods 498  
**collections** 464, 475  
     bulk operations 483  
     **Collection** 321, 475  
     insertion order 464  
     iterator 477  
     **List** 479  
     order 464  
     *see also* lists  
     *see also* sets  
     **Set** 482  
     sorted 464  
     standard string representation 479  
     traversing 477  
**Color** 637  
     colours 637  
**colours** 637, 638  
 combining  
     behaviour 208  
     related properties 208  
**Comma Separated Values** 300  
 command 6  
     **java** 7  
     **javac** 6  
     **javadoc** 228  
     line 6  
     syntax 6  
 command line 228  
 comment 10, 224, 240  
     Javadoc 230  
         multi-line 225  
         single line 225  
 common properties 9  
 communicating with  
     objects 213, 217, 219  
     user 216  
 company 219  
**Comparable** interface 245, 247, 249, 256, 259  
     **compareTo()** 245, 246  
**compareTo()** 245, 246, 256  
 comparing  
     characters 83  
     objects 245, 256  
     primitive values 237  
 comparison  
     better 209  
     good 216  
     ugly 208  
 compartment 771, 772  
 compass directions 653  
 compile time error 304, 307  
 compiler 4, 10, 224, 398  
     error 6  
 compiling 398  
     Java program 6, 18, 767, 768  
     program 6  
     source code 14  
**Component** 636  
 component hierarchy 635, 638

components 635, 636  
     background colour 637  
     foreground colour 637  
     inheritance hierarchy 636  
     preferred size 652  
     stretching 653  
 composite object 374  
 compound statement 52, 240  
     *see also* block statement  
 computer 3, 12, 13, 14  
     type 14  
 computing average of integers 69  
 concrete class 371  
 concrete instance 9  
 conditional  
     AND 48  
     loop 102  
     OR 48  
 confirmation dialog box 313  
 console I/O 749  
 const 734  
 constant 24  
     expression 113  
     naming conventions 24  
 constraint 772  
 constructor 162  
     default constructor 186  
     enumerated type 194  
     explicit default constructor 186  
     implicit default constructor 186  
     non-default constructor 187  
     overloaded 189  
     overloading 224  
 constructor call 78, 189  
 constructors 402  
 containers 635, 638  
     adding components 638  
     inheritance hierarchy 636  
     selected common methods 639  
 content pane 639  
 continue statement 733  
     in `for(;;)` loop 108  
 contract 375, 376  
     abstract method 376  
     extend another contract 379  
     implement a contract 378  
     in Java 376  
     inheritance between 379  
     naming convention 376  
     postcondition 375  
     precondition 375  
     reference 379  
     without methods 379  
 control components 636  
 control flow 51  
     execution 50  
     loop 51  
     selection 50  
 convergence 550  
 convergence towards base case 547

conversion  
     downcasting 342  
     of reference types 341  
     of superclass reference 341  
     upcasting 341  
 conversion code 34, 739, 740, 741, 743  
 conversion flag 739, 740, 742, 743  
 converting  
     between strings 95  
     objects 92  
     primitive values 92  
 cooperation between objects 213  
 copying arrays 146  
 Coq au vin 239  
 counter-controlled loop 102  
     `for(;;)` 102  
 CPU 13, 14  
 Craps 158, 391  
 creating arrays 126  
 CSV 300  
 cube 42, 70  
 currencies 204  
 currency conversions 44  
 current object 173  
     `this` 173, 733  
 cursor 616, 639  
 cylinder 70

**D**

data  
     duplication 212  
     field 295  
 data streams  
     *see* streams  
 data structure  
     dynamic 509  
     self referencing 509  
 data structures 123  
     arrays 124  
     dynamic 463  
     static 464  
 data type 20  
     broader data type 27  
     character 25  
     floating-point number 25  
     integer 25  
     narrower data type 27  
     numeric 25  
     primitive 25  
 database 257  
 DataInput 614  
 DataInputStream 586, 598, 602, 615  
 DataOutput 614  
 DataOutputStream 586, 593, 602, 615  
 De Morgan's laws 50, 67  
 decimal notation 741  
 decimal number 742  
 decimal numeral system 741, 757  
     converting to binary 761  
     converting to hexadecimal 762

converting to octal 762  
 decision point 774  
 declaration  
     array 126  
     array reference 129  
     class 9, 76  
     field variable 77  
     instance method 77  
     parameter 11  
     reference variable 77  
     variable in block 53  
 decrement operator (`--`) 101, 106  
     side effect 102  
 decrypt file 323  
 default  
     value of array 125  
     values 126  
 default accessibility 737  
 default constructor 186  
 default exception handler 275  
 default label 113, 733  
 default value 163  
 deficiencies in code 208  
 deficiencies in program 4  
 defining  
     class 9  
     method 11  
 delegating work 217  
 delegation 529  
 delimiter 320  
 denominator 29  
 derived class  
     *see* subclass  
 derived interface 379  
 describing  
     actions 9  
     tasks 4  
 deserialisation 602  
     *see also* object deserialisation  
 design 207  
     decisions 212  
 design by inheritance 331  
 desktop environment 5  
 destination 293  
 destructive operations 483  
 detecting deficiencies 4  
 development tools 6  
 diagram of classes 78  
 dialog box 308  
     confirmation 313  
     ending program 308  
     input 312  
     message 311  
     message type 310  
     modal 308  
     option type 311  
     user action 310  
 dialogue window 678  
     modal 679  
     non-modal 679

owner of 679  
 sequence diagram 682  
 terminating 685  
 dice roll 150  
 digit 82, 742, 745  
 dimension of array 136  
 directed acyclic graph 533  
*see* graph  
 directory hierarchy 454  
 display 505  
 dissecting source code 4  
 distributing responsibilities 212  
 divide and conquer 556  
 division 26  
     by zero 29  
     floating-point division 29  
     integer division 29  
 division of tasks 546  
**do** 733  
**do-while** statement 60, 63  
 document 3  
 document algorithms 240  
 documentation  
     generated 226  
     source code 212  
 documenting  
     class 225, 230  
     limitation 230  
     member 225  
     method 230  
     program 230  
     source code 224  
 dot notation 78  
**double** 25, 26, 27, 37, 236, 733, 736  
**Double** class 736  
 double linked list 528, 542  
 double quote 7, 82  
     string literals 83  
 downcasting 342  
 drawing program 3  
 drinking 9  
 duplicate work 562  
 duplication  
     code 209, 212  
     data 212  
     eliminating 210  
     minimizing 212  
 dynamic data structure 509  
     linked list 509  
     queue 528  
     stack 528  
 dynamic data structures 463  
     *see* lists  
     *see* maps  
     *see* sets  
 dynamic method lookup 365, 369  
 dynamic strings  
     *see* string builder

**E**  
 Earth circumference 44  
 Eclipse 411  
 edges  
     graph 531  
 editing source code 4, 5  
 egg carton 8  
 element  
     array 124  
     lookup 127  
     type 124  
 element-wise  
     comparison 146  
     copying 145  
 eliminating duplication 210  
**else** 733  
**else body** 54  
 employee  
     board member 378  
     comparison 208  
     hourly worker 362  
     manager 334, 362  
     piece worker 362  
     register 213  
 encapsulation 442  
 encrypt file 323  
 end of file 305  
 end users 3  
 end-of-line character 10  
 ending program 308  
 endpoint 774  
 enhanced **for** loop  
     *see* **for( : )** loop  
 entering  
     command 6  
     source code 4  
 entries  
     *see* maps  
 entry point 11  
 enum  
     *see* enumerated type  
**enum** 192, 733  
 enum constant  
     actual parameter 194  
 enum type  
     ordinal value 594  
 enumerated type 192  
     as class member 196  
     constructor 194  
     enum 192  
     general form 194  
     simple form 192  
 environment variable  
     CLASSPATH 770  
**EOFException** 600  
 epsilon value 238  
 equality  
     == 46  
     between objects 256  
     common mistake 46  
 equals() 245, 256  
 potential problem 237  
 reference 86, 245  
 value 86, 236  
**equals()** 245, 256, 332  
**Error** 569, 571  
 error reporting 405  
 error situations 271  
 errors 398  
     in source code 6  
     logical 272  
     programming 271  
 escaping characters 83  
 evaluation rules 49  
 event delegation model 634, 662, 663  
     event handlers 635  
     listener 635  
     source 635  
 event handlers 635  
 event handling  
     programming paradigms 665  
 event-driven programming 661  
 events 634, 661, 663  
     their sources 664  
**Exception** 569  
 exception 30, 272  
     checked 283, 571  
     exception handler 272  
     exception handling 272  
     handling 307  
     handling of several exceptions 573  
     propagation 275  
     scenarios 282  
     throw and catch 272  
     **throw** statement 284, 571  
     throwing programmatically 571  
     **throws** clause 284, 304  
     **try-catch** statement 276  
     unchecked 571  
 exception classes 569  
     defining new 576  
     partial inheritance hierarchy 570  
 exception handler 272  
 exception handling 272, 276, 278  
     programming errors 575  
     scenario 279, 280, 281  
     sequence diagram 279, 280, 282  
 exception propagation 272  
     sequence diagram 276  
 exceptions  
     unchecked 286  
 executable  
     form 14  
     program 4, 11  
 executing  
     byte code 7, 13  
     machine code 13  
     method 11  
     method body 13  
     program 10, 11

statement 11  
 execution  
     by machines 13  
     exception propagation 272  
     flow 13, 50  
     method execution 272  
 expected result 409  
 explicit array initialization 129  
 explicit default constructor 186  
 exponential growth 560, 562  
 expression 17, 167, 734  
     additional parentheses 20  
     arithmetic expression 26  
     associativity rules 734  
 extended assignment operator 100  
**extends** 331, 733  
 extension 5, 8  
 extract type descriptions 225

**F**

factor 544  
 factorial 544  
**false** 46, 342, 734, 736, 741  
 Fibonacci number 562  
 Fibonacci series 561  
 field 340  
     manipulating 213  
     name 163  
     reference value 218  
     shadowing 348  
     type 163  
 field declaration 163  
     *see also* field  
 field terminator character 594, 599  
 field variable 77  
     *see also* field  
**FIFO** (First in, first-out) 534  
 file 294  
     binary 294  
     close 298  
     CVS 300  
     decrypt 323  
     encrypt 323  
     end of file 305  
     format 300  
     handling 322  
     open 298, 304  
     read text line 305  
     reading 304  
     sequential 307  
     size 294  
     text 294, 298, 300  
     writing 298  
 file length 616, 617  
 file mode 616  
 file name extension 5, 8  
 file path 592  
     file path delimiter 593  
     one dot (.) 593  
     two dots (..) 593

file path delimiter 593  
 file pointer 616  
 file system 454  
**FileInputStream** 586, 610  
**FileNotFoundException** 598  
**FileOutputStream** 586, 594, 603  
**FileReader** 587  
 files  
     extending a file 625  
     file path 592  
     open 592  
     resetting 625  
**FileWriter** 587  
 filters 586  
**final** 24, 349, 374, 443, 733, 737  
 final class 349  
 final method 349  
**finally** 578, 600, 733  
     *see also* try-catch statement  
 fixed upper limit 223  
**float** 236, 733, 737  
**Float** class 737  
 floating-point division 29  
 floating-point number 37, 741, 742  
     hexadecimal 741  
 flow of control 51  
 flow of execution 13  
**FlowLayout** 638, 640, 653  
 focused responsibilities 213  
**for** 733  
**for(;;)** loop 102, 139  
     backwards 106  
     body 102  
     break statement in 108  
     condition 102  
     continue statement in 108  
     declaration 105  
     final value 102  
     header 102  
     infinite 110  
     initialization 102  
     initialization error 110  
     local variable 105  
     loop condition error 111  
     nested 106  
     one-off errors 110  
     optimizing 111  
     other increments 106  
     pre-test 104  
     start value 102  
     updating 102  
     while 103  
**for(:)** loop 143, 144  
     body 144  
     collection 144  
     element variable 144  
     multidimensional arrays 144

foreground colour 636  
 formal parameter 165  
 formal type parameters

*see generic methods*  
*see generic types*  
 format specification 31, 739, 742  
 format string 31, 739  
**Formattable** interface 741, 742  
 formatted document 5  
 formatted output 31  
     argument index 740  
     conversion code 34, 739, 740, 741, 743  
     conversion flag 739, 740, 742, 743  
     example of a bill 31  
     fixed field width 31  
     format specification 742  
     format string 31, 739  
     left-justified 32, 742  
     localizing output 31, 742  
     prefixing 742  
     right-justified 32, 742  
     separator character 742  
     width 742  
**formatTo()** 741  
 four-in-a-row 395  
 Four-in-a-row game  
     GUI development 691  
 frames 639  
 framework 439  
 frequency report 156  
 from peg 551  
 fulfilling requirements 208  
 fulfilling role 217  
 fully qualified name 437  
 function 544  
 functionality 399  
 further development 224

**G**

game of Nim  
     *see* Nim  
 game piece 404  
 garbage collection 221  
 gender statistics 213, 221  
 general case 546  
 generalisation 330  
 generated documentation 226  
 generic classes 471  
     *see generic types*  
 generic interfaces 473  
     *see generic types*  
 generic methods  
     declaration 496  
     method call 497  
     *see also* generic types

generic types 469  
     <? extends T> 495  
     <? super T> 496  
     <?> 496  
     compiling 474  
     formal type parameters 471  
     <T> 471

generic classes 471  
 generic interfaces 473  
 subtyping and wildcard ? 496  
 geometrical object 357  
`getMessage()` 569  
 global constant 180  
 global variable 180  
 good  
     abstractions 212  
     design 207  
     documentation 224  
`goto` 734  
 grading exam results 70  
 graph  
     directed 531  
     edges 531  
     nodes 531  
 graphical user interface 213, 307  
     *see GUI*  
 grid 656  
`GridLayout` 656  
 GUI 213, 307, 633  
     and `main()` method 643  
     avoiding component stretching 660  
     control components 644  
     designing layout 652  
     explicit termination of application 669  
     Four-in-a-row game 691  
     monitor 685  
     programming model 690  
     showing on the screen 636  
     steps in building 640  
 GUI dialog boxes 749, 752  
`GUIDialog` 633

**H**

Hanoi algorithm 554  
 hashing  
     *see maps*  
`HashMap` 464, 490  
     *see maps*  
`HashSet` 464, 482  
     *see sets*  
 head  
     *see linked list* 510  
 helper class 442  
 helper method 210, 406  
 hexadecimal 741  
 hexadecimal integer 741  
 hexadecimal number 742  
 hexadecimal numeral system 759  
     converting to binary 760  
     converting to decimal 759  
     converting to octal 760  
 hiding members 226  
 hierarchical file systems 437  
 high-level language 13  
 hill in New Zealand 16  
 histogram 157

hold intermediate results 12  
 holding value 12  
 HTML 226  
 human readable 3  
 HyperText Markup Language 226

**I**

icon 639  
 identifying role 208  
 IEEE-754 floating-point standard 762  
`if` 342, 733  
`if` statement 51, 54  
     body 51, 54  
     condition 51  
`if-else` statement 54  
     chaining 57  
     condition 54  
     *else* body 54  
     *if* body 54  
`IllegalArgumentException` 280  
 illustration 3  
 immutable strings 83  
`implements` 247, 377, 733  
 implicit default constructor 186  
`import` 733  
`import` statement 437  
 importing types 437  
 improving source code 4  
 income difference 220  
 increment operator `(++)` 101  
     side effect 102  
 indentation 10, 240  
     step 11  
 indexed variable  
     *see array*  
`IndexOutOfBoundsException` 88  
 indices 398  
 infinite loop 61, 104, 110  
 infinite nesting 547  
 infinite recursion 547, 554  
`Infinity` 29  
 inheritance 329, 361, 372, 772  
     abstract class 370  
     abstract method 376  
     abstract superclass 370  
     basic class 330  
     change of behaviour 331  
     change of properties 331  
     child 330  
     client use of inherited member 339  
     common behaviour 330  
     common behaviour through contract 379  
     common properties 330  
     concrete class 371  
     contract 376  
     derived class 330  
     downcasting 342  
     dynamic method lookup 365, 369  
     extend state 362

final class 349  
 final method 349  
 generalisation 330  
 instance method 335  
 interface 376  
*is-a* relationship 330  
 multiple inheritance from interfaces 378  
`object` class 331  
 of field 335  
 overloaded method name 350  
 overridden method 344, 362, 366, 368  
 parent class 330  
 polymorphic reference 365  
 polymorphism 365  
 refer to a field 340, 366  
 refer to an inherited field 367  
 reference to inherited instance  
     method 338  
 reference to inherited member 338  
 single inheritance 331  
 specialisation 330  
 subclass 330, 362  
 superclass 330, 335, 362  
 superclass contract 344, 369  
 superclass reference 340, 366  
 upcasting 341  
     using the `this` reference 338  
 visible member 362  
 inheritance hierarchy 330, 372, 635  
 inheritance relationship 330  
     transitive 369  
 initial setup 551  
 initial state 163  
 initializing  
     array 129  
     multidimensional arrays 136  
     variable 21  
 inner loop 63, 240, 243, 249  
 input 293  
     from keyboard 749  
     reading strategy 213  
     *via* GUI dialog box 312, 752  
 input data 38  
 input stream 586  
`InputStream` 586  
`InputStreamReader` 589  
 insertion sort 242  
     amount of work 244  
     implementation 243  
     pseudocode 242  
 instance 9  
     method 77, 162  
     *see also object*  
 instance member 162  
`instanceof` 342, 733, 735  
 instructions 3  
`int` 20, 25, 26, 27, 236, 254, 733, 737  
`Integer` 92, 245, 737

integer  
     hexadecimal 741  
     octal 741  
**Integer** class 92, 93  
 integer division 29, 272  
 integer representation 762  
 integrated development environments  
     411  
 interaction 213  
**interface** 376  
     abstract method 376  
     accessibility modifier 737  
     derived interface 379  
     extend another interface 379  
     implement an interface 378  
     in Java 376  
     inheritance between 379  
     interface type 379  
     modifier 737  
     naming convention 376  
     reference 379  
     subinterface 379  
     superinterface 379  
**interface** 733  
 interface abstraction 217  
 interface type 379  
 intermediate  
     representation 14  
     result 12  
 internal members 226  
 internal storage 12  
 interpreting  
     byte codes 13  
     errors 6  
 interval checking 68  
**intValue()** 94  
 invoice preparation 43  
 invoke methods 173  
*is-a* relationship 330  
 ISO 8859-1 294  
**Iterable** 527  
 iterating 59, 132, 135  
     multidimensional 139  
 iterative solution 544  
 iterative Towers of Hanoi 555  
**Iterator** 478  
     *see also* collections  
 iterator 527  
     *see* collections

**J**

**JAR**  
     *see* Java Archive file

**Java**  
     byte code 6, 13, 767  
     compiler 4  
     Development Kit 6  
     keyword 733  
     language 4, 9, 13  
     programmers 14

reserved word 733  
 source code 4  
 standard library 9, 13  
 tools 6  
     virtual machine 7, 13

**Java Archive file** 770

**java** command 7, 236, 262, 767, 768  
     options 769, 770

**Java Development Kit** 767

**Java Foundation Classes (JFC)** 635

**Java program**  
     compiling 18  
     running 18, 236, 262

**Java standard library** 35, 245, 259, 331,  
     349, 373

**java.awt** 635

**java.awt.event** 661

**java.io** 586

**java.lang** 373

**java.util** 475

**javac compiler** 6, 767  
     options 768

**Javadoc comment** 225, 230

**javadoc tool** 225, 228, 767, 769  
     options 769

**javax.swing** 635

**JButton** 644

**JCheckBox** 644

**JComponent** 637, 640

**JDialog** 678  
     inheritance hierarchy 679  
     selected methods 679

**JDK** 6, 225, 228  
     *see also* Java Development Kit

**JFrame** 638, 639  
     `BorderLayout` 639  
     selected methods 639

**JLabel** 637  
     labels 637

**journal** 9

**JPanel** 638, 640  
     `FlowLayout` manager 640

**JRadioButton** 644

**JTextField** 644

**JUnit** 439

**JVM** 13, 221

**K**

**key**  
     finding in an array 508

**key view**  
     of a map 491  
     *see also* maps

**keyboard** 35  
     reading from 749

**keys**  
     in a map 485  
     *see also* maps

**keyword**  
     *abstract* 371, 374, 376, 733, 737

**assert** 64, 236, 252, 733, 769

**boolean** 46, 92, 236, 733, 736

**break** 108, 733

**byte** 92, 236, 733, 736

**case** 112, 733

**catch** 276, 277, 733

**char** 25, 82, 92, 236, 733, 736

**class** 733

**const** 734

**continue** 108, 733

**default** 113, 733

**do** 60, 63, 733

**double** 25, 26, 27, 37, 92, 236, 733,  
     736

**else** 54, 733

**enum** 192, 733

**extends** 331, 733

**false** 46, 342, 734, 736, 741

**final** 24, 349, 374, 443, 733, 737

**finally** 578, 733

**float** 92, 236, 733, 737

**for** 102, 733

**goto** 734

**if** 51, 54, 342, 733

**implements** 247, 377, 733

**import** 35, 437, 733

**instanceof** 342, 733, 735

**int** 20, 25, 26, 27, 92, 236, 254,  
     733

**int class** 737

**interface** 733

**long** 92, 236, 733, 737

**native** 733

**new** 78, 733, 735

**null** 90, 734, 741

**package** 439, 733

**private** 226, 733, 738, 772

**protected** 443, 733, 737

**public** 5, 11, 226, 247, 442, 733,  
     737, 772

**return** 174, 733

**short** 92, 236, 733, 737

**static** 11, 733

**strictfp** 733

**super** 346, 348, 733

**switch** 112, 733

**synchronized** 733

**this** 173, 338, 733

**throw** 284, 571, 733

**throws** 284, 733

**transient** 733

**true** 46, 342, 734, 736, 741

**try** 276, 277, 733

**void** 11, 165, 733

**volatile** 733

**while** 60, 62, 63, 103, 733

**keywords in pseudocode** 240

**kitchen** 9

## L

labels 637  
language  
    construct 3, 5, 8, 9, 11, 13  
    structure 9  
    syntax 9  
large program 11  
Latin-1 294  
layout managers 635, 638, 653  
    associating with container 638  
layout of source code 10  
least significant bit 764  
left associative operator 734  
left associativity 29  
left margin 10  
left-justified output 32, 742  
lending date 9  
length of  
    array 124  
    string 7, 13, 87  
letter  
    lowercase 5  
    uppercase 24  
lexicographical order 85, 236, 245, 262, 263, 747  
library  
    loan 9  
    user 9  
lifeline 773  
LIFO (Last in, first out) 528  
limits of search area 550  
line  
    command 6  
    printing 12  
line terminator string 298, 741  
line-by-line documentation 228  
linear search 251  
    amount of work 251  
    extended version 267  
    implementation 251  
    pseudocode 251  
linked list 509  
    head 510  
    iterator 527  
    tail 510  
LinkedList 535  
List 480  
listener 635, 662  
    adapter classes 671  
    anonymous classes 687  
    external 665  
    main window as 672  
    registering with source 662  
    several sources, one listener 672, 675  
    terminating application 670  
listener interface 662, 664  
listener registration  
    sequence diagram 666  
lists 479

## A

ArrayList 479  
circular 528  
double linked list 528  
List 479  
merging 506  
ordered 538  
simple linked list 509  
use of 480  
literal 17  
    character 81  
    false 342  
    string 19, 83  
    true 342  
litmus test 331  
loan 9  
local variable 20, 105, 175, 208, 217  
    initialization required 22  
locale 31, 88  
Locale class 747  
localised output 742  
logarithm 550  
logical error 23, 272  
    infinite loop 61  
logical operator 48  
    ! 48  
    && 48  
    || 48  
    associativity rules 49  
    precedence rules 48  
logical programming error 547  
long 236, 733, 737  
Long class 737  
longValue() 94  
look up employees 224  
loop 50, 59, 544  
    backwards 106  
    body 51, 59, 60, 102  
    break statement 108  
    condition 51, 60, 102  
    conditional 102  
    continue statement 108  
    counter-controlled 102  
    debugging 112  
    for(;;) 103, 139  
    for(:) loop 143  
    header 102  
    infinite 61, 104, 110  
    initialization 102  
    inner 63  
    iteration 59  
    local variable 105  
    nested 63, 106, 139, 240, 243, 249  
    optimizing 111  
    outer 63  
    post-test 60  
    pre-test 60, 104  
    sentinel value 61  
    while 103  
lottery game 268  
lotto row 389

## low-level

language 13  
machine code 14  
lower limit 550  
lowercase letters 5, 8, 82, 745

## M

machine code 14  
    instructions 13  
    level 14  
main() 8, 9, 11, 19, 768  
    program arguments 184  
main() method  
    and GUI 643  
maintaining references 220  
manager 218  
managing several objects 222  
manipulating field variables 213  
Map 490  
map view  
    see maps  
maps 485  
    basic operations 490  
    entries 485  
    hashing 486  
    HashMap 485, 489, 490  
    key 485  
    key view 491  
    keys  
    Map 490  
    map view 491  
    use 492  
    value 485  
    value view 491  
    view operations 491  
    views 491  
markup tag 225, 228  
Master Mind 158  
mathematics 544  
mean value 631  
meaningful variable names 229  
member  
    static 212  
memory 12  
menus 639  
merging sequences 321  
message dialog box 311  
method 9, 12  
    abstract method 376  
    body 13  
    call 12, 13, 78, 213, 224  
    declaration 9  
    dynamic method lookup 365, 369  
    execution 11  
    final method 349  
    helper 210  
    implementation 80  
    name 9, 11, 12  
    overloaded method name 350  
    overloading 223

overridden method 344, 366, 368  
same name 223  
signature 365  
static method 332  
versatility 218  
method call 167, 543  
    actual parameter 167  
    argument 167  
    call-by-value 169  
    invoke methods 173  
    parameter passing 168  
method declaration 164  
    formal parameter 165  
    method name 164  
    non-void method 165  
    parameter passing 168  
    return type 164  
    returning array 177  
    signature 165  
    void method 165  
method execution 174, 176  
    active methods 273  
    exception propagation 272  
    sequence diagram 274  
    stack frame 272  
    stack trace 273  
method name 164  
methods  
    mutator 512  
    selector 512  
milk bottle 9  
minimizing duplication 212  
modal 679  
modal dialog box 308  
modern computers 3  
modifier  
    private 226  
    public 226  
modulus 26  
Morse code 507  
most significant bit 764  
multi-line comment 225  
multi-selection statement 112  
multidimensional array 136, 143  
multiple inheritance  
    interfaces 378  
multiple-choice quiz 158  
multiplication 26  
multiplicity 221, 772  
mutator 512

**N**  
naive methods 399  
naive solution 208  
name overloading 223  
named location 12  
naming  
    conventions 24  
    rules 5, 24  
    source code files 5

NaN  
    *see* Not a Number  
narrower data type 27  
native 733  
natural order  
    objects 236, 246, 263  
    primitive values 236  
nested loop 63, 139, 240, 243, 249  
    inner 63, 240, 243, 249  
    outer 63, 240, 243, 249  
nested selection statement 56  
nesting 546  
nesting language constructs 10  
new 78, 125, 733, 735  
New Zealand 16  
newline 82, 742  
nextDouble() 35, 37, 39  
nextInt() 35, 36, 38  
nextLine() 39  
Nim 357, 391  
    basic player 357  
    computer player 359  
    real player 359  
    round 358  
nodes 509  
    graph 531  
    self referencing 509  
non-default constructor 187  
non-modal 679  
non-obvious program aspects 228  
non-tangible concepts 9  
normal execution 272  
Norwegian 747  
Not a Number  
note 772  
null 90, 131, 734, 741  
NullPointerException 90, 131  
number  
    decimal 742  
    floating-point 742  
    hexadecimal 742  
    octal 742  
Number class 373  
number of characters 4  
number representation 757  
number sequence 561  
NumberFormatException 571, 573  
numeral system 757, 758  
    binary numeral system 758  
    conversions between decimal, octal  
        and hexadecimal numeral systems  
        760, 761  
    decimal numeral system 741, 757  
    hexadecimal numeral system 759  
    octal numeral system 759  
    string representation of integers 764  
numerator 29  
numerical data type 25, 236  
numerical value 12  
    printing 20

**O**  
object 8, 162  
    == 90  
    automatic garbage collection 179  
    availability 221  
    communication 217, 219  
    comparing 90, 245  
    comparing type 735  
    composite object 374  
    creating 78  
    current object 173  
    equals() 90  
    initial state 163  
    initialise state 334  
    interaction 213  
    model 75  
    natural order 236, 245  
    ownership 220  
    part object 374  
    primitive values as 92  
    property 9  
    relationships 218  
    set state 216  
    state 80, 163, 400  
    string 13  
    type 9  
Object class 247, 331, 334, 335, 349  
object diagram 773  
    compartment 773  
    state of object 773  
object hierarchy 603  
object input stream 602  
object output stream 602  
object serialisation 603  
    avoid duplication 612  
    object input stream 602  
    object output stream 602  
    Object reference 610  
    reading binary values 611  
    reading objects 610, 611  
    Serializable 602  
    writing binary values 604  
    writing objects 603, 604  
object-based programming 9  
object-oriented programming 329, 361  
ObjectInputStream 586, 602, 610  
ObjectOutputStream 586, 602  
objects  
    object serialisation 610  
objects in arrays 124, 126  
OBP 9  
    *see* object-based programming  
octal  
    integer 741  
    number 742  
octal numeral system 759  
    converting to binary 760  
    converting to decimal 759  
    converting to hexadecimal 760  
off-the-shelf programs 3

omelette 8  
 one dot (.) 593  
 one statement per line 10  
 one-dimensional arrays 136  
 one-off errors 110  
 one-to-many association 221, 222  
 one-to-one association 221  
 OOP  
     *see* object-oriented programming  
 open a file 298, 299, 304  
 operating system 5, 6  
 operation 8, 9  
 operator  
     ^ bitwise exclusive OR 735  
     ^ logical exclusive OR 735  
     ^= assignment after bitwise exclusive OR 736  
     ^= assignment after logical exclusive OR 736  
     - subtraction 26, 735  
     -- decrement 734, 735  
     -= assignment after subtraction 736  
     ! negation 735  
     != not equal to 735  
     ?: ternary conditional operator 735  
     \* multiplication 735  
     \*= assignment after multiplication 736  
     / division 735  
     /= assignment after division 736  
     & bitwise AND 735  
     & logical AND 735  
     && conditional AND 735  
     &= assignment after bitwise AND 736  
     &= assignment after logical AND 736  
     % modulus (remainder) 735  
     %& assignment after modulus 736  
     + addition 735  
     ++ increment 734, 735  
     += assignment after addition 736  
     < less than 245, 246, 735  
     << bitwise left shift 735  
     <<= assignment after bitwise left shift 736  
     <= less than or equal to 236, 735  
     = assignment 736  
     == equal to 236, 237, 245, 246, 735  
     > greater than 245, 246, 735  
     >= greater than or equal to 236, 735  
     >> bitwise right shift with sign extension 735  
     >>= assignment after bitwise right shift with sign extension 736  
     >>> bitwise right shift with 0 extension 735  
     >>>= assignment after bitwise right shift with 0 extension 736  
     | bitwise OR 735

| logical OR 735  
 |= assignment after bitwise OR 736  
 |= assignment after logical OR 736  
 || conditional OR 735  
 ~ bitwise complement 735  
 arithmetic operator 26  
 binary operator 26, 734  
 cast operator (*type name*) 342  
 instanceof 342  
 left-associative 734  
 precedence 734  
 relational operator 46, 236  
 right-associative 734  
 ternary operator 734  
 unary minus 27  
 unary operator 27, 734  
 unary plus 27  
 option type 313  
 order  
     ascending 238  
     descending 238  
     natural 236  
 ordered list 538  
 ordinal value 594  
 out of scope 53  
 outer loop 63, 240, 243, 249  
 output 12, 293  
 output stream 586  
 OutputStream 586  
 overloaded 189  
 overloaded method name 350  
 overloading method names 223  
 overridden method 344, 366, 368  
 overwriting variables 22  
 owner of objects 220

**P**

package 437  
     compiling 454  
     declaring 438, 439  
     default accessibility 737  
     hierarchy 454  
     importing 437  
     nesting 437  
     public accessibility 737  
     running program 454  
     static import 437  
 package 439, 733  
 package accessibility 737  
 package hierarchy 454  
 pad character 617  
     *see* records  
 palindrome 567  
 panels 636, 640, 646  
 parameter 11, 12, 78, 546  
     declaration 11  
     list 223  
     reference value 211  
     type 224  
 parameter list 164

parameter passing 168  
 parameterized types 472  
     actual type parameters 472  
     *see* generic types  
 parent class  
     *see* superclass  
 parentheses 20  
 parseInt() 94  
 part object 374  
 partially-filled array 147  
 partitioning 560  
 partitioning a task 560  
 passing parameters 12  
 pattern 552  
 pegs 551  
 percent sign 740, 742  
 personal information 9  
 personnel register 221  
 physical machine 13  
 pivot element 556  
 plain text 5  
 platform independence 14  
 play a role 207  
 polymorphic reference 365  
 polymorphism 365  
 positional value 757  
     *see also* numeral system  
**POSITIVE\_INFINITY** 272  
 post-decrement operator (x--) 101, 734  
 post-increment operator (x++) 101, 734  
 post-test loop 60  
 postcondition  
     method postcondition 375  
 pre-decrement operator (--) 101, 735  
 pre-increment operator (++) 101, 735  
 pre-test loop 51, 60, 104  
 precedence 734  
 precedence rules 28  
 precondition  
     method precondition 375  
 predefined class 161  
 preferred size 652  
 primary class 5, 11  
 prime number 548  
 primitive data type 25, 736  
     boolean 46, 236  
     character 236  
     comparing 236, 237  
     natural order 236  
     numerical 236  
 primitive value 95  
     as objects 92  
 principle amount 44, 71  
 print  
     report 208  
     values to file 299  
 print() 17, 36  
 printArray() 260  
 printf() 31  
 printing

boolean values 48  
 formatted output 31  
 numerical values 20  
 strings 19, 20  
 text 12  
     to terminal window 12, 17, 31  
**println()** 17, 19  
**printStackTrace()** 569  
**PrintStream** 588, 589  
**PrintWriter** 587  
**private** 226, 733, 738, 772  
 problem  
     dividing into parts 543  
     solving 543  
 program 3, 11  
     code 9  
     control flow 51  
     entry point 11  
     execution 10, 11, 12  
     form 3  
     large 11  
     logic 57  
     output 12  
     representation 14  
     robust 283  
     small 11  
     testing 4  
     translation 4  
     validation using assertions 64  
 program argument  
     **main()** 184  
 program design 207  
 program stack 554  
 programmer 14  
 programming 3, 14  
     contract 375  
     errors 14, 271  
     event-driven 635  
     language 3, 8, 14  
 programming error 547  
 promoting operands 27  
 prompt 35  
 proper indentation 11  
 properties 330, 331  
 property 9, 207, 209  
     combining 208  
**protected** 443, 733, 737  
 proverb 4  
 provision-based salary 42  
 pseudo-random numbers 149  
     distance 150  
     offset 149  
     range 149  
 pseudocode 224  
     as comments 240  
     document algorithms 240  
     implementation 240, 243  
     indentation 240  
     insertion sort 242  
     keywords 240

linear search 251  
 selection sort 239, 264, 266  
**public** 5, 11, 226, 247, 442, 733, 737, 772  
 push button 645

**Q**

**Queue** 535  
**queue** 528, 534, 536  
     FIFO 534  
     operations 534  
**Quicksort** 556, 563  
     partitioning 560  
     pivot 556

**R**

radio button 645  
 radius of circle 42  
 ragged array 141  
 random access file 614  
     cursor 616  
     file length 616, 617  
     file mode 616  
     file pointer 616  
     pad character 617  
     records 618  
     selected 615  
**Random** class  
     **nextInt()** 149  
     *see also* pseudo-random numbers  
 random numbers  
     *see* pseudo-random numbers  
**RandomAccessFile** 614  
 ranking two values 236  
 read a text line 305  
 readers 587  
 reading  
     from text file 304, 306  
     record 306  
 reading input  
     error handling 38  
     floating-point numbers 37  
     from keyboard 35  
     multiple values per line 38  
     prompt 35  
     reading integers 36  
     skipping rest of line 39  
     syntactic validation 38  
**readObject()** 610  
 recompiling 14  
 record 295, 320  
     field 295  
     reading from file 306  
     using **Scanner** 320  
     writing to file 300  
 records 617, 618, 625  
     field terminator character 594, 599  
     file length 617  
     pad character 617  
 recursion 543

base case 546  
 convergence towards base case 547  
 division of tasks 546  
 general case 546  
 infinite recursion 547  
 method call 543  
 nesting 546  
 recursion depth 547, 550  
 recursive method 544  
 subtasks 546  
 tail recursion 551  
 recursion depth 547, 550  
 recursion level 554, 560  
 recursive algorithm 546, 552  
 recursive binary search 548  
 recursive definition 544  
 recursive method 544  
 recursive method call 560  
 refactoring 217, 420  
     for clarity 228  
 reference  
     assigning to 89  
     declaration 77  
     equality 86, 245  
     type 77, 89  
     value 77, 125  
     variable 77  
 reference conversion  
     downcasting 342  
     sub-to-super 341  
     super-to-sub 341, 342  
     type control 342  
 reference to abstract class 372  
 reference type  
     conversion 341  
 reference value 168  
     as actual parameter 171  
     as parameter 211  
 referring to variable 12  
 refrigerator 8  
 register of employees 213  
 regression 402  
 related behaviour 209  
 related properties 209  
 relational operator 46, 236  
     character 236  
     numerical data type 236  
     precedence rules 46  
 relationships  
     between classes and interfaces 771  
     between objects 218  
 reliable execution 3  
 repeating code 210  
 report 208  
 required initialization 22  
 responsibility 207, 212  
     clear 217  
     focus 213  
     splitting sensibly 218  
 restructuring

code 209  
 program 217  
**return** 733  
**return statement** 174  
 expression 176  
**return type** 164, 176  
**return value** 176  
 reuse of source code 330  
 right-associative operator 734  
 right-associativity 29  
 right-justified output 32, 742  
 ring 551  
 risk of errors 14  
 robust program 283  
 role 207  
 roles 396  
 root container 636, 638, 639  
     content pane 639  
 rules for Towers of Hanoi 551  
 running Java program 4, 7, 18  
 runtime behaviour 278  
 runtime environment 7  
 runtime errors 271  
**RuntimeException** 569

**S**

**Scanner** class 35, 38, 39, 61, 87, 320  
     delimiter 320  
 scientific notation 741, 742  
 scope 53  
     out of scope 53  
**Scrabble** 267  
 search key 251, 253  
 searching  
     binary search 253  
     linear search 251  
     search key 251, 253  
 searching for class files 770  
 selection sort 238  
     amount of work 241, 249  
     implementation 240, 249  
     pseudocode 239, 264, 266  
 selection statement 50, 51  
     condition 51, 54  
 selector 512  
 self-explanatory code 229  
 sentinel value 61  
 separator character 742  
 sequence  
     *see lists*  
 sequence diagram 773  
     active object 774  
     lifeline 773  
     method call 773  
     supplementary notes 774  
 sequence of  
     actions 8  
     characters 12  
     instructions 3  
     statements 9  
 sequential access 614  
 sequential file 307  
 sequential streams 586  
 serialisation  
     *see object serialisation*  
**Serializable** 602  
**Set** 482  
 set object state 216  
 set of instructions 3  
 set theory  
     *see sets*  
 sets 482  
     destructive operations 483  
     difference 482  
     `HashSet` 482  
     intersection 482  
     operations 482  
     `Set` 482  
     union 482  
     use 482  
 shadowed field 348  
     super 348  
 shadowing fields 174  
 shifting values 242  
**short** 236, 733, 737  
**short class** 737  
 short names 6  
 short-circuit evaluation 49  
 showing nesting 10  
 side effect 65  
 signature 365, 376  
     of a constructor 189  
     of a constructor call 189  
     of a method 165, 172  
     of a method call 167  
 simple linked list  
     *see linked list* 509  
 simple selection statement 51  
 single inheritance 331  
 single quote 82  
 single-line comment 225  
 sink 293  
 size of array 124  
 small program 11  
 software 3  
**sort()** 259  
 sorted array 238, 550  
 sorted elements 548  
 sorted subarray 242  
 sorting  
     array of integers 238  
     array of objects 245  
     bubble sort 266  
     insertion sort 242  
     selection sort 238  
     sorting strings 557  
     sorting task 556  
     source 293, 635, 661  
     registering listeners 662  
     source code 3, 4, 8

// comment 21  
 comment 224  
 comments 10  
 documentation 212  
 file 4, 5  
 indentation 10  
 layout 10  
     simple program template 19  
 space character 7, 10, 11, 82, 742  
 spaghetti code 99  
 special character 10  
 specialisation 330  
 specific role 207  
 speed improvements 14  
 sphere 70  
 split tasks 218  
 splitting  
     responsibility 218  
     tasks 212  
 splitting source code 11  
 spreadsheet 3  
**Stack** 529  
 stack 528, 531  
     LIFO 528  
     operations 529  
     top of stack 529  
 stack frame 272, 554  
     *see also* method execution  
 stack overflow 554  
 stack trace 273, 274  
 stacking 551  
 standard deviation 631  
 standard error 589  
 standard in 35, 588  
 standard out 18, 588  
 starting JVM 7  
 starting point 774  
 state 163, 400  
 statement 9, 11  
     **assert** statement 64  
     block 52, 57, 102  
     **break** 99, 108, 110, 112, 114, 115,  
         117, 733  
     compound 52, 240  
     **continue** 99, 108, 110, 733  
     **do-while** 60, 63  
     execution 11  
     **for(;;)** 102  
     **for(:)** 143  
     **if** 51  
     **if-else** 54  
     loop 50, 59  
     loop statement 51  
     nested selection 56  
     **return** 174, 175, 176, 733  
     selection 50  
     simple selection 51  
     **switch** 112  
     **throw** 571  
     **try-catch** 276

**w**  
 while 60  
**s**  
 static 11, 733  
 static data structures 464  
 static import 437, 438  
 static member 162, 212  
     accessing 182  
 static method 162, 332  
 static variable 162, 332  
     initializing 183  
 statistics 213, 221  
 stderr 589  
 stdin 588  
 stdout 588  
 stereotype 771  
 storing value in variable 12  
 stove 8  
 streams 586  
     byte streams 586  
     character streams 587  
     close 594, 599, 603  
**strictfp** 733  
 string 12, 19, 81  
     + 19, 84  
     **compareTo()** 86, 87  
     comparison 85, 245  
     concatenation 19, 83, 735  
     converting primitive 87  
     creating 85  
     **equals()** 87  
     escaping characters 83  
     immutable 83, 89  
     index out of bounds 88  
     length 7, 13, 87  
     **length()** 87  
     lexicographical order 85, 236, 245,  
         262, 263  
     literal 19, 83  
     natural order 236  
     object 13  
     printing 20  
     reference equality 86  
     **String** class 81  
     value equality 86  
**string builder** 465  
     capacity 465  
     creation 465  
     modifying content 466  
     selected methods from **String-**  
         **Builder** 467  
     size 465  
**String** class 11, 13, 236, 245, 249, 741  
     *see also* string  
 string representation of integers 764  
**StringBuffer** 467  
     *see* **StringBuilder**  
**StringBuilder**  
     *see* string builder  
     *see* string builder 312 464  
**strings**  
     dynamic 465

subarray 242  
     sorted 242  
**subclass** 330, 362  
     convert super-to-sub reference 341  
     inheritance of field 335  
     overridden method 344  
     reference conversion 341  
     reference to inherited instance  
         method 338  
     reference to inherited member 338  
     super 346  
     **super()** 334  
     superclass constructor 334  
         superclass contract 344  
**subinterface** 379  
**subordinate** 218  
**subtasks** 546, 554  
**subtraction** 26  
**subtyping**  
     *see also* generic types  
**Sun Microsystems** 6  
**super** 346, 348, 733  
**super()** 334  
**superclass** 330, 335, 362  
     super 346  
**superclass reference** 340, 366  
**superinterface** 379  
**supertype** 380  
**supplementary notes** 774  
**swapping values** 238, 240, 249  
**Swing** 635  
**switch statement** 112, 733  
     block 112  
     **break statement** in 112  
     **case label** values 113  
     **case labels** 112  
     common actions 116  
     **default label** 113  
     expression 112  
     falling through 115  
**synchronized** 733  
**syntactic validation** 38  
**syntax** 6, 9  
**System.err** 589  
**System.in** 588, 589  
**System.out** 588  
**System.out object** 12, 18, 31, 48

**T**

**tab** 82  
**tail**  
     *see* linked list 510  
**tail recursion** 551  
**tangible items** 9  
**target peg** 551  
**task**  
     splitting 212, 218  
**temperature conversion** 41  
**temporary peg** 552  
**terminal window** 6, 12, 17, 36, 208, 213,

588, 749  
 reading from 589, 590  
 writing to 589  
**ternary operator** 734  
**test class** 407, 411  
**testing**  
     actual result 409  
     automated testing 397  
     expected result 409  
     regression 402  
     test class 407  
**testing program** 4  
**testing technique**  
     printing values 48  
     using assertions 65  
**text**  
     editor 5  
     files 3  
     formatting 5  
     output 12  
     printing 12  
     *see also* string  
**text field** 644  
     selected constructors 645  
     selected methods 645  
**text file** 294, 298, 300  
**text line** 298  
**text representation** 587  
**text string**  
     *see* string  
**text user interface** 213  
**text-based user interface** 411  
**the**  
     bad 223  
     good 207, 212, 216, 224  
     ugly 208  
**this** 173, 338, 733  
**thread** 308  
**throw** 733  
**throw statement** 284, 571  
**Throwable** 569  
**throws** 733  
**throws clause** 284, 304  
**tools for Java** 6  
**top-level windows** 639  
**top-of-stack (tos)**  
     *see* stack  
**toString()** 94, 247, 332, 741  
**Towers of Hanoi** 551  
     from peg 551  
     iterative 555  
     target peg 551  
     temporary peg 552  
**transient** 733  
**transitive** 369  
**translating** 4  
     class 6  
     source code 14  
**true** 46, 342, 734, 736, 741  
**truncating floating-point values** 28

try 733  
 try block  
     *see* try-catch statement  
 try-catch statement 276  
     activity diagram 277  
     form 277  
     normal execution 272  
     scenarios 277  
     *see also* exception handling  
 two dots (...) 593  
 type 9  
     annotation 407  
     array 125  
     conversion 27  
     element 124  
     name 77  
     *see also* data type  
 type conversion 27  
     explicit conversion 27  
     implicit conversion 27  
 type of  
     computer 14  
     object 8  
     value 12

**U**

UML 454  
     *see* Unified Modelling Language  
 UML diagram 224  
     *see* UML notation  
 UML notation  
     interface 377  
 unary operator  
     minus 27  
     operator 27, 734  
     plus 27  
 unchecked exception 571  
 unchecked exceptions 286  
 understanding program 208  
 Unicode 245, 589, 626, 745  
 Unicode character 81, 741  
 Unified Modelling Language 771, 772  
     activity diagram 774  
     class diagram 771  
     note 772  
     object diagram 773  
     sequence diagram 773  
     stereotype 771  
 upcasting 341  
 upper limit 223, 550  
 uppercase letters 5, 8, 24, 82, 745  
 user 3  
     dialog 61

user communication 216  
 user interface  
     graphical 213  
     text based 213  
 user manual 224  
 user-defined class 9, 161  
 using  
     computers 3  
     indentation 10  
     variables 12  
 UTF-16 592  
 UTF-8 294, 768, 769

**V**

validate program behaviour 64  
 validation using assertions 64, 769  
 value 12  
     comparing 236  
     epsilon 238  
     equality 86, 236  
     intervals 68  
     ranking 236  
     shifting 242  
     swapping 238, 240, 249  
 value view  
     of a map 491  
     *see also* maps  
 valueOf() 741  
 values  
     in a map 485  
     *see also* maps  
 variable 12, 20, 167  
     assigning to 21  
     Boolean variable 46, 741  
     constant 24  
     declared in block 53  
     declaring 20  
     initializing 21  
     local 20, 208, 217  
     meaningful name 229  
     name 20  
     naming conventions 24  
     naming rules 24  
     out of scope 53  
     overwriting 22  
     scope 53  
     static variable 332  
     updating the value 22  
 variance 631  
 Vector 529  
 versatility of methods 218  
 very small program 11  
 views

*see* maps  
 village in Wales 16  
 virtual machine 13  
 visibility 772  
 visual representation 415  
 void 11, 165, 733  
 volatile 733  
 volume of  
     cube 42, 70  
     cylinder 70  
     sphere 70

**W**

Wales 16  
 water bottles 205  
 well-designed program 207  
 while 60, 62, 63, 733  
 wildcard ? 496  
 window 6, 12  
 window close box 642  
 WindowEvent 661  
 WindowListener 664  
 windows 639  
     top-level 639  
     window close box 642  
 word processor 3, 5  
 wrapper class 92  
     Boolean 92, 736  
     Byte 92, 736  
     Character 92, 736  
     Double 92, 736  
     Float 92, 737  
     Integer 92, 737  
     Integer class 245  
     intValue() 94  
     Long 92, 737  
     longValue() 94  
     parseInt() 94  
     Short 92, 737  
     toString() 94  
 wrappers 95  
 write to a text file 298  
 writeObject() 603  
 writers 587  
 writing  
     large programs 4  
     programs 3  
     source code 4, 5  
 writing record 300

**X**

x86-processor series 14

