## Table of Contents

Table of Contents ............................................................................................................. i

Preface to the reader ........................................................................................................ 1
  To students ......................................................................................................................... 1
  To teachers ......................................................................................................................... 4
  To everyone ....................................................................................................................... 6

Acknowledgements ........................................................................................................... 9

Notes ................................................................................................................................... 11

Chapter 0 – Introduction ................................................................................................. 13
  Learning objectives .......................................................................................................... 13
  What is computer science? ............................................................................................... 13
  Hardware ......................................................................................................................... 13
  Software .......................................................................................................................... 15
  Programming languages ................................................................................................. 15
  Summary .......................................................................................................................... 16

Chapter 1 – Classes and objects: identity, state, and behaviour ...................................... 17
  Learning objectives .......................................................................................................... 17
  Introduction ....................................................................................................................... 17
  Definitions ......................................................................................................................... 17
  Classes .............................................................................................................................. 19
  The Student class ............................................................................................................ 21
    Class diagrams .............................................................................................................. 21
    Datatypes ....................................................................................................................... 23
  The Professor class ......................................................................................................... 24
  Tradeoffs of storing the name as one field instead of as several .................................... 25
  Behaviours ....................................................................................................................... 27

A first look at Java .............................................................................................................. 28
  Documentation ................................................................................................................. 29
  Programming Style – documentation ............................................................................ 30
  Class declaration ............................................................................................................. 30
  Programming Style – class names .................................................................................. 30
  Instance variables .......................................................................................................... 31
  Programming Style – instance variables ....................................................................... 32
  Constructor(s) .................................................................................................................. 32
  Programming Style – constructors ............................................................................... 35
  Getters and setters ......................................................................................................... 36
  Programming Style – getters and setters ....................................................................... 37
<table>
<thead>
<tr>
<th>Chapter 2 – Introducing BlueJ</th>
<th>43</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning objectives</td>
<td>43</td>
</tr>
<tr>
<td>Introduction</td>
<td>43</td>
</tr>
<tr>
<td>BlueJ</td>
<td>43</td>
</tr>
<tr>
<td>Creating a new project</td>
<td>44</td>
</tr>
<tr>
<td>Virtual machines and bytecode</td>
<td>49</td>
</tr>
<tr>
<td>Testing the Student class</td>
<td>49</td>
</tr>
<tr>
<td>Inspecting an object</td>
<td>54</td>
</tr>
<tr>
<td>Unit testing - definition</td>
<td>56</td>
</tr>
<tr>
<td>Unit testing with BlueJ</td>
<td>57</td>
</tr>
<tr>
<td>Unit testing – the results</td>
<td>65</td>
</tr>
<tr>
<td>Smoke testing</td>
<td>68</td>
</tr>
<tr>
<td>The Professor class</td>
<td>69</td>
</tr>
<tr>
<td>The College class</td>
<td>69</td>
</tr>
<tr>
<td>BlueJ minutiae</td>
<td>69</td>
</tr>
<tr>
<td>Unit testing tools</td>
<td>70</td>
</tr>
<tr>
<td>Font size</td>
<td>70</td>
</tr>
<tr>
<td>Line numbers</td>
<td>70</td>
</tr>
<tr>
<td>Colours</td>
<td>70</td>
</tr>
<tr>
<td>Default code</td>
<td>72</td>
</tr>
<tr>
<td>Summary</td>
<td>73</td>
</tr>
<tr>
<td>Exercises</td>
<td>74</td>
</tr>
<tr>
<td>Chapter 3 – Making decisions</td>
<td>77</td>
</tr>
<tr>
<td>Learning objectives</td>
<td>77</td>
</tr>
<tr>
<td>The if statement</td>
<td>77</td>
</tr>
<tr>
<td>Boolean algebra</td>
<td>77</td>
</tr>
<tr>
<td>Boolean algebra – an example</td>
<td>79</td>
</tr>
<tr>
<td>A revised toString method</td>
<td>80</td>
</tr>
<tr>
<td>Using the Java documentation</td>
<td>81</td>
</tr>
<tr>
<td>Programming style – if statements</td>
<td>83</td>
</tr>
<tr>
<td>Checkstyle</td>
<td>83</td>
</tr>
<tr>
<td>Simpler tests</td>
<td>84</td>
</tr>
<tr>
<td>More complicated tests</td>
<td>85</td>
</tr>
<tr>
<td>Summary</td>
<td>88</td>
</tr>
<tr>
<td>Exercises</td>
<td>89</td>
</tr>
<tr>
<td>Chapter 4 – Inheritance</td>
<td>93</td>
</tr>
<tr>
<td>Learning objectives</td>
<td>93</td>
</tr>
<tr>
<td>Abstract classes</td>
<td>93</td>
</tr>
<tr>
<td>The Person class</td>
<td>94</td>
</tr>
<tr>
<td>toString in an abstract class</td>
<td>97</td>
</tr>
</tbody>
</table>
Chapter 10 – Collections, part 3 ................................................................. 213
Learning objectives .................................................................................. 213
Introduction ............................................................................................... 213
The Meeting class .................................................................................... 213
  Instance variables ................................................................................... 214
  Behaviours .............................................................................................. 214
  Time arithmetic ....................................................................................... 214
  Exceptions ............................................................................................... 216
Section uses Meeting ................................................................................. 216
  The if statement, revisited ..................................................................... 219
Chapter 9 – Collections, part 2 ................................................................. 187
Learning objectives .................................................................................. 187
Introduction ............................................................................................... 187
A collection of courses ............................................................................. 187
The Course class ....................................................................................... 187
  Course – unit tests .................................................................................. 188
  Formatting the output ............................................................................. 188
printf .......................................................................................................... 190
A collection of courses, continued .......................................................... 190
The Section class ...................................................................................... 190
  Set? List? ................................................................................................. 190
  What is a section? .................................................................................... 192
The Singleton pattern revisited ............................................................... 194
Listing a collection in order ..................................................................... 195
  Comparable ............................................................................................. 196
  Comparators ......................................................................................... 196
  Unit testing the comparator .................................................................. 198
  Alternative versions of the comparator .................................................. 199
Definitions – class and object - revisited .................................................. 200
Producing an alphabetical list of professors ............................................. 200
  The for-each statement, revisited ............................................................ 202
  BlueJ revisited ....................................................................................... 202
Producing a professor list in numeric order ............................................. 203
  Passing a Comparator to a method ........................................................ 203
Registering students in sections ............................................................... 205
Producing an alphabetic class list ............................................................ 206
  The for-each statement, revisited ............................................................ 208
Producing a class list in numeric order .................................................... 208
Students enrol in sections ........................................................................ 209
SortedSet .................................................................................................. 209
Summary ................................................................................................... 209
Exercises ................................................................................................... 210
Collection of departments ...................................................................... 183
Collection of students ............................................................................. 183
Summary ................................................................................................... 184
Exercises ................................................................................................... 185
Creating the collection of Meeting objects ................................................................. 220
Section uses Meeting – displaying the times .............................................................. 221
  Arrays ........................................................................................................................... 222
  StringBuffer .............................................................................................................. 225
  Adjusting to different colleges .................................................................................... 226
  Another for loop ......................................................................................................... 227
Processing all the meetings .......................................................................................... 228
Manipulating StringBuffers ......................................................................................... 228
  Placing data in StringBuffers ................................................................................... 228
  Converting numbers to day of the week ..................................................................... 230
Processing second and subsequent meetings ............................................................... 231
A lot of code ................................................................................................................... 232
Programming Style ..................................................................................................... 234
Course contains Section ............................................................................................... 235
Mark, and Student contains Mark .................................................................................. 236
Professor contains Section ........................................................................................... 237
Summary ......................................................................................................................... 238
Exercises ......................................................................................................................... 239

Chapter 11 – Collections, part 4 .................................................................................... 241
Learning objectives ......................................................................................................... 241
Introduction ..................................................................................................................... 241
Grade Average ................................................................................................................ 241
  When to calculate an average ...................................................................................... 241
  Weighted average ....................................................................................................... 242
  The transcript method – original version ................................................................. 242
  Knowing when a semester changes ......................................................................... 243
  Transcript – a skeleton for changes ......................................................................... 243
  When the semester changes ....................................................................................... 244
    Integer division .......................................................................................................... 245
  When the semester remains the same ....................................................................... 246
  The declarations .......................................................................................................... 246
  The overall average ..................................................................................................... 246
  The transcript String – problems and deficiencies ................................................ 247
  The String class and its methods .............................................................................. 247
    Details of split .......................................................................................................... 248
    Details of indexOf and lastIndexOf ........................................................................ 250
  Reversing a String ....................................................................................................... 250
  Palindromes ............................................................................................................... 251
  The transcript String – reordering ............................................................................ 252
  The transcript String – problems and deficiencies, continued ............................... 254
Summary ......................................................................................................................... 256
Exercises ......................................................................................................................... 257

Chapter 12 – Exceptions revisited ................................................................................ 259
Learning objectives ....................................................................................................... 259
Definition ....................................................................................................................... 259
Examples ......................................................................................................................... 260
### Chapter 16 – Creating a GUI: data entry screens, part 1

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning objectives</td>
<td>331</td>
</tr>
<tr>
<td>Warning to the reader</td>
<td>331</td>
</tr>
<tr>
<td>Introduction</td>
<td>331</td>
</tr>
<tr>
<td>GUI</td>
<td>331</td>
</tr>
<tr>
<td>Some HCI considerations</td>
<td>333</td>
</tr>
<tr>
<td>Layout managers</td>
<td>334</td>
</tr>
<tr>
<td>Installing the RelativeLayout layout manager</td>
<td>336</td>
</tr>
<tr>
<td>Creating a data entry screen</td>
<td>336</td>
</tr>
<tr>
<td>How the data entry screen should appear</td>
<td>336</td>
</tr>
<tr>
<td>Import statements</td>
<td>337</td>
</tr>
<tr>
<td>Creating the JFrame</td>
<td>337</td>
</tr>
<tr>
<td>Making the JFrame visible</td>
<td>340</td>
</tr>
<tr>
<td>Explaining the code</td>
<td>340</td>
</tr>
<tr>
<td>The JFrame</td>
<td>341</td>
</tr>
<tr>
<td>The layout manager</td>
<td>341</td>
</tr>
<tr>
<td>BindingFactory</td>
<td>341</td>
</tr>
<tr>
<td>The labels</td>
<td>342</td>
</tr>
<tr>
<td>The label Bindings</td>
<td>343</td>
</tr>
<tr>
<td>The other labels</td>
<td>344</td>
</tr>
<tr>
<td>JTextFields</td>
<td>345</td>
</tr>
<tr>
<td>Other text fields</td>
<td>346</td>
</tr>
<tr>
<td>Done!</td>
<td>346</td>
</tr>
<tr>
<td>Summarizing building a data entry screen</td>
<td>346</td>
</tr>
<tr>
<td>Displaying the frame</td>
<td>346</td>
</tr>
<tr>
<td>Threads – definition</td>
<td>347</td>
</tr>
<tr>
<td>Threads and Swing</td>
<td>347</td>
</tr>
<tr>
<td>Runnable</td>
<td>347</td>
</tr>
<tr>
<td>Create the frame</td>
<td>348</td>
</tr>
<tr>
<td>Position and size the frame</td>
<td>348</td>
</tr>
<tr>
<td>box.pack();</td>
<td>349</td>
</tr>
</tbody>
</table>

---

**Creating the**

How the data entry screen should appear ................................................................. 336
Import statements .................................................................................................................. 337
Creating the JFrame ............................................................................................................... 337
Making the JFrame visible .................................................................................................... 340
Explaining the code ............................................................................................................... 340
The JFrame ............................................................................................................................ 341
The layout manager ................................................................................................................ 341
BindingFactory ....................................................................................................................... 341
The labels ............................................................................................................................... 342
The label Bindings ................................................................................................................... 343
The other labels ..................................................................................................................... 344
JTextFields ............................................................................................................................ 345
Other text fields .................................................................................................................... 346
Done! ........................................................................................................................................ 346
Summarizing building a data entry screen ......................................................................... 346
Displaying the frame .............................................................................................................. 346
Threads – definition ............................................................................................................... 347
Threads and Swing ................................................................................................................. 347
Runnable ................................................................................................................................. 347
Create the frame ..................................................................................................................... 348
Position and size the frame .................................................................................................... 348
box.pack(); ........................................................................................................................... 349
Create the middle panel, continued

Building the Section data entry screen

Learning objectives
Exercises
Summary
Data entry for other classes
Improvements
Verification behind the Okay button
Editing data
JButtons
Cancel button
Create a JButton
Create a JButton’s bindings
Add a JButton and its bindings to the frame
Create a mnemonic
The default button
Button action
The OK JButton
ActionEvents and ActionListeners
Creating an ActionListener
Linking an ActionListener to a JButton
What does an ActionListener do?
Behind the Cancel Button
Editing data
Verifying year and month text fields
Verifying the day text field
Verification behind the Okay button
Improvements
Remove intrusive dialogs
Creating the message
Using the message
Disable the OK button until needed
Data entry for other classes
Testing a GUI
Summary
Exercises

Chapter 17 – Creating a GUI: data entry screens, part 2
Learning objectives
Introduction
Building the Section data entry screen
Create the frame
Create a column of panels
Create the three panels
Populate the top panel
Place the boxes in the panel
Decorate the panel
Create the middle panel
Labels
The meetings
Arrays
Create the middle panel, continued
Radio buttons
<table>
<thead>
<tr>
<th>Chapter 22 – Mathematical methods</th>
<th>459</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning objectives</td>
<td>459</td>
</tr>
<tr>
<td>Introduction</td>
<td>459</td>
</tr>
<tr>
<td>Greatest common divisor (gcd)</td>
<td>460</td>
</tr>
<tr>
<td>A recursive greatest common divisor method</td>
<td>460</td>
</tr>
<tr>
<td>Sample greatest common divisor calculations</td>
<td>461</td>
</tr>
<tr>
<td>Fibonacci numbers</td>
<td>462</td>
</tr>
<tr>
<td>Iterative Fibonacci method</td>
<td>463</td>
</tr>
<tr>
<td>Recursive Fibonacci method</td>
<td>464</td>
</tr>
<tr>
<td>The factorial function</td>
<td>465</td>
</tr>
<tr>
<td>The choose function</td>
<td>466</td>
</tr>
<tr>
<td>A Formula!</td>
<td>466</td>
</tr>
<tr>
<td>Recursive choose method</td>
<td>466</td>
</tr>
<tr>
<td>The Ackermann function</td>
<td>467</td>
</tr>
<tr>
<td>Definition of the Ackermann function</td>
<td>467</td>
</tr>
<tr>
<td>Recursive implementation of the Ackermann function</td>
<td>467</td>
</tr>
<tr>
<td>Dynamic programming</td>
<td>468</td>
</tr>
<tr>
<td>Storing the data – Fibonacci</td>
<td>469</td>
</tr>
<tr>
<td>Saving the data – Fibonacci</td>
<td>470</td>
</tr>
<tr>
<td>Restoring the data – Fibonacci</td>
<td>471</td>
</tr>
<tr>
<td>Saving the data – Ackermann function</td>
<td>472</td>
</tr>
<tr>
<td>Restoring the data – Ackermann function</td>
<td>473</td>
</tr>
<tr>
<td>Summary</td>
<td>473</td>
</tr>
<tr>
<td>Exercises</td>
<td>474</td>
</tr>
<tr>
<td>Chapter 23 – Loose ends</td>
<td>475</td>
</tr>
<tr>
<td>Learning objectives</td>
<td>475</td>
</tr>
<tr>
<td>Introduction</td>
<td>475</td>
</tr>
<tr>
<td>public static void main(String[] args)</td>
<td>475</td>
</tr>
<tr>
<td>Summary</td>
<td>476</td>
</tr>
<tr>
<td>Exercises</td>
<td>477</td>
</tr>
<tr>
<td>In Conclusion</td>
<td>479</td>
</tr>
<tr>
<td>References</td>
<td>481</td>
</tr>
<tr>
<td>Printed Reference materials</td>
<td>481</td>
</tr>
<tr>
<td>Online references</td>
<td>482</td>
</tr>
<tr>
<td>Lab Assignments</td>
<td>483</td>
</tr>
<tr>
<td>Index</td>
<td>485</td>
</tr>
</tbody>
</table>
Preface to the reader

Oh no, not another Java textbook!

Perhaps that’s a valid comment, but wait until you’ve had a chance to look at this one. There is a lot of discussion amongst teachers about when to introduce objects in Java. I feel that objects should be introduced early, as early as possible, perhaps on the first day of an introductory course. I also believe that students should use a large application as an example, so they can see the advantages of objects.

Reflecting these ideas, this is not just another Java book.

To students

What is computer science? How does this textbook help you study computer science?

Much, but not all, of computer science involves using a computer to solve a problem. The instructions to solve the problem are written as a program or programs, instruction which the computer performs.

In this book we gloss over some of the details of how the program you write is translated into something which the computer can carry out, and we focus on how to write good programs. By “good”, we mean clear, easy-to-read, well-documented, efficient, and tested to ensure they meet the requirements.

Clear, easy-to-read, and well-documented go together. Can someone else read your program and understand what it does? If not, you have failed in creating something which can be used. This is because programs are continually changing. The problem they are solving changes, or a better solution becomes available. If you or someone else cannot understand what you have written, then how can you improve or change it? How can you fix it if it contains an error? We will use a tool called Checkstyle to help us meet these goals.

Efficient means that the program carries out its task quickly, or with a small amount of computer resources. Often these two characteristics are in conflict; a program may work faster when you give it more memory. But a detailed discussion of efficiency is beyond the scope of this book.

Tested means that you can trust the results the program produces. We emphasize unit testing using a tool named JUnit, which allows us to test each portion of the program separately, and together. It allows us to save the tests and reuse them whenever we make a change to the programs.
Tested to ensure requirements means that the program does what it is supposed to do, and does it correctly. You can test a program and it may produce what you thought it would, but does it produce what it should? We don’t have a tool to test that it meets the requirements, but we will be very careful in ensuring we do meet them.

When we write programs, we use the Java language. This is a modern language which I find very enjoyable and easy to use. The language itself is very simple, and gains much of its power from libraries which people have created to carry out specialized tasks. There are many other languages available so you will need to learn many languages during your career in computing. Java is a good language with which to start.

A significant part of learning Java is becoming familiar with the documentation describing these libraries; it is available online. The number of libraries is truly amazing. This is a measure of the popularity of Java and the various types of programs it is used to create.

We will use a tool called BlueJ to help us integrate all these features – the Java language, JUnit testing, the Java libraries, and Checkstyle. Since BlueJ is written in Java, it is a constant reminder of what you can produce using Java. Most importantly, since Java will run on many different operating systems, you can use computers running Linux, Mac OS, or Windows.

For much of this book, we use one example. The example is a North American college, what some call post-secondary education and others call tertiary education. At this college, the professors teach students who are enrolled in sections of courses. A course is a collection of related topics. Students sign up to take many different courses. Since there are limits on how many students a professor can teach at one time (the college has a goal of having no large classes), when there are many students wanting to take a course there may be a number of different sections of the course being offered, perhaps all taught by the same professor, perhaps taught by different professors.

Some courses are best taught using lectures, where the professor teaches and the students participate by listening and asking questions. Some are best taught using labs, where the students
are doing predefined activities, and the professors provide assistance. Some are best taught using seminars, where the students study material on their own and present their discoveries to the rest of the class. Some courses use a mix of two or all of these.

Whether a course uses lectures, labs, or seminars, there are meeting times associated with each.

The academic year is divided into three portions, called semesters. The semesters run from September through December, January through April, and May through August.

Some courses may be offered in two or three semesters. A section of a course offered in one semester is different from a section of a course offered in a different semester.

In order to complete a program of study, perhaps resulting in a diploma or a degree, students may take a course more than once, with marks being awarded for each attempt.

To model this college, we need a way to save large amounts of data.

Some have suggested that an introductory textbook like this should use many small examples, rather than one large example. But others have suggested that the way to see the benefits of objects is to see them in a large project. I agree with the latter perspective.

The exercises at the end of each chapter include other examples, some small and some large. The lab assignments included in the appendix also include a large project. I hope you find it interesting. Although it is based on a sports league, it should be understandable even if you are not interested in sports. Another time the course was taught, the labs involved the records you keep when watching birds.

I hope you enjoy the approach I have used here.

Note that I do assume you have used computers before and are familiar with some of the terminology. I do not assume any programming experience.

Some of you may be a little concerned about learning to program computers. You may be worried that you will break something. You won't break anything, but you will make lots of mistakes on the way to becoming good, even excellent, programmers. You may find this quotation interesting.

_Nobody tells this to people who are beginners, I wish someone told me._

_All of us who do creative work, we get into it because we have good taste. But there is this gap. For the first couple years you make stuff, it's just not that good. It's trying to be good, it has potential, but it's not._

_But your taste, the thing that got you into the game, is still killer. And your taste is why your work disappoints you._
A lot of people never get past this phase, they quit.

Most people I know who do interesting, creative work went through years of this. We know our work doesn’t have this special thing that we want it to have. We all go through this. And if you are just starting out or you are still in this phase, you gotta know it’s normal and the most important thing you can do is do a lot of work.

Put yourself on a deadline so that every week you will finish one story.

It is only by going through a volume of work that you will close that gap, and your work will be as good as your ambitions.

And I took longer to figure out how to do this than anyone I’ve ever met. It’s gonna take awhile. It’s normal to take awhile. You’ve just gotta fight your way through.

Ira Glass, quoted at http://galadarling.com/article/i-want-to-be-a-sex-journalist

To teachers

Like many textbooks, this one arose out of a perceived need. I wanted to teach an introductory programming course using Java, and I wanted to do it objects-first. I could not find a textbook that met my needs, so I decided to create my own. The competition was between editions, and other competitors were not yet available. In the time it has taken to complete this book, other competitors have appeared. To see this textbook published, I have used the Creative Commons approach.

When I began writing, Java 5 was the most up-to-date version of Java available at the time. Since then, Java 6 (and then Java 7 and then Java 8) became available (and Java 9 is becoming/became available in the Fall of 2015), but it does not appear to affect anything here. I have not used anything specifically beyond Java 6.

I originally used BlueJ version 2.1.2 as it was the most recent stable version at the time. Since then other versions have become available. In checking all the code in this textbook, I have used version 3.1.4 along with the Checkstyle extension. I have also used the RelativeLayout manager from Google for designing GUIs.

BlueJ 3.1.4 includes unit testing tools. I believe unit testing is very important, even crucial, so I include it throughout. There may be more lines of unit tests than there are lines in the code being tested.

My reviewers have given me much interesting feedback. One thing which caused many of them some difficulty is the use of one large project to form the examples in this book. Of course, using one example in the lab assignments also caused similar concern. In the preface to students I have explained the project for the benefit of those who are not familiar with the way a North American college or university works.
Also in the preface to students, I addressed the question of one large example instead of several small ones. Basically, the problem is that it is very easy to write small problems using objects, but the objects are not crucial to success. That is, you could write the small programs without using objects if you wished. But I believe that using objects is an important way of designing your software, so I want to force people to use objects. Thus I need a language which forces everything to be an object (Java almost meets that requirement), or I need a project which is so large that you must use objects. I have taken the second approach, in the example in the textbook, and in the lab assignments.

Notice that after a while I find I have exhausted the North American college example of ideas. At that point we look at some other examples, starting with mathematics.

I have included my most-recent set of lab assignments in an appendix. I doubt you'll want to use them (the project they implement is too esoteric for many) but they serve as an idea of what is possible.

My reviewers have also asked why is <that topic> included. Depending on the reviewer, <that topic> could be persistence, GUIs, applets, patterns, and/or mathematics. I have included them because I feel they are important topics, which can be appreciated by students learning to program for the first time.

However, you may wish to omit any or all of those topics, depending on the length of your course, the students, or your preferences. All of those chapters are not dependent on the rest of the book, except that there is some persistence in the mathematical chapters but you could omit that too, should you prefer.

Other brief sections which present interesting but non-essential material are shown with a grey background.

Like this.

I have used this textbook as the basis of a two-semester course, to be followed by a data structures course. That is, instead of teaching the ACM courses CS1 and CS2, we cover the content in three semesters. I have used this textbook with a small group of students and with larger groups. Other colleagues have used the textbook with large groups of students. They had some difficulty, not because of the class size, but because they were not objects-first people.

We find that the material in this book is adequate for about 80% of a two-semester course. That is, it covers all the material in CS1 plus a little of CS2. For the rest of the course we use material specific to Okanagan College, related to the environment and follow-up course.

Thus, this textbook is more than adequate for a one-semester course and there are chapters, mentioned earlier, which you could omit without disturbing the integrity of the course. But should you be using this textbook for a two-semester course, be prepared to supplement it
towards the end of the second semester. The teamwork features of BlueJ would be an ideal supplement, particularly version control.

To everyone

Thank you for purchasing this book. I hope it meets your needs. Remember that Dr Samuel Johnson said “The two most engaging powers of an author are to make new things familiar, and familiar things new.” This is a quotation which I found at http://www.chiasmus.com/welcometochiasmus.shtml. I hope I have those powers.

Should you have any comments, positive or negative, and especially if you find any errors or omissions or unclear sections, please contact me.

Rick Gee
Now retired, formerly with Okanagan College
Kelowna, BC
Canada

rdgee at shaw dot ca

p.s. Yes, I am Canadian, so the spelling in the book is Canadian, and the examples are, in many ways, Canadian.

p.p.s. I have a quirky sense of humour so you may find jokes or cartoons at various points in this book. The cartoons are from xkcd.org and are “licensed under a Creative Commons Attribution-NonCommercial 2.5 License. This means you're free to copy and share these comics (but not to sell them).”
http://xkcd.com/844/

Fortunately this cartoon does not describe how we will develop code.
Acknowledgements

I’d like to thank my colleagues Rob McArthur (formerly of Okanagan College) and Joseph Fall (formerly of Capilano College, now Capilano University) for their help in the writing of the examples in this textbook.

I would also like to thank the first students, Andrew Faraday, Jonathan Gaudet, Nicholas Goertz, Maria Nemes, and Brandon Potter for their patience and assistance with version 1.

I’d like to thank Rob McArthur and his students for their patience and assistance with version 2.

I’d like to thank Peter Kossowski and his students, and my students in 2009-10, 2010-11, 2011-12, and 2014-15 for their patience and assistance with previous versions of this book.

I’d like to thank Seth A Maislin and his website, http://taxonomist.tripod.com/indexing/wordproblems.html, for his assistance in creating the index for this document. Microsoft should take note of some on the points mentioned there. Please.

Thanks to Microsoft for the problems with the index. Of particular concern are entries for object/Object, short, long, while, for, and this.

In the beginning I thought of publishing this as a real book, using paper and ink. I would like to thank the reviewers who provided me with interesting and challenging feedback at that time. I’d like to thank the authors who published similar books at that time for encouraging me to take the electronic route.

Rick Gee
Kelowna, BC

Originally completed summer 2008
Most-recently revised summer 2015. Final revision!
Notes

Statements in the Java language are shown using this font.

Java snippets (collections containing more than one statement), including variable and class names referred to in the body of the text, are also shown using this font. Be aware that it may be a little difficult to see spaces in this font as they are quite narrow.

Thus, Address is the name of a class we create but address is the place a person lives.

Similarly Person is a class but a person is a member of the human species.

Student is a class derived from Person but a student is a person.

Professor is a class derived from Person but a professor is a person.

Section is a class but a section is a specific offering of a course.

The occasional paragraph is shown with a light-grey background. This is additional material, which expands on the material nearby. While interesting, the material is not crucial.

All the examples used assume the Java Platform, Standard Edition, and have been run through BlueJ 3.1.4.

The first time a word is defined, it is usually shown in italics.
Chapter 0 – Introduction

Learning objectives

By the end of this chapter, you will be able to:

- Describe hardware
- Describe software
- Name a few programming languages
- Say why the author believes Java is the appropriate language to use to introduce programming

What is computer science?

*Computer science* is not the study of computers themselves; that would be computer engineering.

Computer science is the study of the use of computers in various parts of life, both commercial and non-commercial. Perhaps we should call it “computing science”, the study of computing. Some universities and colleges have done that. European institutions often use the word “informatics”.

To study computer science, we need to understand how to design solutions to problems, solutions which often use computers. To understand computers, we need to understand hardware, software, and programming. Let’s have a quick look at those terms, about which I assume you already know something.

Hardware

*Hardware* is the parts of a computer which you can touch. Some define hardware as the parts of a computer you can hit when something goes wrong.

Hardware is capable of responding to instructions on how to do a task. These instructions come via programming. Depending on the programming, a computer may perform various tasks.

Thus, an MP3 player is not a computer, since it performs only one task, but a computer can function as an MP3 player. Older cell phones are not computers, but a computer can function as a telephone (with the addition of a microphone and speakers.) An address book (a personal digital assistant, or PDA) is not a computer, but a computer can function as an address book.
A smartphone is a computer.

A computer transforms data into information. Data is the things we know or are given. (The word data comes from Latin, from a verb meaning “to give”.) Data includes the students who have enrolled in a course and the marks they have earned. Information is “the result of processing, manipulating and organizing data in a way that adds to the knowledge of the person receiving it.” (http://www.orafaq.com/glossary/faqglosi.htm)

For example, the raw data about the students in a class can be transformed into a class list, or information. The marks students earn can be transformed into semester averages, information which will in turn become data for determining which scholarships students win.

The November 2010 issue of Communications of the ACM contains an article entitled Turning Data into Knowledge. It reports that the Large Synoptic Survey Telescope will produce “an SDSS-equivalent dataset every night for the next ten years.” The Sloan Digital Sky Survey (SDSS) contains 45 terabytes of data. That is a lot of data to convert into information.

Turning data into information depends on what programming can be done for the computer.

Hardware – the parts of a computer which you can touch. This includes:

- Central Processing Unit (CPU) – the brains of the computer. This is the part of the computer which carries out the instructions given by a program. Modern computers often have many CPUs (called cores), often on one chip.

  Heinrich Rudolf Hertz was a German physicist in the late 1800’s. His name has been adopted as the unit to measure frequency. 1 hertz = 1 oscillation of a wave per second.

  The speed of the CPU used to be measured in megahertz (millions of cycles per second, abbreviated MHz) but is now measured in gigahertz (billions, or thousands of millions, of cycles per second, abbreviated GHz).

  In general, the faster the CPU, the better, although when you have multiple CPUs (or cores) you can use a slower speed for each since they share the tasks.

- Random Access Memory (RAM) – used to store data while a program is running.

  In general, the more RAM your computer contains the better. In the old days, RAM was measured in kilobytes (KB). Then memory became cheaper and computers contained megabytes (MB) or gigabytes (GB) of memory.

  The reason you want more memory is twofold. First, you can run larger, and hence more powerful, programs. Second, you can have more programs in memory simultaneously, with the CPU giving each a portion of its time.

- Read-only Memory (ROM) – a portion of memory which uses a battery to retain its contents when the power to the computer is turned off.

  ROM is used for things like configuration information, and a boot program, which is
executed when the power is first turned on.

ROM is more expensive than RAM, so you will have less ROM than RAM, often much less.

- Monitor – the screen on which output from a program is displayed. This output may be text or graphics.

You'll find that like many things in life, bigger is better. If you can't have a bigger monitor perhaps you can have more than one.

- Keyboard – used for input. There are many different layouts for keyboards (Where is the backspace key?) depending on the manufacturer of the keyboard and many different special keys. The keyboard for a Macintosh computer has an “Apple” key, for example.

- Mouse or trackball – used for input, clicking buttons or selecting items from a menu.

We will use the mouse quite extensively to control BlueJ.

Software

Every computer has some programming available for it. This includes an operating system (a version of Windows, UNIX, Linux, Mac OS, Solaris, etc.). The operating system allows other programs to run. When you get an operating system, it often comes with some other programs – word processing, calendar, browser, various utility programs, and a few games.

You will be using some parts of the operating system in your labs, to locate files you have created, to make backup copies of them, and to run programs.

You may also want to create new programs, perhaps one to track the inventory in your warehouse, or to track the hours worked by your employees and then pay them. Or perhaps you will want to create programs to help you write more programs. Perhaps you want to create a game. For those tasks, you need a programming language and the ability to use it. Or you may need the money to hire someone who does.

Programming languages

Programming is done using a variety of languages, in a variety of styles. You may have heard of BASIC, C, Pascal, etc. These are all programming languages which have been used in the past and continue to be used, to some degree. Today there are many languages in common use – including Java, C++, C#, and Visual Basic.

Why are there so many different programming languages? Because they were designed to serve special purposes.

BASIC (an acronym formed from the phrase Beginner’s All-purpose Symbolic Instruction Code) was designed in 1963 as a tool for teaching people to program.
C was developed in 1972 for use with the UNIX operating system. (And yes, its predecessor language was called B.)

Pascal was developed in 1970 as a language for teaching people how to program.

When you learn your first programming language, it’s good to start with an easier one, so I have chosen Java. Why Java?

It’s a language commonly used today, and is being used more and more as time goes by. It is a powerful language. That is, it allows you to do many different tasks and thus has applicability in a wide variety of domains (different areas of programming – games, database, and student registration are examples) on a variety of platforms. These platforms include mainframes, personal computers, and mobile devices.

Java is a language that uses a popular programming paradigm, object-oriented programming. What is a paradigm?

It’s a way of thinking about what you are doing. It’s the set of practices that define a discipline during a particular period of time. (http://en.wikipedia.org/wiki/Paradigm)

Wikipedia is an online encyclopaedia, in which the entries in the encyclopaedia are written by people like you and me. That means there is a certain risk in using Wikipedia as a source. How do you know that the author is reliable?

In general, I have found that the articles about computer science are reliable and I will quote many of them in the material that follows.

In programming, a paradigm is a way of creating programs, using popular techniques which allow you to create “good” programs. Over time, paradigms change as people develop different, hopefully better, ways of doing things. At one time, the paradigm was “structured programming”. Now, the dominant paradigm is object-oriented. Perhaps we should define what we mean by object-oriented.

That is one of the topics in chapter 1.

Summary

In computing, hardware, software, and programming are combined to create instructions to control a computer. The instructions are expressed in a programming language, using a particular programming paradigm. Java supports the object-oriented paradigm.
Chapter 1 – Classes and objects: identity, state, and behaviour

Learning objectives

By the end of this chapter, you will be able to:

- Define class
- Define object
- Draw a class diagram
- Implement simple classes in Java
- Define method, constructor, getter, setter
- Create constructors, getters, and setters
- Describe some primitive datatypes and some datatypes represented by classes
- Document Java code
- Describe the purpose of, and implement, a toString method

Introduction

This chapter contains a lot of material, so you’ll want to take it slowly. First we discuss what objects and classes are, and how we model them.

Then we discuss how to write a class.

Then we look at BlueJ, the tool we will be use to help us in our modelling and our coding.

Then we look at JUnit, the tool we will use to test the programming that we create.

Finally, we use what we have learned to create a second class.

Definitions

When you write a program, you provide the instructions which tell the computer how to process pieces of data. Data in the real world comes in many different types. Depending on the language you are using, you will have different datatypes provided to you, but not always the datatypes necessary to describe the real-world data. The programming language you use will allow you to create new datatypes. In object-oriented programming, we create new datatypes by creating new classes and using the classes to create objects.
In computing, the word *object* has a very specific meaning – an object is an instance of a class. Similarly, the word *class* has a very specific meaning – a blueprint for creating objects.

But those definitions are circular; they don’t really tell us anything. We need to explore the concepts of object and class more fully before we can understand the definitions.

So what is an object? Since we are talking about computer programming, an object is a representation, implemented using a computer language and stored in a computer, of something in which we are interested. This “something” may be real, like a person or a piece of furniture, or imaginary, like a mathematical function.

You, a student, are an object in a system involving college registration. There are many students at a college; all are objects. All students have things in common, like name, address, and student number.

A system involving college registration will need other objects. Each course the college offers is an object. You receive a final mark in each course you take, so there can be *associations* between one object and another. When two objects are related in some way, we say there is an association between the objects.

A student registers in a section, and a section is an object. The registration creates an association between a student and a section.

One object may participate in many associations. For example, you are registered in sections, you earn marks in courses, and you may incur library fines.

But think about this for a moment. There are many student objects enrolled at Okanagan College. All these objects have some things in common; they each have a name, a student number, and an address. They each have associations with sections of a course. Each section has an association with a course.

Whenever you find things having common features, you create a blueprint for creating objects, rather than just creating objects.

Thus we have a definition of a *class* – a blueprint for creating objects – which makes more sense now than when we first proposed it.

This idea of a blueprint also works in other areas as well.

Think of a new building containing apartments (flats). Many of the apartments have the same layout. They were all made from the same blueprint.

Many of the floors of the building have the same layout. They are all made from the same blueprint.
And look at some of the new subdivisions being built around our cities. Many of the houses in those subdivisions look alike; they are made from the same blueprint.

In the building in which I have recently purchased a home, some of the units are identical and some are mirror-images of each other. But they are built from the same blueprint.

And we have a definition of an object – an instance of a class. To flesh out the definition of object, in computing an object has an identity (its name), it has state (the values which it knows), and behaviours (the things it can do, like tell you about its state).

In the construction examples above, several apartments may have the same layout, but they are different apartments.

They too have identities (apartment number 307, for example), state (the names of the occupants), and behaviours (what is the rent and when was it last paid?)

Houses have identities (the street address), state (including the owner and the colour of the front door), and behaviours (who owns the house? Is it owner-occupied?)

So what is object-oriented (OO) programming?

It is a programming style (a paradigm) which uses classes and objects. Its processing is based on messages being sent to and between objects, and to and between classes.

An OO language is one which supports OO programming (OOP). It allows you to create classes and objects. A pure object-oriented language lets you use only the OO style.

C++ is not a pure OO language. This is because of its roots in the C language. Smalltalk is a pure OO language. Java is not a pure OO language but it comes close. As we will see, there are a few things in Java that are not objects.

Classes

We have identified “student” as a class whose objects are in a classroom room. What other classes can we identify whose objects are in a classroom?

Well, depending on the application, we could consider the room itself. It has a room number and a capacity. A Room class could represent a room. Note that I have shown the name Room in a different typeface, one which I have reserved for code in the Java language. This indicates that we may be implementing the Room class in Java.

The word code means the instructions we write in Java. These usually are statements. A statement in English is a complete thought. A statement in Java is a complete instruction to the computer. To implement an object, we write code.
Looking for other objects in the classroom, we could consider the desks, the chairs, and the tables themselves. They may have serial numbers attached. They have a description. A Furniture class could represent them.

All your textbooks (except this one) have an ISBN, a title, and an author. A Textbook class could represent a textbook.

Your professor represents a class. Your professor has a name and an employee number, the same way you have a name and a student number. A Professor class could represent a professor.

So we are surrounded by objects and can use our language to determine the classes from which they are instantiated (created as instances. An instance is an object created using a class as a blueprint.) Once we have identified the classes, we will go on and identify the associations between them.

So where do we start when we want to model these real-world classes?

Why do we want to model real-world classes in a computer?

Perhaps we would like to be able to determine the effects on our institution if we reduced the class size. How many more sections of courses would we need to offer? Do we have enough classrooms?

A model would help answer those questions.

Those questions are relatively simple, and we could do the experiment in the real world. Some experiments are not so simple.

To add an environmental note, we have been adding large amounts of greenhouse gases (carbon dioxide, methane, etc.) to the atmosphere during the past 250 years. Most people believe this is having an effect on climate and weather.

The weather is becoming more changeable. Most glaciers are melting, whether they are in the mountains, the Arctic, or the Antarctic. Plants are blooming earlier than previously seen. The Inuit of the Canadian Arctic are seeing birds (the American Robin, for example) which they have never seen before; they had to create a new word for the American Robin!

We appear to be making some serious changes to the world, without knowing the consequences of these changes. Would it not have been better to make a model and do the experiments there?

When I first began teaching, there was some discussion about “nuclear winter.” This was the idea that if there was a nuclear war, the resulting explosions would throw so much dust into the atmosphere that the amount of sunlight reaching the surface of the Earth would be reduced significantly. This would lower the temperature, and reduce much of the plant growth.
Again, this is not something we would like to do in the real-world to see what would happen. Well, we may actually try something like this if a large volcano erupts. When Mount Pinatubo erupted in June 1991, we did see a cooling effect but only for a short time.

So how do we make models using the Java language?

**The Student class**

Let’s begin with the people in the room. We identified two types of people in the room (not male and female!), student and professor. Notice the word “type”; this is an English word that makes experiences programmers think “class”. We will model these two types of people with Student and Professor classes.

What standard am I using for the name of a class?

Right, the class name is a singular noun and its first letter is capitalized. When the name of a class includes several words, each is capitalized. Thus, in a different example, we might have a DepartmentStore class.

**Class diagrams**

We need to be able to describe these two classes, using some standard language. In this course we will use two languages. One, called the Unified Modelling Language (the UML), will be used for drawing diagrams; after all, a picture is supposed to be worth 1000 words. When we use words, we will use the Java programming language. (Note that Java is also an island in Indonesia and a synonym for coffee. There are many coffee jokes associated with Java, including JavaBeans and the coffee cup logo. There are no Indonesian island jokes of which I am aware, however.)

In the UML, a class is represented by a rectangle, displaying one, two, or three subsections. The first subsection always appears, and it contains the name of the class.

The second subsection of the rectangle contains the *attributes* (the names of the data items, the pieces of data which an object contains), whose values we mentioned earlier. This subsection is optional in the UML but is obviously necessary when you want to translate the diagrams into Java.

The third subsection of the rectangle contains the names of the *methods* the class supports. These are the pieces of Java code that implement the behaviours we mentioned earlier. This subsection is optional in the UML but is obviously necessary when you translate the diagrams into Java.

Thus, let’s consider the UML representation of a Student class. It can be represented simply as follows.
This represents a class. All that we know about the class is its name. We don’t know its attributes. We don’t know its methods. But we usually need to know more about the class.

Consider the attributes of a Student object, an object instantiated from the Student class. Remember that a Student class is simply a blueprint, describing what a Student object knows and can do.

We have already mentioned that a student has a name and a student number. Thus a Student object will have attributes named studentNumber and studentName. Since names are not unique, everyone needs a number, which will be unique; no other student will have that number. The Registrar’s Office at my college is responsible for ensuring that the student numbers are unique. Part of that responsibility is to ensure that once a student has graduated, her student number is never reused.

In a previous career, I worked for a company which reused employee numbers. That caused many problems when you looked at historical data for employees who were no longer with the company. Their numbers may have been reused two or more times so information on two or more employees would be combined.

Java coding standards specify that attribute names do not begin with a capital letter. If an attribute name contains several words, then the second and subsequent words are capitalized. As English is my native language, the attribute names consist of English words. There is nothing to stop you from using words in your native language to create attribute names.

These are two different attributes of every Student object, so they can be shown in the Student class diagram. Every Student object has a name and a number, as each of you do.

Thus, the UML notation of the Student class, showing the attributes is

Unfortunately BlueJ does not allow this form of the class symbol. These class diagrams are created by a free UML editor called Violet, available at http://horstmann.com/violet. There are other tools for drawing UML diagrams, both free and for sale.
Is it a problem that BlueJ only offers simplified UML diagrams? It would be a problem were we developing a commercial product, but it doesn’t seem to be a problem for our project.

**Datatypes**

What type of data do we use to store the studentNumber and studentName?

Everything stored in a computer is ultimately stored as patterns of 0s and 1s, called *bits*. The word *bit* is an abbreviation for “binary digit.” Binary means you only have two choices (usually represented by 0 and 1) and *digit* refers to the individual parts of a number. When you collect eight bits and treat them together, as one *byte*, then a byte can represent 256 different values.

If you have one bit, then you can represent only the values 0 and 1. If you have two bits, you can represent 00, 01, 10, and 11, four different values. If you have three bits, you can represent 000, 001, 010, 011, 100, 101, 110, and 111, eight different values. Similarly, four bits can represent 16 different values, five bits can represent 32 different bits, six bits can represent 64 different values, seven bits can represent 128 different values, and eight bits can represent 256 different values. In general, when you have n bits, you can represent $2^n$ different values.

Having 256 possible values allows us to represent all the letters of the English alphabet, the digits from 0 through 9, some punctuation symbols, plus a few other letters. However, this does not represent letters in many other alphabets. To do that, we need to group the bits into a larger unit.

The encoding of these 256 values came from the American Standard Code for Information Interchange (ASCII). You might suspect that the encoding came from the United States and you would be correct. Hence the emphasis on English.

The Unicode standard, described at [http://www.unicode.org](http://www.unicode.org), uses 16 bits to represent individual characters. That is apparently enough to represent all Earthly languages. Klingon, an invented language used in the Star Trek television series and movies, is not officially included.

Fortunately, Java has a datatype named *String* which contains characters, encoded using the Unicode standards. If we can create the character, this datatype can represent it faithfully.

What about studentNumber?

One of the problems of the English language is its ambiguity. Just because we use a particular word to describe something, that word is not necessarily the best for the situation. At first glance you would assume studentNumber is a number.

Java does support many different types of numbers. As we noted above, a bit is the smallest unit of memory. Everything is stored as a collection of bits. Java has a number of primitive datatypes, which are not objects. These include a byte which contains 8 bits (256 values which, when interpreted as numbers, represent integers from -128 to 127), a short which contains 16 bits
(which, when interpreted as numbers, represent integers from -32768 to 32767), an int which contains 32 bits (representing integers from -2147483648 to 2147483647), and a long which contains 64 bits (representing integers from $-2^{63}$ to $2^{63} - 1$). All of these could be used to store studentNumber as long as we know they are large enough (contain enough digits); byte and short are not long enough, since student numbers at Okanagan College contain nine digits. Thus int and long contain enough digits.

Java also has datatypes called float and double, to store numbers containing decimal points. At least we know the studentNumber does not contain decimal points.

This discussion of numeric datatypes is interesting, but is studentNumber really a number?

A good guideline in deciding whether something called a number is really a number or is something else is to ask the question “Is it going to be used in mathematical calculations?” If so, you should store it as some type of number.

But a studentNumber will probably not be used in calculations, so we could just store it as a String as we did with the name of the student.

Thus a Student object has two attributes (the UML term. In Java we will call them instance variables.), both of which are of type String, since studentName is obviously a String. Now the more-complete UML Student representation is as follows.

![Student UML Diagram]

Not only do we see the name of the class, we see the names of the attributes and their datatypes.

**The Professor class**

What is the corresponding UML representation for the Professor class? Recall that a professor also has a number and a name.

![Professor UML Diagram]
We’ll note for the moment that these two diagrams reflect a certain commonality. We’ll explore that commonality later in this chapter.

**Tradeoffs of storing the name as one field instead of as several**

We discussed how to represent names and numbers—that-are-not-numbers earlier, but let’s revisit that discussion and look a little more closely at how to represent a name most efficiently. The most important questions relate to how we will need the name. Will we always use the whole name? Will we sometimes use the parts of the name separately?

I would argue that it is safest to assume we will sometimes need the whole name, and sometimes the parts of the name. Perhaps we need to write a letter that begins “Dear Jane.” Perhaps we need to create a transcript for “Jane Doe”, or maybe the name will be represented as “Doe, Jane.”

If we store the name as one field, we would need to split it up whenever we needed the parts. That may seem to be a straight-forward task, and it’s a well-known algorithm, described below.

An *algorithm* is a series of instructions to solve a problem, in a finite amount of time, using a finite amount of space.

In this case, a possible algorithm is “start at the beginning of the string and look for the first space. The part before the space will give us the given (first) name, the part after the space will give us the family (last) name.” This algorithm expects the string containing the name to have the first name at the beginning of the string, the last name at the end. Is that a sensible requirement?

Does that algorithm really solve the problem? What about Billy Bob Thornton (whose given name is Billy Bob)? Our algorithm does not give the correct answer!

Many Asian names don’t satisfy the conditions of the algorithm. The family name is given first instead of last. Thus the algorithm will say that Yang Changji has a first name of Yang, but that is actually the family name.

A similar algorithm (having similar problems) may be used to find the last, or family, name. “To identify the family name, just start at the end and work forward until we find the first blank.”

Does that algorithm really solve the problem? What about Manfred von Richthofen (whose family name is von Richthofen). It appears we have a flaw in our logic.

Of course, there is also the problem of hyphenated names. Both first names and last names may be hyphenated. How do we handle hyphens?

To get around the fact that breaking a name into its constituent parts is challenging (if not impossible), we should store the first and family names separately, using attributes `firstName` and `familyName`. Since we are modifying our design, we should consider what other name-related fields we should include.
Perhaps a “preferred name” should be there. Not all Alberts like to be called Albert; they may prefer Al or Bert or Bertie. Not all people like to be called by their given name. Cholmondley David Smith may prefer to be called David, or Dave. (Cholmondley is a good English name pronounced Chumly.)

I use some software that uses my full name when it says goodbye; that’s too formal for my taste. So, let’s have a preferredName attribute.

There may even be times when a person’s middle name is required, so let’s have a middleName attribute.

If our program is to be useful in other cultures, we may wish to just use the attributes name1, name2, and name3 and let the user of the program decide what to enter in each.

In addition, there will be times when a person’s full name is required, so we’ll store it as well, but only for use when the full name is necessary.

I will, at least temporarily, use the word student as part of the attribute names for a Student, and professor as part of the attribute names for a Professor. Thus the UML diagrams for our Student and Professor classes look like this.

```
Student
studentNumber: String
studentFirstName : String
studentMiddleName : String
studentLastName: String
studentPreferredName : String
studentFullName : String

Professor
professorNumber: String
professorFirstName: String
professorMiddleName : String
professorLastName: String
professorPreferredName : String
professorFullName : String
```

When we create instances of these classes (Remember that an object is an instance of a class, so we are speaking of creating a Student object or a Professor object), these objects contain the parts of the name. When we have several different objects, several students as an example, each object will contain a student number, unique to that object. In each object, the student number will be
contained in an attribute called studentNumber but, since the objects will each have separate identities (usually each object has its own name, different from the values its attributes contain), it is no problem that studentNumber occurs in each.

While we have used the term “attribute”, an equivalent term when using Java is instance variable. Each instance of the class (each object) has the same attributes, and the value of those attributes can change, or be variable. Hence the term “instance variable” is a reasonable one.

But how do we tell the object what those values are? How do we retrieve those values when we need them?

**Behaviours**

Recall that an object has identity (the name we give the object), state (the values it contains), and behaviours (the things it can do.)

First of all, we should note that “things” is not a very precise word.

I would like to define *behaviours* as “the messages to which an object responds.” We will send a “This is the value for your given name attribute. Remember it.” message to a Student object and then we can, at some later time, send a “Please tell me your given name.” message to that object and it will return the value it remembered.

In Java, methods receive the messages and respond appropriately.

This idea of sending messages to an object is a powerful one, and we can extend it to sending messages to a class, in particular, a message like “Create an object of your class, and here are the attribute values you need.” Most classes contain a method, called a constructor, which responds to this message. It will take all the attributes you supply, process them appropriately, and then return an object for the program to use.

Similarly we will need messages to remember (or set) each of the individual attributes (one message per attribute.) These messages are called *setters*. The values we provide to be remembered are attached to the message through parameters.

We also need messages to retrieve the values of each of the individual attributes (one message per attribute). These messages are called *getters*. Another name for these methods is *accessors*. They access attributes.

Constructors, getters, and setters are usually not shown in the third subsection of a class symbol, the area where methods are shown.

Later on, we will see a special type of class, a JavaBean. For these classes, there is a constructor which takes no parameters, and a setter for every attribute. To make and populate such an object,
first use the constructor to create an object whose attributes have some default values. Then use the setters to specify the values you want. For now, our constructors will take parameters.

Once we have the ability to create an object, we should have the ability to display it, to be sure that it has been created correctly. Every object needs to be displayed, so the class needs a method, usually called toString. This method converts an object to a human-readable form suitable for display, perhaps through printing. We’ll see the code (remember that code is computer-speak for anything written in a programming language.) for this method in the next section of this chapter.

A first look at Java

So now we have a diagram which describes what a Student class contains and what it does. Computers generally don’t know how to interpret diagrams, so we must somehow translate this diagram into Java code.

Here’s part of the code for the Student class. It’s followed by an explanation of the individual statements.

```java
/**
 * A Student
 * @author Rick
 */
public class Student {

    private String studentNumber;
    private String studentFirstName;
    private String studentMiddleName;
    private String studentLastName;
    private String studentPreferredName;
    private String studentFullName;

    /**
     * @param studentNumber 9-digit student number assigned by the college
     * @param studentFirstName Student First Name
     * @param studentMiddleName Student Middle Name
     * @param studentLastName Student Last Name
     * @param studentPreferredName Student preferred name or nickname
     * @param studentFullName Student legal name
     */
    public Student(final String studentNumber,
                    final String studentFirstName,
                    final String studentMiddleName,
                    final String studentLastName,
                    final String studentPreferredName,
                    final String studentFullName) {
        this.studentNumber = new String(studentNumber);
        this.studentFirstName = new String(studentFirstName);
        this.studentMiddleName = new String(studentMiddleName);
    }
```
this.studentLastName = new String(studentLastName);
this.studentPreferredName = new String(studentPreferredName);
this.studentFullName = new String(studentFullName);
} //end constructor

/**
 * @return student number
 */
public String getStudentNumber() {
    return studentNumber;
}

/**
 * @param studentNumber the student number
 */
public void setStudentNumber(final String studentNumber) {
    this.studentNumber = new String(studentNumber);
}

} // end class

Let’s examine the code in detail, starting with the documentation.

**Documentation**

The class begins with some documentation, comments you make to read at a later time, and comments which will be visible to anyone viewing the code for your class. These comments typically explain the purpose of the code and give some information about its contents.

In a Java class, the documentation section begins with /** and ends with */. If you have done any programming before, you may be used to /* to begin a multi-line comment, and */ to end it. While that style still is acceptable, there is a utility program named javadoc which will process your Java program and identify the comments, as long as they are preceded by /** and followed by */. Once it has identified the documentation, it formats it into a readable, useful, form.

Normally the first block of comments in a class will contain information about the class. It should include such information as the author (using the @author tag) and the date the class was originally written, along with a revision history (using the @version tag).

Details on writing comments for the javadoc utility program are available at http://www.oracle.com/technetwork/java/javase/documentation/index.

It is important that you follow these standards for writing comments since javadoc is accessible via BlueJ and you want your documentation to appear as professional as possible.

Important points to note are the following.

- Documentation for a class must appear immediately before the class header. Any import statements, which we will see later, must appear before the class documentation.
Documentation for a method must appear immediately before the method header, the line that gives the name of the method. The description of the method is the first complete sentence in the documentation, terminated by a punctuation symbol like a period.

Documentation for the parameters to the method, or the value returned, may extend over several lines, if necessary.

You may be used to writing single-line comments that begin with //. As you can see above, anything on the same line after // becomes a comment.

**Programming Style – documentation**

Style is a way of expressing something, or the appearance of something. If you search the web using the three words programming style Java, you will find many sources that provide the way to write Java programs. Throughout this book we will include small sections describing appropriate style issues and, more importantly, we will use generally-accepted good style. Since style is important, we will enforce good style by using a tool called Checkstyle.

Documentation is made a little easier when we use BlueJ as it generates some skeleton documentation you for every class and method it creates. That is, it gives you the beginning of the documentation. It gives some ideas of what your documentation should say. As noted above, for every class, you should provide a description of the class (what it represents). You should also provide a “revision history”, who wrote the class and when, plus a description of the changes made by whom and when.

Following the class documentation, we begin to see Java statements.

**Class declaration**

The class header says public class Student.

This marks the first line of the class itself, giving the name of the class (Student), and an indication (the word public) that it can be used by methods outside the class. Of course it must be used by others; that’s the whole point of having classes! Other people don’t need to reinvent your work. Yes, there are private classes and we will use some ourselves, but they are relatively rare.

The body of the class, including its instance variables and methods, are enclosed in a pair of braces, sometimes called “curly brackets.” Whether you place the opening brace on the same line as the name of the class or on the following line is the subject of discussion in some circles. It doesn’t really matter.

**Programming Style – class names**

Class names are nouns, generally single (not plural) nouns. Thus we could have a class names Textbook; we would not have a class named Textbooks. Be careful with your class names; there
are some words in English which are singular but end in the letter “s”. For example, Player, Team, and League are all acceptable class names. Statistics may be an acceptable class name, since the word statistics may be plural or singular, and statistic may not be a meaningful word, depending on the situation.

Remember that class names always begin with a capital letter. If the class name contains several words, all are capitalized, for example DepartmentStore.

**Instance variables**

Looking back at the code, we have six statements allocating memory for the six data items (instance variables) that each object contains.

```java
private String studentNumber;
private String studentFirstName;
private String studentMiddleName;
private String studentLastName;
private String studentPreferredName;
private String studentFullName;
```

Each of these lines is a separate Java statement. A Java statement ends with a semi-colon (;). A Java statement may extend over more than one line, but it must eventually end with a semi-colon. If it does extend over more than one line, you should break it in a sensible place, usually where a space is acceptable.

Each student has a number and name(s). Note that all these instance variables are declared private. The only way the outside world may access these variables is via the getters and setters which we examine below.

Remember that the name of an instance variable begins with a lowercase letter. If the name actually contains several words, the second and subsequent ones are usually capitalized. studentNumber is a good example of this.

After identifying these instance variables as appropriate for a name, I read (in The Globe and Mail, a newspaper published in Toronto, Canada, on, 2006-08-16) about the inhabitants of Norfolk Island.

Descendants of the Bounty mutineers, there are very few family names represented on the island. The telephone phone directory for Norfolk Island includes nicknames for the inhabitants. Here it is: http://phonebook.nf/

Perhaps we should include a nickname attribute!

Using private instance variables with getters and setters is an example of *encapsulation*; there is no way to access the variables except via the getters and setters. This prevents uncontrolled access to the variables.
Programming Style – instance variables

All of these instance variables are of datatype String. That is they are instances of the String class; thus they are objects!

The name you select to give an object should be meaningful. That is, the name should convey some idea of the object’s purpose or role. If so, then you do not need to provide documentation of what the object represents. For example, studentNumber really needs no further documentation (although you may provide it should you wish), while an instance variable named x definitely does need further documentation.

Constructor(s)

Continuing our examination of the Java code, we see that the declaration of the instance variables is followed by a constructor. We know it is a constructor not just because the documentation says so but because it has the same name as the class.

This constructor is executed whenever we need to create a new Student object. A constructor uses the values we provide as parameters to initialize the instance variables for each object we create.

Within a constructor, my programming style is to use the name of the attribute as the name of the parameter. That leads to statements like

```
this.studentNumber = new String(studentNumber);
```

which means, in English, take the value provided as a parameter (the second use of the name studentNumber), create a copy of it (via the String constructor), and save the copy (the equals sign) in another location. The reserved word this means “the current object.” Thus, this.studentNumber refers to the studentNumber instance variable within the current object; that is where the value in the parameter is saved.

Why do we make a copy? To prevent other portions of the program from changing something that they shouldn’t.

The word final is a promise that the constructor will not change the data it is provided.

When you create an object, you are using two areas of memory. The first, only one word (typically 32 bits or 64 bits, depending on the operating system) in size, is the address of the second, a larger block of memory.

Suppose we have two objects, a and b. We could draw their memory use as follows. The larger box refers to the memory occupied by all the instance variables and the methods which the object can use.
But then suppose we have the assignment statement

\[ b = a; \]

Now the memory use is drawn as follows.

Note that we have two references to the same data. In most cases that is a bad idea since you don't have control over what parts of your program are changing the contents of the objects.

Garbage collection is the process by which objects which have no references are made available for reuse. This is done automatically for you in Java.

It is possible to have many constructors in a class, as long as they have different parameters, different that is in type or in number.
You cannot have two constructors, both with two Strings as parameters. But you could have one constructor with two Strings, and another constructor with a String and an int as parameters, and another with only one String.

Perhaps all you know about a Student is the student number and name, but you don’t have the preferredName; the student will provide that later. A second constructor could then be the following.

```java
public Student(final String studentNumber,
                final String studentFirstName,
                final String studentMiddleName,
                final String studentLastName,
                final String studentFullName) {
    this.studentNumber = new String(studentNumber);
    this.studentFirstName = new String(studentFirstName);
    this.studentMiddleName = new String(studentMiddleName);
    this.studentLastName = new String(studentLastName);
    this.studentFullName = new String(studentFullName);
    this.studentPreferredName = "";
}
```

Since we don’t know the preferred name, we simply remember an empty String, signified by the consecutive double quotation marks, with no characters between them.

Recall that we said earlier that a method is Java’s way of handling messages. If you send a message, you must identify the object to receive the message, and we will see how to do that later. You must also identify the message.

Part of the message is its name but part is its parameters. If you ask a friend for dinner, that is a part of the message. The rest of the message involves the choice of the restaurant and the time at which you would like to meet. The name of the restaurant and the time are parameters, the variable parts of the message.

When we send a message to create an object, we usually provide the values the instance variables are to assume. These values are the parameters.

We have seen several assignment statements, the ones using the equals sign, which transfer the values of the parameters into the values of the instance variables, and we have had a simple explanation of how that statement works.

But what does it really mean to say this.studentFullName = new String(studentFullName);?

Start with the names of the variables. When a computer stores data, it uses its memory. Each memory location is assigned a unique address. You could write programs that use those addresses, but it is very difficult to keep the addresses straight in your mind, so today we use an association between names that we understand and the numbers of memory locations. The process of translating our program from Java into something the computer can understand directly (the addresses) is called compiling. The program we use to do the compiling is called a
compiler. The Java compiler is called javac. Of course, the “c” part of its name stands for “compiling”, and the “java” part stands for, well, Java.

When we write our program, we use names for the variables we wish to use. The compiler establishes a correspondence between the names we use and the numeric addresses the computer knows.

An object requires a block of memory, a collection of memory locations, as we saw above. The block of locations is referred to using the address of the first location in the block. An object has an address, and so do each of its instance variables. So too do each of its methods. Thus, when we refer to the variable this.studentFirstName, we are referring to the variable studentFirstName within the current object. Java’s shorthand for “the current object” is this. Thus, this.studentFirstName refers to a specific area of memory within the memory the computer has allocated to the current object.

studentFirstName (without the this) is another, different, variable. It represents the name which we provide to the constructor. studentFirstName is a different area of memory from this.studentFirstName.

So now we have two areas of memory that are involved in the assignment statement, a statement which calculates a value and assigns it to a variable. Now consider the equals sign.

In mathematics, an equals sign means that the values on the two sides of the equals sign are identical. In computing, an equals sign is an instruction to make a copy of the memory associated with the variable name on the right of the equals sign and copy it to the variable on the left hand side. After that copy is complete, the two areas of memory contain identical values.

In fact this is not strictly true. If we were to look inside a String, we would find that it contains an address of memory. If you go to that address in memory, you will find the actual contents of the String. When we say

\[
\text{this.studentFullName = new String(studentFullName);}\]

we are actually saying that the address contained in this.studentFullName should become the same as the address contained in the copy of studentFullName.

We explore this in the exercises to this chapter when we inspect the objects we create.

**Programming Style – constructors**

Giving the constructor and the class the same name is not just style, it’s one of the rules of the language.

You do not need to follow my style of naming the parameters and instance variables identically, but it is a very common style and other styles may confuse other programmers (including your professor) with whom you work.
A different style which you may see is that instance variable names all begin with an underscore. By using this style, you may write

```java
_studentNumber = new String(studentNumber);
```

A third style which you may see is to precede each parameter with a specific letter or phrase. Depending on which style is chosen you see statements like this.

```java
studentNumber = new String(inStudentNumber);
```

or

```java
studentNumber = new String(aStudentNumber);
```

One way to learn how to program is to look at what others have done, and see how they have done it. Thus it is appropriate to say “you may see.” Sources of programs to read are other textbooks, magazines, websites, and open-source projects.

**Getters and setters**

So far we have seen the class header, the instance variables, and the constructor. Continuing our examination of the class, we find the getters and setters. Having the parts of the class in this order is a very common standard, though not part of the language, and you should follow it.

Here example, we have one getter. We know that a method is a getter because its name begins with “get.”

```java
/**
 * @return student number
 */
public String getStudentNumber() {
    return studentNumber;
}
```

Getters return a value to wherever the method was called. As such, a getter must be declared public (allowing the method to be called from outside the class). It must provide a datatype describing the value returned. The body of the getter is very simple. The Java statement to return a value from a method is simply the word return followed by the value.

It is left as an exercise to the reader to provide the other getters, all correctly documented. We need one getter for each instance variable.

Finally, we have one setter. We know that a method is a setter because its name begins with “set.”

```java
/**
 * @param studentNumber the student number
 */
public void setStudentNumber(String studentNumber) {
    this.studentNumber = studentNumber;
}
```
public void setStudentNumber(final String studentNumber) {
    this.studentNumber = new String(studentNumber);
}

A setter must be declared public (allowing the method to be called from outside the class). Setters do not return a value to wherever the method was called, so each must specify a void datatype.

It is left as an exercise to the reader to provide the other setters, all correctly documented. We need one setter for each instance variable.

All the methods we write will have a visibility modifier (the ones we have seen are public and private but there are others), the datatype of the value to be returned (or void when nothing is being returned), the name of the method, the parameter list (which may be empty. We’ll see an example of this momentarily.), and the body of the method.

### Programming Style – getters and setters

As noted above, the name of a getter method always begins with the word get and the name of a setter method always begins with the word set. The end of the method name is the name of the attribute being manipulated, with its first letter capitalized.

Getters and setters, like all methods, will have some documentation. Given their names, though, their documentation may be quite minimal.

#### toString

Earlier we mentioned the toString method, a method which allows us to display the contents of an object in human-readable form. Here it is.

```java
/**
 * @return a Student, as a String
 */
public String toString() {
    return "Student number: " + studentNumber +
           " Student name: " + studentFullName +
           " (" + studentPreferredName + ")";
}
```

Note the structure of this method and how it compares to the discussion above. The visibility is public. The datatype returned is String. The method name is toString. The parameter list, in this case, is empty. Those four parts make up the method header, which is followed by an opening brace. This is followed by the method body, and then a closing brace.

All methods have the same structure: method header, opening brace, method body, closing brace. When I am keying in a method, I always type the opening brace, press Enter twice, then type
the closing brace and press . Then I can enter the body of the method and will not accidentally omit a closing brace.

Note the method’s similarity to the structure of a class. First appears the class header, then an opening brace, followed by the class body, and finally a closing brace.

Since we wish to be able to call toString from elsewhere, it needs to be public. Since it is returning a String, its return type must be String. The name toString is a name that has been adopted as a Java standard and is used by many classes.

Here example, all of the work is done in the statement which begins return. It describes the value the method will send back to the calling method.

```java
return "Student number: " + studentNumber +
    " Student name: " + studentFullName +
    " (" + studentPreferredName + ")";
```

That value is built by concatenating (putting together, one after the other, into a long String) several shorter Strings. The concatenation symbol is the plus sign.

Here, we combine seven Strings. Three are the values of instance variables, two are descriptions of what the instance variable represents (captions), and two are just punctuation.

"Student number: " is a description. It is an example of a string literal, where we specify the value, enclosed in double quotation marks, containing whatever text you would like to display. It will appear as typed, but without the quotation marks. If there is nothing between the quotation marks, you have an empty string, as we saw earlier.

The plus sign (+) indicates that what you have so far (a description) is to be followed by another String, this one being the value of the instance variable studentNumber. One String followed by another one is still a String, and we continue creating the final result. When used with Strings, the plus sign is known as the concatenation operator.

Next we append another description “ Student name: ” and the value of the instance variable studentFirstName. The String which will be our result is getting longer, as we append (add to the end) items. Note that sometimes you prepend, or add to the beginning of an existing String.

Perhaps it would be nice to show the preferred name since it may not be obvious. We will do that in parentheses after the full name. Thus, toString continues by adding an opening parenthesis, the value of the instance variable studentPreferredName, and a closing parenthesis.

Once the complete String is available, it is returned to the method which asked for it.

Note that this Java statement illustrates the idea that a statement may extend over many lines in your editor. However, a literal may not extend over more than one line. That is, the opening and closing double quotation marks for a string literal must be on the same line. If a string literal is
very long, you may wish to break it over two or more lines, using concatenation to combine the portions.

**Programming Style – toString**

All classes should have a toString method. Thus we will be able to ask any object to display itself in a form we can read.

In some cases, there is only a return statement in this method. In other cases, the method may look like this.

```java
// start with an empty string
String result = "";
// many statements that build the String by concatenation
return result;
```

Thus, we have an alternative form for the toString method. There are other alternatives too.

```java
/**
 * @return a Student, as a String
 */
public String toString() {
    String result = "Student number: " + studentNumber;
    result += " Student name: " + studentFullName;
    result += " (" + studentPreferredName + ")";
    return result;
}
```

The expression result += is shorthand for result = result +.

**Creating another class, the College**

Now that you have seen how to create and test classes, let’s create another one, to model the college itself.

What are the instance variables we need?

For now, we’ll model the college very simply; the only instance variables will be the name of the college, its phone number, and the URL of its website. As we progress through this book, we’ll need other instance variables, including the mailing address of the college, a list of the students it enrolls, the people it employs, and the courses it offers.

What is the appropriate datatype for the name of the college? A String, of course.

What is the appropriate datatype for the telephone number of the college? Since we’re not going to do any calculations with the telephone “number”, a String is adequate here, too. If we wish the
telephone number to be formatted with parentheses and hyphens, there is no question about it being a String.

What is the appropriate datatype for the URL of the college’s website? It too is a String. Yes, there is a URL class in the Java libraries, but we won’t use it for now.

Thus, the first attempt at creating a College class could be

```java
/**
 * @author rick
 * @version november 2006
 */
public class College
{
    // name and contact information
    private String name;
    private String phoneNumber;
    private String homePage;

    // constructor
    /**
     * @param name  The name of the college.
     * @param phoneNumber  The college’s main phone number, including area code
     * @param homePage  The address of the college’s home page on the web
     */
    public College(final String name, final String phoneNumber, final String homePage) {
        this.name = new String(name);
        this.phoneNumber = new String(phoneNumber);
        this.homePage = new String(homePage);
    }

    // name and contact information methods
    public String getName() {
        return name;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }

    public String getHomePage() {
        return homePage;
    }

    public void setName(final String name) {
        this.name = new String(name);
    }

    public void setPhoneNumber(final String phoneNumber) {
        this.phoneNumber = new String(phoneNumber);
    }

```
public void setHomePage(final String homePage) {
    this.homePage = new String(homePage);
}

public String toString() {
    return name + '
' + phoneNumber + '
' + homePage;
}

The toString method will display the college name on one line, the phone number on a second line, and the home page on a third line.

The expression "\n" is shorthand for a new line. "\n" is a single character, a char, while "\n" is a String which contains only one character. There is a difference. Can you draw a diagram which shows the difference?

Summary

The process to create the code for a class is as follows.

- Identify the name of the class
- Identify its state, implemented in Java as its instance variables
- Identify its behaviours, implemented in Java as its methods
- Implement the constructor
- Implement the toString method
- Implement the getters and setters

Test and document everything as you go.

That’s it for your introduction to Java. My approach to teaching programming is to discuss a model of something and then build the model. If we need some special feature of language to build that model, we will look at that feature when we need it. Thus, we needed to speak about modelling and Java in this chapter. We needed to talk about constructors, getters, setters, and the toString method.

Now we need to see how to communicate our program to a computer.
Exercises

1. Complete the getters for the Student class.

2. Complete the setters for the Student class.

3. Complete the Professor class, including getters, setters, and toString.

4. In this chapter, we have mentioned several other classes we could develop. These included Room, Furniture, and Building. Select one of those classes and identify the attributes it could contain. What datatypes are appropriate? Why did you select those datatypes?

5. Explore the online documentation for the URL class. The URL for the documentation is given in the chapter. What modifications would we need to make in our College class to use the URL class?

6. One simple class to model is a door. A door has width, height, and thickness. It is either open or closed. Implement a Door class. For a good solution, you will need to use int (or double) and boolean variables. As an additional complication, you may wish to model whether the hinge is on the left or the right.

7. Consider some other real-world classes which you may have seen. An automobile is suitable. So is a digital camera, a digital photograph, or an MP3 file. Select one of these classes. What is its name? What are its attributes? What are its behaviours?
Chapter 2 – Introducing BlueJ

Learning objectives

By the end of the chapter, you will be able to:

- Start BlueJ
- Use BlueJ to create a class diagram
- Use BlueJ’s editor to create and modify classes
- Use BlueJ’s Object Inspector to examine objects
- Define unit tests
- Use BlueJ to create and run unit tests

Introduction

So now we have some source code (the statements we have written in Java) for a Student class and some other source code for a College class. A program is designed to be executed by a computer, so we must have a way of communicating the program to the computer.

There are many tools which will allow us to communicate with the computer. We will use one called BlueJ.

BlueJ

BlueJ is an integrated development environment (IDE). This IDE provides us with many tools:

- An editor which understands the syntax (the grammar or the rules) of the Java language. You use the editor to type the statements in your program. By understanding the rules of the Java language, the editor helps by colouring reserved words, by showing you matching braces and parentheses, and by indenting statements according to standards.
- A simple UML diagramming tool. BlueJ draws class diagrams but none of the other diagrams which are included in the UML.
- A tool for creating and testing classes. Testing is crucial to ensuring that your code works correctly. Having it included in the environment is very important.
- A link to a Java compiler. The computer can not directly understand Java statements. A compiler is a program which translates Java statements into something the computer can understand.
- A debugger (which we can use to fix hard-to-find errors.)
An integrated development environment provides access to all these tools through one common interface; it is integrated!

If you are doing your programming at a college or a university, BlueJ will probably have been installed on the computers you will be using. If you wish to do some work in Java elsewhere, you will need to install BlueJ and Java on your personal computer. That task is included as an exercise at the end of the chapter.

Once BlueJ and Java are installed, you can continue to create a new project.

**Creating a new project**

Start BlueJ, select Project from the main menu, and New Project from the drop-down menu. Give the project a meaningful name. Each project is stored in a separate folder on your computer. You select the precise location when you create the project. If your operating system allows a space in the name of a folder, BlueJ allows a space in the name of the project.

The project skeleton that BlueJ creates already contains one file, described in the next paragraph. BlueJ displays a class diagram, and places that file on it. The title on the window containing the class diagram is the name BlueJ followed by a colon and the name of your project. The icon for the file looks like a piece of paper, so you may decide to double-click it to see what it contains.

The file is a project “readme” file, and the first few lines of it tell you its purpose. “Here, you should describe your project. Tell the reader (someone who does not know anything about the project) all he/she needs to know.” Then it identifies some headings under which you can place information for the reader. This is the beginning of the documentation you will be creating.

To place a class on the class diagram, you have several choices.

- From the main menu, select Edit, New Class, or
- From the column of buttons to the left of the class diagram, select New Class, or
- Right-click a blank area of the class diagram and select New Class, or
- Press Ctrl+N. That is, press and hold down the Ctrl key, press N (either uppercase or lowercase), release N, and release Ctrl.

Whichever you select, the New Class dialogue appears.
Type the name of the class, here it's Student, and click Ok (or press Enter.) BlueJ updates its class diagram. Note that Java requires that class names contain no spaces. You could use underscores to separate individual words in a class name, but the standard, as we have seen, is to capitalize each word of the class name which contains more than one word.

The class displays diagonal lines as a visual reminder that it has not been compiled. Since it hasn’t been compiled, we can’t use any of its features. Right now, that’s fine since we don’t know how to compile it. But later on, when you have many classes, and are making changes to several of them, it’s good to know which have been compiled and which have not.
To see the contents of a class, the documentation and Java statements it contains, right-click the class and select Open Editor (or double-click the class.)

BlueJ provides you with a skeleton of a class, some documentation, the declaration of an instance variable and a method. Keep what you need, delete the rest. Don’t delete the first few blank lines; we’ll need them later.

Note that you can change the skeleton, if you wish. To do that, go to the folder in which you installed BlueJ and then to lib/<<language>>/templates/newclass. Note that <<language>> represents the language in which BlueJ’s interface appears; in this book, that would be English, so the folder is lib/english/templates/newclass.

The file you wish to change is stdclass.tmpl, a file which provides a template for a standard class. Open that file in any text editor, make your changes, and then save the file. Here is the template I prefer.

```java
PKGLINE
/**
 * Write a description of class $CLASSNAME here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class $CLASSNAME
{
    // instance variables - replace the example below with your own
    private String x;

    /**
     * Constructor for objects of class $CLASSNAME
     */
    public $CLASSNAME(final String x)
    {
        // initialize instance variables
        this.x = new String(x);
    }

    /**
     * @return the instance variable
     */
    public String getX()
    {
        return x;
    }

    /**
     * @param x the new value for the instance variable
     */
    public void setX(final String x)
    {
        this.x = new String(x);
    }
}
```
Enter the Java code for the Student class which we seen above. You may copy it from a previous page, or you may type it yourself. I'd suggest that you type it yourself do that you become more comfortable with the type of statements you need. If you insist on copying from the textbook, be aware that some quotation marks may not copy correctly and you'll need to correct them in BlueJ.

After you have entered the declarations for the instance variables, and the constructor, stop. Click Class, Save (or press Ctrl+S) to save your work.

Look at what you have typed. Are there any obvious errors? If so, fix them. Obvious errors will make themselves obvious by unexpected indenting, or unexpected colours.

There are a number of advantages to using an IDE. Since the IDE knows the language you are using it can make intelligent decisions about indenting. It can display different aspects of the language in different colours. If you wish to suppress these helpful features (why would you want to?), select Tools from the main menu, then Preferences from the drop-down menu. On the Editor tab, uncheck the appropriate combo boxes. When you first run BlueJ, you will find that most of the combo boxes are checked. I leave mine that way, possibly checking “Make backup of source files” also.

IDEs can also match parentheses and braces.

That is, it will if you ensure that Tools, Preferences, Editor, Match Brackets is checked.

You should also check that Tools, Preferences, Editor, Auto-indent (Enter and tab keys) is checked. If it is, your code will be indented the way it is in the chapter. If not, all lines begin in column 1. To correct that problem at any time while editing a class, press CTRL + SHIFT + I.

When you type a closing parenthesis or brace, the IDE will momentarily highlight the matching opening one. When you type an opening double quotation mark, the screen will show red characters until you type the closing double quotation mark.

If you have forgotten your punctuation symbols, an opening parenthesis is (. A closing parenthesis is ). An opening brace is { and a closing brace is }. The double quotation mark in programming is " and the same mark is used at the beginning and end of all strings. In typesetting there are two double quotation marks, an opening one “ and a closing one ”.
Click the Compile button to first save and then compile the class. If you have made any errors, the compiler will find them and BlueJ will highlight (in yellow) the line containing the first one. If you see the error immediately, fix it and compile again. If you don’t see the error, click the question mark in the lower-right corner of the BlueJ window, and BlueJ will offer some hints about what the error may be.

There will be at least one error detected, since the instance variable named x has not been declared, but is used in a method. That is because we are in the middle of creating the class.

Sometimes one error will cause many others, so don’t worry about fixing every error at the same time. Sometimes one error will hide another, so you’ll see more errors after you fix one than you had before. But never fear, you’ll soon reduce the number of errors to zero.

When you have no more errors, continue entering the other getters and setters. Don’t forget getters and setters for each instance variable you created. Remember that the editor allows you to cut/copy and-paste, and search-and-replace. These are particularly useful when entering similar methods. But compile before you cut/copy and paste; there is no point in copying code which does not compile.

Enter the toString method as well.

The order in which you list the declarations of the instance variables does not matter. Nor does the order in which you list the methods. It's a matter of personal preference.

Following the instance variables, I list the constructor(s). Then I list the getters and setters. Sometimes I list all the getters and then all the setters, sometimes I do the reverse. And sometimes I list a getter followed by its setter, or vice versa. It depends on the style you (and your teacher) prefer.

I usually list the toString method after the getters and setters.

When you are finished typing, select Class, Save from the menu in the window containing your class. This saves your work but does not compile it.

Later on we will have many classes open at the same time. Then it may be more efficient to select Project, Save from the menu in the main BlueJ window, and save all the classes you have open.

Now all your work has been saved should you need to leave. Assuming you haven’t left (or you have returned and opened the project again, using Project, Open Project, or Project, Open Recent), click the Compile button again.

Note that there are two Compile buttons, one in the window where you have entered the class, and one in the main BlueJ window. The one in the window where you have entered the class compiles only that class. The one in the main BlueJ window compiles all uncompiled classes. To
show which is being compiled at a specific time, BlueJ changes the colour of the class from light brown to a darker brown.

**Virtual machines and bytecode**

Every type of computer uses its own instruction set for the commands it executes. These instruction sets are very complicated and depend on the design of the CPU your computer is using. This normally makes it difficult to run a program on one type of CPU when it was designed for another.

A main principle behind Java is that you can write a program on one type of machine and have it run on others, without change.

This miracle works because there is a *virtual machine* involved. This Java virtual machine is just a program which pretends to be a computer running its own language. It translates the instructions output by the Java compiler (known as bytecode) into the appropriate instruction set of the host computer.

When you compile a Java class, you translate from the Java statements themselves (stored in a .java file) into the instruction set of the virtual machine (stored in a .class file).

Due to the virtual machine, you can develop Java programs using a Macintosh computer, as an example, and run it on a PC running Windows. In the old days that was quite a thrill. Now, however, it’s not such a big deal. The big deal is to develop on a Macintosh and then run on an IBM mainframe using the Linux operating system.

Yes, you can do that too.

**Testing the Student class**

An article in the September 2010 issue of the Communications of the ACM (Injecting errors for fun and Profit) began with an interesting quote. “That which isn't tested is broken.”

One of the features in my textbook is its insistence on testing the methods you write. Here begins the first of many sections dealing with testing.

Recall that the methods in the Student class (and the class itself) needed to be declared public so that some other methods can use them. The benefit of using a tool like BlueJ is that we don’t need to create any of these other methods right now. BlueJ takes care of the details by magic, in the background. Well, it’s not really magic. Later we will lift the curtain and see what is in the background.

The noted science fiction author Arthur C. Clarke said that “Any sufficiently advanced technology is indistinguishable from magic.” ([http://en.wikipedia.org/wiki/Clarke%27s_three_laws](http://en.wikipedia.org/wiki/Clarke%27s_three_laws)) I use magic in that sense.
The reference to “lifting the curtain” comes from the movie The Wizard of Oz.

For now, simply right-click the Student class in the class diagram. A menu appears showing the public methods of the class, plus Open Editor, Compile, Inspect, Remove, and, separated by a line, Create Test Class. We will explore those options later. For now, simply click the name of a constructor. Constructor names are highlighted because they are the only methods which you can call now. The getters and setters, and toString, cannot be used until we have created an object.

Recall that a constructor allows you to create an instance of the class, an object. By clicking the name of the constructor, you run the constructor.

First, however, you need to specify the name of the object; student1 is acceptable for now, but you can change it if you wish.

Then you need to specify the parameters the constructor expects. Both are done through the window shown below.

Remember that all the values being passed to the constructor are Strings, so should be enclosed in double quotation marks. Remember that you can use an empty string (""") to represent a value you do not know or do not wish to provide. At the moment, you may use an empty string for any or all of the parameters.

Note that the documentation you placed at the beginning of the method appears in the dialogue, to guide you in providing parameter values. Maybe we should have used better documentation!

Click Ok when you have specified all the parameters.

Once BlueJ has created an object, it appears in the object tray at the bottom of the BlueJ window.
Create a couple more students, at least one of which does not have a preferred name. Try some experiments with them.
To experiment with an object, right-click it in the object tray and execute one or more of its methods. Note that the methods are listed in alphabetical order, regardless of the order in which you placed them in the class.
The menu choice, inherited from Object, allows you to execute any methods which the class Object contains. Object is the base class (parent class) for all classes we will create. A child class can do everything its parent can, as well as having additional capabilities, the child's methods.

Note that the Object class contains a toString method, as does Student. We say that toString in Student overrides toString in Object. BlueJ says that toString in Object is *redefined* in Student.

Be sure to execute the toString method for the Student objects you created. You will get the same result if you click toString in Object or toString in Student.

See how the output of toString changes depending on whether or not there is a preferred name. Use two different Student objects, one with a preferred name and one without, or create one Student object and then change its preferred name.

Try executing some of the getters and setters you wrote. Do they produce the correct answer? If not, fix them so they do produce the correct answer.
**Inspecting an object**

Recall that in the previous chapter we talked about the meaning of the equals sign, and how one string contains the address of another. We can see this, now that we have objects on the object tray.

Right-click an object.

Select **Inspect**. The Object Inspector window appears. It shows the values you specified for the parameters. Were you like me and you specified your own name? Or were you more creative?
Highlight any of the six instance variables by clicking them, and then click Inspect. For the next two illustrations, I’ll assume we left studentNumber highlighted.

Now we are looking under the hood, at how Java represents a String. The line reading private char[] value shows an arrow. This is a reference to, or the address of, another area of memory. That area actually contains the contents of the String. If you click Inspect in the dialog, you will see the image below, showing the characters which make up the student number. I entered a two-digit student number.

Now that we are as deep into a String as we can go, the Inspect button is disabled. We see that the String actually has two instance variables of its own. The first (length) is the number of characters
in the String. I only entered two characters. The second (unnamed) is an array that contains the actual characters I entered.

In a similar way, we can see how the studentFullName is stored.

Here, there is too much information to show in a window of the default size, so a scrollbar allows you to see the rest of the information.

Click Close as many times as necessary to close the Object Inspector.

**Unit testing - definition**

The testing we just did to see that the object was created correctly is a form of *ad hoc* testing. *Ad hoc* testing involves components individually, with whatever data came to mind. Normally we wish to test all the parts (the units) of the system, and we may need to test some many times. *Ad hoc* testing would be very difficult; a better way to test is to do systematic unit testing.

Wikipedia (http://en.wikipedia.org/wiki/Unit_testing) tells us that “a **unit test** is a procedure used to validate that a particular module of source code is working properly. The procedure is to write test cases for all functions and methods so that whenever a change causes a regression, it can be quickly identified and fixed. Ideally, each test case is separate from the others; … This type of testing is mostly done by the developers and not by end-users.” The word “regression” refers to the need to redo previously-passed tests and ensure that the code still passes them.

Suppose you made a major change and want to make sure that you have not broken any other code.

Breaking code refers to having previously-working code stop working due to some change you have made. When you change a datatype from one type to another, I guarantee you will break
some other code. When you change the order by which you sort some data, you’ll probably break code.

Yes, I did say earlier that you would not break anything. I meant you wouldn't break any hardware. If you break software you have the tools to fix it.

After a change, you will perform regression testing. To do this, create and save a series of unit tests so that we can use them over and over again as necessary. BlueJ provides access to a tool named JUnit which facilitates this testing. JUnit is a testing framework, described in more detail at www.junit.org. The examples in this book use the latest version of JUnit which is included with BlueJ, JUnit 4.

**Unit testing with BlueJ**

Let’s create a series of unit tests for the class Student. “How do you know what to test?”, Riley asked me one day.

The short answer is “Everything!” But that answer is not very useful.

How do you test a constructor? You test that the object has been created (is not null) and that the instance variables have the correct values. That is, in the process of testing the constructor you also test the getters.

A getter works when it returns the value you expect. Hint: when you create an object, ensure that all the constructor's parameters have different values. That way you can expect the getters to all return different values.

How would your unit test detect an error in this constructor if you provided two parameters having the same values?

```java
public SomeObject(final String param1, final String param2) {
    this.param1 = param2;
    this.param2 = param1;
}
```

How do you test toString? Once you know that the getters are working properly, you can test that toString works; you know that the instance variables have the correct values. toString works when it returns the value you expect. You may need several tests for toString if toString contains if statements that cause differing output. For example toString may return one value if a student has a preferred name (or nickname) and a different value if not.

How do you test a setter? Once you know that the getters work (you tested them as part of the test of the constructor), you can write a test which changes an instance variable (using the setter) and then looks at the value the instance variable holds (using the getter, which you know works). If the test fails, then the problem is not in the getter, it's in the setter. Unless you wrote the test incorrectly, but there is no way to detect that automatically.
As we work through this book and its College project we'll see many other places where we need to test. If you pass a piece of “bad” data into a method (often a constructor or a setter), is the correct type of exception thrown? Does the exception contain the correct message? Exceptions will be a large part of this project.

Sometimes it will be harder to write tests. How do you ensure that the contents of the file your program creates are correct, particularly when the file contains 100 lines of output? How do you test a graphical interface? Standard unit tests may not be useful in those cases.

But let's start writing some tests for our Student class, a class which will have relatively-simple unit tests.

Right-click Student in the class diagram and select Create Test Class.

If “Create Test Class” does not appear, use the main BlueJ menu and select Tools, Preferences, Miscellaneous, and click “Show unit testing tools.”
Then click Ok to dismiss the Preferences dialog and right-click the Student class again. Select Create Test Class.

BlueJ names test classes by taking the name of the class being tested and appending the word Test. The test class for Student is thus named StudentTest.

This may not look exactly like what you see. My test class was drawn too close to the top of the window; when that happens, click on the class with which the test is associated (here the Student class) and drag it down a little. The test class will follow along. You can’t do the same thing by dragging the test class.

Note that BlueJ continues its practice of colour-coding parts of the diagram. Unit tests are green. If you miss the hint provided by the colour, a class is clearly shown as a unit test by the words unit test, enclosed in guillemets.

Guillemets are used in some languages, including French, to mark the beginning and end of speech, much as English uses the double quotation marks. These symbols have been adopted in the UML.

When writing about guillemets you may sloppily use « or », but the correct symbols are « (CTRL+`,< in Microsoft Word. Note that is a back quote, the one found on the tilde key.) and » (CTRL+` ,> in Microsoft Word).
Open StudentTest in the editor. Document it properly, adding author and version information.

To do a test, we need some objects. One possibility is to create the objects anew in each test, or we can create them once and then use them in all the tests. My preference is the latter, as long as we restore them to a known state before each test.

In the unit test class, before the default constructor and its comments, declare the objects you’ll be using. These objects will exist during all of the tests and can be accessed from any of them. These declarations are of the form `datatype variableName;`. In our case, enter `Student s;`

Yes, I know that I previously insisted on meaningful variable names. In this context `s` is a meaningful variable name. And so are `s1`, `s2`, and `s3` if you need several objects! As a general rule, in unit tests, I will generate variable names by taking the first letter of the class being tested along with, when necessary, a digit, to distinguish between objects.

This unit test class skeleton contains three methods. Generally, we need to write nothing in the constructor.

In the `setUp` method, we describe any processing, generally initialization, which must take place before each test. In this test, we need to create a Student object. The variable was declared, but it doesn’t have an area of memory associated with it, so it has no value. Here we give it a value.

```java
s = new Student("123456789", "Richard", "Dennis", "Gee", "Rick", "Richard D. Gee");
```

As suggested earlier, all the parameters have different values.

If we initialized the object when we declared it, and then did the unit tests in varying orders, we could change the state of the object. We would not know for sure its state (the values of the instance variables) before a test and so we would not be able to forecast the results of the test beforehand.

By initializing the objects in the `setUp` method, the tests are independent of each other. That is, the outcome of one does not affect the outcome of another. By doing the initialization in the `setUp` method, we are guaranteed to know the state of the object before each test.

Note the use of the word “beforehand” above. We will not run a test and then look at the result, deciding if it is correct. We will determine what the result of the method should be before testing it, and place the result we expect in the test. This sounds an unusual way to do things, but you will see that it works.

In the `tearDown` method, we describe any processing which must take place after each test. Generally, we need to write nothing in `tearDown`.

Then create the tests you wish to perform. Each test is similar. Every test:

- Has public visibility.
- Returns nothing (or void).
Has the annotation @Test.

Has a name. In JUnit 3, the name consists of the word test followed by the name of the method being tested. In JUnit 4, I still use that naming, but you could use the name of the method being tested. If I need several tests for the same method, I place a sequence number (or occasionally, a letter) at the end of the method name. Thus, for example, we will need testToString1 and testToString2 since we need to test that method twice, once for a student with a preferred name and once for a student without a preferred name.

Carries out some processing and produces a result.

Uses an assert method to check that the result matches the expected result. The many possible various assert methods available are shown at http://kentbeck.github.com/junit/javadoc/latest/org/junit/Assert.html

Here's a simple to ensure that the Student object was created.

```java
@Test
public void testConstructor() {
    assertNotNull(s);
}
```

This test simply tests that the object was created. It does not test anything about the values the instance variables contain. The getters (and hence the instance variables) can be tested within this first test.

```java
@Test
public void testConstructor() {
    assertNotNull(s);
    assertEquals("123456789", s.getStudentNumber());
}
```

as noted above, or they can be tested in separate tests, one or more tests for each getter.

Here's a simple test for the getStudentNumber method.

```java
@Test
public void testgetStudentNumber() {
    assertEquals("123456789", s.getStudentNumber());
}
```

To test a getter, we start with an object whose state (instance variables) we believe we know and then ask that object to execute its getter (in this case, we ask s to execute getStudentNumber()). We use the name of the object followed by a period and the method we wish to use.

Since the attributes have private visibility, we can only access them directly from inside the object. To retrieve them from outside the object we use a getter.

To modify them from outside the object we use a setter.
Then we use the `assertEquals` method (with the expected value as the first parameter, and the actual value as the second parameter) to check that the student number returned is the expected value. `assertEquals` is one of many methods provided as part of JUnit.

Note that if this form of an `assertEquals` fails, we will see an error message that begins with the word `null`. The error message is not saying that the value we are testing is `null`, it is saying that we have not provided text for an error message.

That is, it is saying we have used the two-parameter form of `assertEquals` rather than the three-parameter form.

The three-parameter form is `assertEquals("message", expected value, actual value);

You may use either form, but I will stick with the two-parameter form.

Once we are sure that `getStudentNumber` is working correctly, we can test that the `setStudentNumber` method also works correctly. The test for the setter involves the getter as well, so we need to test the getter first.

```java
@Test
public void testSetStudentNumber() {
    // change the student number
    s.setStudentNumber("678678678");
    // confirm that the change took place
    assertEquals("678678678", s.getStudentNumber());
}
```

While the test seems overly simple, you may find it fails if you made a typing mistake which nevertheless was correct Java. We have already seen one of these, when we confused `param1` and `param2`. Consider the following method.

```java
public void setStudentNumber(final String stuNumber) {
    this.studentNumber = stuNumber;
}
```

This will compile but the test will not give the correct result since the value of the parameter is never saved. The parameter is never used in the body of the method because you made a typing mistake.

When there is confusion between the name of an instance variable and a parameter, the instance variable must be prefixed with this. If there is no confusion, this may be omitted. In the method shown, the parameter and the instance variable have different names, so there is no confusion. It would be correct to say

```java
public void setStudentNumber(final String stuNumber) {
    this.studentNumber = stuNumber;
}
```
But that violates our (my) standard way of writing constructors.

In this example, both this.studentNumber and studentNumber refer to the instance variable.

Here are the unit tests for the studentNumber getter and setter.

```java
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

/**
 * The test class StudentTest.
 *
 * @author Rick
 * @version May 2012
 */
public class StudentTest {
    Student s;
    /**
     * Default constructor for test class StudentTest
     */
    public StudentTest()
    {
    }

    /**
     * Sets up the test fixture.
     *
     * Called before every test case method.
     */
    @Before
    public void setUp()
    {
        s = new Student("123456789", "Richard", "Dennis", "Gee", "Rick", "Richard D. Gee");
    }

    /**
     * Tears down the test fixture.
     *
     * Called after every test case method.
     */
    @After
    public void tearDown()
    {
    }

    @Test
    public void testConstructor() {
        assertNotNull(s);
        assertEquals("123456789", s.getStudentNumber());
    }
```
As part of the first test, we use the assertNotNull method. This method comes in two forms; the one I use most often takes one parameter, the name of the object we expect to be not null. If it is null, the test will fail. The second form has two parameters; the first is a message which can be displayed when the object is null.

As part of each test, we use the assertEquals method. This method comes in many forms; the one we use most often takes two parameters. The first parameter is a String containing the expected value; the second parameter is a String containing the actual value returned from a method. That is, the first parameter is the answer we expect, the second is what the program gives us. Of course, they should be the same. If they are not the same, the assertion will fail, and the test will fail. JUnit will indicate that very clearly.

A second form of assertEquals has three parameters; a message to be displayed if the test fails, the expected value, and the actual value.

In testConstructor, we know the student number we used to create the student, so we test that the value returned by getStudentNumber is the expected value.

In testSetStudentNumber, we change the student number and then retrieve it. We know what the expected value is, so we test that the correct value is returned. Technically this is perhaps better called an “integration test” since we are combining different methods and seeing that they work together.

When we use JUnit in more advanced situations, we’ll see some other assert methods. These include assertNull, and assertTrue. We’ll also see a method named fail.

For details on all the assert methods, look at the documentation at http://www.junit.org and follow the link to javadoc or go directly to http://kentbeck.github.com/junit/javadoc/latest/org/junit/Assert.html. For an alternative introduction to JUnit, see http://junit.sourceforge.net/doc/cookbook/cookbook.htm.

Notice that when you compile a class, the corresponding unit test class (if it exists) is also compiled. The reverse is not true; that is, you can compile a unit test without compiling the class it is testing.

If you look closely at the code above, you will see methods in which the opening brace is on the same line as the method name (testConstructor is an example) and methods in which the opening brace is on the next line (tearDown is an example). Whichever you use doesn't matter.
I prefer the first, but there is a good argument for using the second; the opening and closing braces should be indented to the same level.

Unit testing – the results

StudentTest is on the class diagram, so we can right-click its class symbol. The resulting menu offers the possibility of running all the tests, or just one.

When you elect to run all of the tests (usually a good idea), a window (shown below) appears showing the results of all the tests, one per line. (Of course a scroll bar appears in the upper portion of the window when there are many tests.)
A successful test is shown with a green tick beside it.

An unsuccessful test (you will see some soon, I am sure), one that did not complete (crashed) for some reason, is shown with a black X beside it, if there is an error message available (You used an assert method which included a message as a parameter), or a red X is there is no error message available. A common reason for an unsuccessful test is a typing mistake. Another common reason for unsuccessful tests is an unexpected value of null for an instance variable. null means the variable has no value, while you thought it probably had a value.
For a failed test, click the message in the upper pane and abbreviations of the expected and actual values are shown in the lower pane.

Note that a test may fail because the test was created incorrectly or because the method it is testing is incorrect. For example, in testConstructor, I mistyped the expected value for the student's middle name. Of course the assert method failed.
The test explanation shows how and where the expected and actual values differed. In this instance, they begin and end with the same characters, but the expected value contains an uppercase E in the second position, while the actual value contains a lowercase e. Note that the problem is in line 49 of the file StudentTest.java.

You want to see a green line across the middle of the test window; this tells you that all of the tests were successful. When any tests fail, the line is red.

Below the line, is the number of tests run, and the number of errors detected or tests that failed.

We wish to test each method in the Student class. There are six getters, one for the student number and five for the parts of the name, and six setters. The getters may all be tested as part of testConstructor, but the setters need separate tests. We need seven unit tests and must ensure each test works successfully. Cut and paste, and be careful as you make changes!

To test the toString method, we need more tests, one applied to a student with a preferred name, one applied to a student without a preferred name. As noted above, these could be called testToString1 and testToString2. If you prefer, you could use the names testToStringWithPreferredName and testToStringWithoutPreferredName.

Testing toString is a little challenging, since you need to remember all the literals (especially considering spaces and \n and \t (what does that do?)) which are added to the instance variables.

Thus you need to implement 14 tests to ensure your class behaves correctly in all cases.

Make it so.

**Smoke testing**

These simple unit tests are exhaustive. That is, they test everything an object can do.

An alternative type of testing is “smoke testing.” A simple definition of smoke testing is provided by [http://searchvb.techtarget.com/sDefinition/0,,sid8_gci930076,00.html](http://searchvb.techtarget.com/sDefinition/0,,sid8_gci930076,00.html)

Smoke testing is non-exhaustive software testing, ascertaining that the most crucial functions of a program work, but not bothering with finer details. The term comes to software testing from a similarly basic type of hardware testing, in which the device passed the test if it didn't catch fire the first time it was turned on. A daily build and smoke test is among industry best practices advocated by the IEEE (Institute of Electrical and Electronics Engineers).

JUnit allows us to create many tests, and run only selected ones. When we run only a few of the tests, we are doing smoke testing.
Testing is a good practice; using GOTO statements, as the programmer in the cartoon did, is not a good practice in the object-oriented paradigm.

The Professor class

In a similar manner, we must create the Professor class and unit tests for all the methods in the Professor class.

Make it so.

The College class

In a similar manner, we must create the College class and unit tests for all the methods in the College class. You created the College class earlier. Ensure that all methods are tested and pass the tests.

BlueJ minutiae

There are some details about BlueJ that you should know if you wish to customize it to your preferences. These details generally refer to one of the configuration files BlueJ uses. The file bluej.defs applies to all users, bluej.properties applies to only you, the current user.

For simplicity and generalness, assume BlueJ has been installed in a folder which I will reference as <blueJ-home>. On my Windows machine, this is C:\BlueJ, and bluej.defs lives in <bluej-home>\lib. I will reference your home folder as <your-home>. On my Windows machine, this is Z:\Documents and Settings\300100122\ (don’t ask!), and bluej.properties lives in <your-home>\bluej.

The information in bluej.properties overrides any information in bluej.defs and will remain unchanged when you update to another version of BlueJ.

We have already see how to ensure the unit testing tools are available. Here’s another way to make the unit testing tools visible, along with some other customizations.
Unit testing tools
Open bluej.defs (since we are going to change a setting which applies to all BlueJ users.) in an editor. Search for the phrase “testing.showtools”, without the double quotation marks. In the statement you find, change the word “false” to “true.”

Font size
This one is easy. Tools, Preference, Editor, Font size.

Alternatively, open bluej.defs (if you want to change the font size for all users or bluej.properties if you want to change it only for yourself) in an editor and search for “editor.fontsize.” Change to the font size you prefer. The poorer your vision, the larger the number you should specify.

Line numbers
This one is easy. Tools, Preference, Editor, Display line numbers.

Alternatively, open bluej.defs (if you want to display line number for all users or bluej.properties if you want to change it only for yourself) in an editor and search for “displayLineNumbers.” Change “false” to “true.”

Colours
You can use the slider on Tools, Preference, Editor to change the intensity of the colours used for scope highlighting. The scope of a variable refers to the portion of our program which knows about the variable.

A variable declared within a method is only accessible within that method, so the method is its scope.

A private instance variable is accessible anywhere within the object; the object is its scope.

A public instance variable is accessible anywhere within the object and outside it as well.

To change the colours themselves, and not just their intensity, you are out of luck!

However, you can change the colours which are used to display Java statements and documentation. In the <blueJ-home>/lib folder, open the file moe.defs in a text editor. The last few lines define the colours used by BlueJ.

```plaintext
# Syntax colour definitions
# =========================

# Key to values
# ---------
# comment Single line comments (//) and standard multi-line comments (/* */)
# javadoc Multi-line javadoc comments (/** */)
# keyword1 Standard Java keywords (e.g. abstract, final, do, if, else, new, catch etc.)
# keyword2 Class creation keywords (package, import, class, interface, extends, implements)
```
# keyword3 Remaining Java keywords (this, null, super, true, false)
# primitive Java primitives (int, float, double, char)
# string String literals (anything in "quotes")
# label Labels for loops or in switch/case statements
# invalid Unclosed string literals or other detected errors
# other Anything else
# background Editor background colour

# Any of the values above that are not defined are given the BlueJ default colours.

# Key to colours
# ---------------
# Each colour should be given a six digit hexadecimal value of the from rrggbb where
# the pairs of digits refer to the red, green and blue values respectively.

comment = 999999
javadoc = 000099
stand-out = ee00bb
keyword1 = 660033
keyword2 = cc0000
keyword3 = 006699
primitive = cc0000
string = 006600
label = 999999
invalid = ff3300
other = 000000
background = ffffff

The one which I would most like to change is the colour for comments. The default colour is 999999. This is the RGB equivalent of a medium gray. I don't like it since it doesn't stand out. Thus, I'd prefer a colour like FFFF00 (a bright yellow) or FF0000 (a bright red) or FF6600 (orange) or 006633 (green). If I use 006633 I'll need to change the colour of strings as well, since 006633 is close to the default colour for strings.

The changes I've made to moe.defs are as follows.

#comment = 999999
comment = 006633
#string = 006600
string = FF6600

I commented out the original values and then added my own values.

To understand these colours and the strings representing then, search the web for RGB. We will see this way of representing colours when we design GUIs and applets in later chapters.
Default code

As you have already seen, when you create a new class, BlueJ provides some skeletal code. If you don’t like that skeleton, look in the `<blueJ-home>/lib` folder for a folder with name of the language in which your menus appear. Within that folder (in my case C:\BlueJ\lib\english) is a folder named templates. Within templates are several files, each with names ending in .templ. There is also a folder named newclass; it too contains templ files. You may open them in an editor and change them to your liking. For example, here is my personal version of stdclass.tmpl, from the newclass folder. I may start using it as the default for my students, too.

```java
$PKGLINE
/**
 * Write a description of class $CLASSNAME here.
 *
 * @author yourName
 * @version Created (a date)<br>
 * modification history<ul>
 * <li>the date and what was changed</li>
 * </ul>
 */
public class $CLASSNAME {
  // instance variables
  String x;

  /**
   * Constructor for objects of class $CLASSNAME
   */
  public $CLASSNAME(final String x) {
    // initialize instance variables
    this.x = new String(x);
  }

  /**
   * Getter for an instance variable.
   * @return x
   */
  public String getX() {
    return x;
  }

  /**
   * Setter for an instance variable.
   * @param x The new value for the instance variable
   */
  public void setX(final String x) {
    this.x = new String(x);
  }

  /**
   * Create a human-readable version of the object.
   * @return a String which represents the object
   */
```

public String toString() {
    String result;
    // construct the resulting string
    result = x;
    return result;
}

Summary

That’s it for your introduction to BlueJ. As I stated earlier, my approach to teaching programming is to model something and then implement the model. This is sometimes known as contextual teaching.

BlueJ and unit testing are crucial to building the model.

An untested model is a useless model. No architect would build a structure without creating a model first.

Of course the model may show flaws. That’s better than building the structure and then finding its flaws. The Tacoma Narrows Bridge shows what can happen when your model has flaws which you do not catch before you build. 
http://en.wikipedia.org/wiki/Tacoma_Narrows_Bridge#Film_ofCollapse

Did you see how strange the output of toString looked when a student did not have a preferred name? We need to fix that. Thus, the next feature we need is some way of taking one of two branches through a method, depending on some condition. This will also allow us to deal with some of the unit tests mentioned, checking if a value satisfies some conditions.
Exercises

1. Develop unit tests for all the methods in the Student class. Ensure that your code passes all the tests. My style is to develop the tests for the getters first, since they do not depend on the setters. The values of the instance variables are all set by the constructor.

   Test the toString method. It also does not require any setters.

2. Develop unit tests for all the setters in the Student class. Ensure that your code passes all the tests. Note that the unit tests for the setters will require the getters. Thus you create the getters first, test them, and fix any errors they contain before you begin to write the setters.

3. Create the Professor class. As you do so, develop unit tests for all the methods in the Professor class.

4. Download (from bluej.org) and install BlueJ on your laptop or other home computer. Download and install the Java SE Development Kit. Download (also from and install the Java class documentation so that you do not need to be connected to the Internet while using BlueJ, unless you want to be, of course. Use Tools, Preferences from the main menu, and then the Miscellaneous tab to tell BlueJ where the documentation is installed locally.

5. Use BlueJ to create a Student object on the workbench. Inspect it and see how the attributes are stored. In particular, what is stored when you provide an empty String for a parameter?

6. When you write a book or make a presentation, you want everything to look as good as it can. One way to do this is to associate a Font object to each piece of text. [http://en.wikipedia.org/wiki/Typeface](http://en.wikipedia.org/wiki/Typeface) gives an interesting background on type and its terminology. Read it.

   A Font class will need to provide the typeface, the point size, and an indication of whether the font is to use bold, or italic, or superscript, or subscript. Implement the Font class.

   To do this properly you may need to use int and boolean datatypes (what are they?) in addition to the String datatype.

7. One of the standard examples of a class is a bank account. Without downloading a BankAccount class from the Internet (yes, there are lots of them there!), design and implement a simplified BankAccount class.

   A bank account should know about its number, owner, balance, and the service charge per transaction. Test all the methods.

   The account should contain methods to make a deposit and to make a withdrawal. It is a simplified bank account since it does not yet contain a record of all the transactions, so
deposits and withdrawals affect the account balance only. In subsequent chapters, we will implement the record of transactions.

8. A playing card is another example of a class. Different cultures have different playing cards. One type of card, a card used in the USA or United Kingdom, and countries in the Commonwealth, knows about its suit (hearts, diamonds, spades, and clubs) and its value within the suit. Images for the suits are available in Unicode. A playing card also knows how to draw itself.

Leaving the drawing for much later, create an AngloAmericanPlayingCard class, along with its associated getters and successors. Test all its methods.

As an alternative, create a playing card class from your culture. In subsequent chapters, we will use this class to play card games.


9. “Die” is the singular form of the word “dice.” Dice are used in many games. Model a die.

What does a die know? It knows how many faces it has.

Don’t jump to the conclusion that a die always has six faces. There are five convex regular polyhedra (the Platonic solids) which can be use as dice. These regular polyhedra have all their faces identical; that’s the “regular” part of their name. The tetrahedron has four triangular faces. The cube has six square faces. The octahedron has eight triangular faces. The dodecahedron has 12 pentagonal faces. The icosahedron has 20 triangular faces.

A die also knows how to determine the value that appears when it is thrown. To implement that we need to know a little about random numbers, something we will see later.

For now, implement a Die class, having only a constructor and a toString method. In subsequent chapters, we will use this class to play a board game.
10. Dominoes have a relationship to dice, as described at http://en.wikipedia.org/wiki/Domino. Design and implement a Domino class.

11. Birders are people who watch birds and record the details of their watching. Every bird species has a common name (American Robin, as an example) and a scientific name (Turdus migratorius, for example).

Design and implement a Bird class. Include all appropriate unit tests. We will explore the world of birding in the exercises of subsequent chapters.

Consider the websites http://www.natureinstruct.org/dendroica/ and http://birding.bc.ca/ if you are interested in Canadian birds.
Chapter 3 – Making decisions

Learning objectives

By the end of this chapter, you will be able to:

- Describe situations when your program may need to make decisions
- Use boolean variables
- Write if statements
- Use the online Java documentation to find out about classes

The if statement

As we noted at the end of the previous chapter, the output of the toString method may appear rather strange when you output someone without a preferred name, displaying () at the end of the name, instead of (Rick), for example.

How do you suppress the printing of the parentheses when there is no preferred name to display? There are several ways to do so, all involving checking conditions.

First, let’s look at what we mean by conditions.

Boolean algebra

A statement like “Today is Tuesday.” is either true or it is false.

A statement like “It is raining.” is either true or it is false. (Yes, there are statements like “This statement is false.” which are neither true nor false. We will not deal with them here.)

When we are dealing with conditions, we are dealing with statements or expressions which are either true or false. True and false are called Boolean values. The Java datatype containing Boolean values is, not surprisingly, boolean.

“Boolean” is a tribute to George Boole, a British logician (1815-64) whose life and work are described at [http://www-history.mcs.st-andrews.ac.uk/Mathematicians/Boole](http://www-history.mcs.st-andrews.ac.uk/Mathematicians/Boole).

There is an algebra associated with these Boolean values, describing the rules under which they may be combined. You have seen one algebra already in your mathematics classes in elementary school, when you studied integers and the addition, subtraction, and multiplication operations. If you add two integers, you get an integer. If you subtract two integers, you get an integer. If you multiply two integers, you get an integer.
Division is a problem, since there are many cases where you divide one integer by another but the result is not an integer.

Boolean algebra is based on three operations (and, or, and not) and the values true and false. The first two operations are binary, the third is unary. That is, you need two Boolean values when you use the and operation, two with the or operation, but only one with the not operation.

Two true conditions combined using and make a true; any other combination of two Boolean conditions combined using and produces a false.

Two true conditions combined using or make a true, as do a true and a false (in either order) combined using or. Two false conditions combined using or produce a false.

Not true is false, and not false is true.

These statements are usually summarized in the following tables. The first cell of the first column is a label. The other cells in the first column (and the other cells in the first row, if any) show the values of the Boolean variables we are combining. The other cells of the table show the result of performing the operation on the value(s) specified in the appropriate row (and column).

Negation

<table>
<thead>
<tr>
<th>Not</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
<td></td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td></td>
</tr>
</tbody>
</table>

Logical conjunction

<table>
<thead>
<tr>
<th>And</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td></td>
</tr>
</tbody>
</table>

Logical disjunction

<table>
<thead>
<tr>
<th>Or</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

What relevance does George Boole have to programming? To have computer programs do anything interesting, they need to be able to examine the data they are processing and make decisions about what to do next.
A program may need to do different processing when a number is positive, negative, or zero.
A program may need to do different processing if a string is empty or if it contains characters. Think of a missing preferred name.
A program may need to do different processing when a specified number of pieces of data have been processed. Think of printing text on a page. You may need to place footer on a page once a specific number of lines of data have been displayed.
A program may need to do different processing depending on a number being even or odd.
A program may need to do some special processing when there is no more data to be processed.
A program may need to do some special processing when a data file is missing.

All of these situations require evaluating conditions, whose results are either true or false, and doing different processing depending on the value of the condition.

Most programming languages contain an if statement which allows for these decisions.

In an if statement, you have one or more conditions, combined using the three Boolean operations. The result of this combination is either true or false. If the result is true, one series of statements will be executed; if false, a different series will be executed.

**Boolean algebra – an example**

For example, this paragraph was originally written in early-April 2006. We just switched to Daylight Savings Time (DST). Most computers did this automatically by the following logic.

If today is Sunday, and this is the first Sunday in April, and the time is 0200 or later, and we have not already switched to Daylight Savings Time, do so now by adding an hour to the time, and remember that we have done so.

This example contains four conditions.

- Today is Sunday, and
- this is the first Sunday in April, and
- the time is 0200 or later, and
- we have not already switched to Daylight Savings Time

They are combined using the **and** operation. The same way that two Boolean values combined using **and** produce a true result only if both are themselves true several Boolean values combined using **and** produce a true result only if all are true.

The situation was actually a little more-complicated. What happened if you did not use your computer on the Sunday when DST started?

The logic should be
(If today is Sunday, and this is the first Sunday in April, and the time is 0200 or later, and we have not already switched to DST) or (today is after the first Sunday in April and we have not switched to DST), do so now by adding an hour to the time, and remember that we have done so.

The parentheses are used to clearly indicate which parts of the logic should be combined.

Note that much of North America changed the rules in 2007, so that Daylight Savings Time now begins three weeks earlier. This is an attempt to save energy on the assumption that it’s light outside longer so people don’t need to turn on their indoor lights. Early indications were that people didn’t go home but drove around, thus saving some electricity but using more gasoline, for a net increase in energy usage.

An if statement is a programmer’s way of writing conditions and executing one of two different scenarios, depending on the result of the conditions being true or false.

**A revised toString method**

Here is a possible revision to toString. It involves only one condition.

```java
public String toString() {
    if (studentPreferredName.length() == 0)
        return "Student number: " + studentNumber + " Student name: " + studentFullName;
    else
        return "Student number: " + studentNumber + " Student name: " + studentFullName + " (" + studentPreferredName + ")";
}
```

The logic used in this method is: Check the number of characters in the preferred name by using the length method from the String class. How did we find that there was such a method?

By looking in the Java documentation for the String class, of course. We're going there in a moment.

When that number is zero, there is no preferred name, so simply return the number and full name. Otherwise (as described in a clause beginning else), return the number, full name, and preferred name, with the latter enclosed in parentheses.

The two adjacent equal signs is shorthand for “Check the values on each side and see whether or not they are identical. Should they be identical, the result of the comparison is true; otherwise it is false.”

The double equal sign is an example of a relational operator, one which checks the relation between its two operands. The relational operators are what we use to create simple conditions.
We combine simple conditions with the Boolean operators to produce more complicated conditions.

When creating an object in BlueJ, you have found that the constructor requires values for all its parameters. Missing data is a problem, and results in an error message. Thus, we had to provide a null string, which we represented as "". How do we know that such a string is empty?

Its length is 0, the number of characters between the quotation marks.

**Using the Java documentation**

The Java language is well-documented, in some textbooks and online. (In the main BlueJ window, select Help from the main menu, then Java Class Libraries.)

When you look at details on the String class, you are looking at the output of javadoc (when online), or at an edited version of the output of javadoc (when using the dead-tree form of documentation.) I prefer the online version, as it is always up-to-date.

Assuming you are using the online version of the Java documentation, your browser opens, with three frames visible. The frame in the top left corner contains a list of all the Java libraries. You could start there (if you know the name of the library which contains your target, java.lang since we are looking for String) by scrolling down the list until you see java.lang and then clicking it. The bottom frame would then show all the topics available from the java.lang library. Scroll down that list until you find String.

Or you could use the frame in the bottom left corner, and scroll down until you see String. This is probably the best solution until you know which classes are in which libraries. Warning: there are many classes listed in this frame.

Perhaps your browser allows a shortcut. Left-click in a frame and then select Edit, Find to search within that frame.

Once you have found the String class, click it. The large frame on the right of the screen now displays the javadoc output for the String class.

Look in the section of the documentation entitled Method Summary. Scroll through it to find descriptions of all the methods this class supports. In particular, find the length method. You see a short explanation of its use; we use this method to determine how many characters are in the string, by looking at the value it returns, an int.

Further details on the method are in the Method Details. (Why is that section of the documentation not in alphabetical order? I have no idea.)

A value of 0 tells us we are dealing with an empty string, a value greater than 0 tells us there are characters in the string. When a string contains many characters but only blanks, it will have a
non-zero length. Thus, " " is different from "". The first contains two blanks, the second contains none.

To determine the length of a string in Java, we use the name of the variable followed by a period and the name of the method. Thus, studentPreferredName.length() is the number of characters in the variable named studentPreferredName.

This is the same way we used the name of a variable in our tests followed by a period followed by the name of a getter or a setter.

To check if the length of a string in Java is zero, we write

```java
if (studentPreferredName.length() == 0)
```

Recall that the double equal sign is shorthand for “check the values on each side and see if they are identical.” Since there are only two possibilities (the string is empty, and thus its length is zero, or it is not empty and its length is greater than zero), the if statement allows us to distinguish between the two cases.

When the condition is true (meaning there is no preferred name), simply return the value "Student number: " + studentNumber + "Student name: " + studentFullName.

When the condition is false (meaning there is a preferred name), return the value "Student number: " + studentNumber + " Student name: " + studentFullName + " (" + studentPreferredName + ")".

The statement or statements describing the action(s) to take when the result is true begin(s) immediately after we test the condition. The statement or statements describing the action(s) to take when the result is false begin(s) after the word else.

```java
public String toString()
{
    if (studentPreferredName.length() == 0)
        return "Student number: " + studentNumber + "Student name: " + studentFullName;
    else
        return "Student number: " + studentNumber + " Student name: " + studentFullName + " (" + studentPreferredName + ")";
}
```

If you wish to combine more than one simple condition in a larger condition, you may use the symbols && to represent and, and you may use || to represent or, or you may use one if statement within another. Note that not is represented by the exclamation point. We will see these possibilities later on.

The pipe symbols used to make || are typically on your backslash key.
Programming style – if statements

if statements are usually much more complicated than the one we have seen. You may wish to do several things when the condition is true instead of just one. It may take several statements to describe the action. In that case, enclose the statements in braces. Some authors suggest you should always use braces when working with if statements.

Therefore a common style is to enclose the statement(s) to be executed for a true condition and the statements(s) to be executed for a false condition in braces, even if there is only one statement.

If there is only one, the braces are not required, but having them there may save some problems later. If you add another statement, without adding the braces, your program may not compile and, if it does compile, it may not produce the correct results.

This alternative toString method illustrates this use of braces.

```java
public String toString() {
    String result;
    if (studentPreferredName.length() == 0) {
        result = "Student number: " + studentNumber;
        result = result + "Student name: ");
        return result;
    } else {
        result = "Student number: " + studentNumber;
        result = result + "Student name: ");
        result = result + studentFullName + "(");
        result = result + studentPreferredName + ")";
        return result;
    }
}
```

Braces are necessary in this method since several statements are processed when a decision is made of which path to follow.

Checkstyle

Now is a good time to introduce a BlueJ extension called Checkstyle, particularly since I have mentioned it earlier.

BlueJ extensions are features which have been added to BlueJ, but not by the BlueJ development team. You can find a list of the extensions which are currently available at [http://www.bluej.org/extensions/extensions.html](http://www.bluej.org/extensions/extensions.html).

To install Checkstyle, visit the extensions page and scroll down until you find Checkstyle.
Follow the Download link; the text early on the page notes that Checkstyle was not written for BlueJ, but has a wrapper around it that allows it to operate within the BlueJ environment.

Click the latest .jar file (as I review this in May 2012 the link is labelled Checkstyle-extension-5.4-0.jar) and save it to the lib/extensions folder within your BlueJ installation. Then open BlueJ and select Tools. The Checkstyle extension will appear on the menu. Click it.

A jar file is a Java archive, a zipped collection containing all the files necessary to use a collection of Java classes. Thus, a jar file may contain .java files, .class files, and the .html files which document the classes. You can see this by opening a .jar file in an unzipping program, but don’t unzip it.

To create a jar file of your project, select Project, Create jar file. If you are submitting the file for marking, check include source and check include project files.

Checkstyle displays a window appears containing two panes. On the left is a list of the classes in your project. As you click each, the right pane lists the style violations found in that class.

There are many different style files available, but I’ll leave it to you to explore them.


**Simpler tests**

Sometimes you have processing in which you do nothing when the condition is true but do something when the condition is false, or you do something when the condition is true, but nothing when it is false. Consider the following alternative version of toString.

To create this version, we first notice that some of the processing in the previous version is the same whether there is a preferred name or not. So we do that processing, and then we use the relational operator greater than (as a change from ==) to see if there is a preferred name and, if so, do a little more processing.

```java
public String toString() {
    // done whether or not there is a preferred name
    String result;
    result = "Student number: " + studentNumber;
    result = result + "Student name: ";
    result = result + studentFullName

    if (studentPreferredName.length() > 0) {
        // done only if there is a preferred name
        result += "(" + studentPreferredName + ")";
    }
    return result;
}
```
The first relational operator we saw was ==, for testing equality. Now we have greater than, or >. Not surprisingly, there are other relational operators. These include >= (greater than or equal), < (less than), <= (less than or equal), and != (not equal).

**More complicated tests**

The chapter began with an example that combined four conditions and mentioned that conditions could be combined with && and ||. But we haven’t seen how yet. Let’s remedy that deficiency right now.

At Okanagan College, everyone has a number, whether they are students or professors or any other employee of the college. Anyone can have any number, it appears, except that students whose permanent residence is outside Canada are given student numbers which begin either with an eight or a nine.

Pause for a moment to create a method, isInternational, which will examine the student number and decide whether the student is an international student. Use the charAt method in the String class to extract the first digit of the student number as a char, a single character.

Does your solution look like this?

```java
public boolean isInternational() {
    char firstDigit = studentNumber.charAt(0);
    if (firstDigit == '8')
        return true;
    if (firstDigit == '9')
        return true;
    return false;
}
```

Or does it look like this?

```java
public boolean isInternational() {
    boolean result = false;
    char firstDigit = studentNumber.charAt(0);
    if (firstDigit == '8')
        result = true;
    if (firstDigit == '9')
        result = true;
    return result;
}
```

Or does it look like this?

```java
public boolean isInternational() {
    return studentNumber.charAt(0) == '8' || studentNumber.charAt(0) == '9';
}
```
Or does it look like this?

```java
public boolean isInternational() {
    char ch = studentNumber.charAt(0);
    return ch == '8' || ch == '9';
}
```

All the solutions are correct. The third and fourth solutions use the `||` operator to test if the first character is an eight or is a nine. Which solution is best? That decision is up to you and your teacher.

How many unit tests did you need to test this method?

You should have used at least three: one for numbers beginning with a nine, one for numbers beginning with an eight, and one for numbers beginning with any other digit. Really there should be tests for any first digits the college uses. Here are two of my tests.

```java
@Test
public void testisInternational1() {
    assertFalse(s1.isInternational());
}

@Test
public void testisInternational2() {
    assertTrue(s2.isInternational());
}
```

You can see that s1 should not be an international student, while s2 should be.

For the next examples, you should create new project called Testing and a class within it called Example, since the examples have nothing to do with students.

An integer n is defined to be even when the remainder when you divide it by two is zero. Add the following method to Example.

```java
public boolean isEven(final int n) {
    return (n % 2) == 0;
}
```

The modulus operator (%) gives the remainder when you divide the first value (in this case n) by the second (in this case two). The method calculates that remainder and compares it to zero. If the remainder is zero, n must be even.

To test this method, you could create unit tests. I would create two, one for a positive even number and one for a positive odd number. Being a suspicious person, I would create three additional tests; one for zero, one for a negative even number, and one for a negative odd number.
Or, you could compile Example, then right click it and select the constructor. This places an Example object in the object tray. Right-click it there and select the isEven method.

Now write an isOdd method. Which did you select?

```java
public boolean isOdd(int n) {
    return (n % 2) == 1;
}
```

Or

```java
public boolean isOdd(int n) {
    return (n % 2) != 0;
}
```

or

```java
public boolean isOdd(int n) {
    return !isEven(n);
}
```

The first two are modelled on isEven, but the third is based on the mathematical idea that an integer is either even or odd; there are no other alternatives.

There is a mathematical concept called “evenly even.” There are a couple of different definitions for “evenly even.” Let’s use the one that says an integer is evenly even if it is evenly divisible by four. You should be able to write isEvenlyEven quite quickly. Do it!

Leap years in the western calendar are an interesting challenge. The simplified rule is that a year is a leap year when it is evenly divisible by four but when it is also divisible by 100, it will only be a leap year when it is also divisible by 400. Thus 1996, 2000, and 2004 were all leap years. But 2100, 2200, and 2300 will not be leap years. Create an isLeapYear method and then come back and look at mine.

Welcome back.

```java
public boolean isLeapYear(int year) {
    boolean result;
    if (year % 4 == 0){
        // divisible by four so might be a leap year
        result = true;
        // check the centuries
        if ((year % 100 == 0) && (year % 400 != 0))
            // oops, a century but not divisible by 400 so not a leap year
            result = false;
    } else
        // not divisible by four so not a leap year
        result = false;
    return result;
}
```
In this method you see that we can nest one if statement within another. That is, we evaluate a second condition only when a previous condition is true (in this example). Within a single if statement we can write compound conditions, ones which use && and ||. This example uses the && operation. Note that we also use the != relational operator.

Of course, there is another way to test if a year is a leap year. Look in the GregorianCalendar class. To use this class, you'll need to import it into your class. But that is a subject for a later chapter.

You can close the Example project; we are finished with it for now, but you can use it for exploring Java if you'd like.

**Summary**

This chapter has introduced one of the fundamental features of programming: using the data you are processing to control the flow through your program. It has also introduced Boolean algebra, the Boolean operations and, or, and not, and various relational operators.

It has also shown you how to use the Java documentation to answer questions about classes which are part of the Java libraries.
Exercises

1. A simple object to model is a coin. (The image is of coins, not a bunch of grapes!) A coin has two sides, typically called heads and tails.

Assume that a Coin class implements the idea of heads and tails by using a random number between 0 and 1. For a fair coin, if the number is less than 0.5 consider the coin to be heads, otherwise consider it to be tails. java.lang.Math contains a random method which you could use. (Check the documentation to see how. Note that you will use Math.random. Until now we have used the name of an object, followed by a period and a method name. Here we use the name of the class, followed by the period and the method name. This is because random is a static method; that is, it is a class method, not an instance method.)

Create a method, flip, which generates a random number which two other methods, isHeads and isTails will examine to decide if the coin is heads or tails. The names of these two methods are based on practice which says that methods which return a boolean value should have names beginning with the word is. Such methods are also considered to be getters.

How would you implement your class to allow for biased coins, ones which come up with either heads or tails more than expected?

Recently I read that when you flip a coin it ends up the same face up as it starts with 51% of the time. This is mentioned at http://en.wikipedia.org/wiki/Coin_flipping.

2. Stores have customers. Each customer has a number and a name. Each customer also has a discount rate. That is, when the customer makes a purchase, he/she will be granted a discount. This discount may be zero, or it may be a positive percentage (or a positive amount) based on his/her previous purchases. Of course the discount might also be based on being a neighbour or friend of the owner, being a student, or being an employee. How would you model such a discount scheme?

3. In Canada, each province has its own provincial tax rate, which may be zero. Anyone who purchases something may need to pay the provincial sales tax or PST. Some goods are exempt from taxes, but we will ignore that detail for now.

There is also a federal Goods and Services Tax (GST) whose payment is required in all provinces except Nova Scotia, New Brunswick, Prince Edward Island, and Newfoundland and Labrador. In those four provinces, there is the Harmonized Sales Tax. Instead of separate PST and GST, purchases there are subject to an HST. Starting in July 2010,
Ontario and British Columbia adopted an HST as well.

Create a Purchase class. In it, write a method which has parameters amount and postalCode, and returns the tax (PST plus GST, or HST) necessary for a purchase. Note that http://www.canadapost.ca/personal/tools/pg/manual/PGaddress-e.asp#1380608 contains a map showing all the provinces of Canada plus the first character of the postal code for that province. The larger provinces may use several postal codes. To extract the first character from the postal code, use the charAt method from the String class and provide the value zero as a parameter, as we did when determining if a student number represents an international student. Alternatively, you may wish to use the startsWith method from the String class.

4. In an exercise in the previous chapter, we mentioned modelling a bank account. What modifications would you need to make to your class if the bank changed the rules to allow some number of free transactions before it began to charge for transactions?

What modifications would you need to make to your bank account class if the bank changed the rules to allow free transactions as long as your balance stayed over a specified amount?

5. In an exercise in the previous chapter, you modelled a die. Use the random method in java.lang.Math to create a roll method.

See exercise 1 of this chapter for further information on the random method.

6. In an exercise in the previous chapter, you designed a Bird class. In many parts of the world, birders often use abbreviations for the bird name. Common systems use four characters (http://elibrary.unm.edu/sora/NABB/v003n01/p0016-p0025.pdf) or six (http://infohost.nmt.edu/~shipman/z/nom/6home.html).

Different people prefer different abbreviation systems, but one person will always use the same system. Add the abbreviation as an instance variable of the Bird class.

Create a method to determine the abbreviation.

In the constructor, the only parameters will be the common name and the scientific name. Use a separate method to determine the abbreviation, given the common name. Note that common names change over time, so the setter for the common name will use this separate method to determine the abbreviation. Since the method to determine the abbreviation is called only by methods within the Bird class, that method will be private, not public.

We will explore the world of birding in subsequent chapters.

7. The Canadian postal code is a six-character string. The first, third, and fifth characters (counting from the left) must be letters. The second, fourth, and sixth characters (counting from the left) must be digits.
Create a method, isValidPostalCode, which accepts a String as a parameter and decides if it represents a valid postal code, returning the appropriate boolean value.

You will need to use the charAt method from the String class, and the Character.isLetter and Character.isDigit methods from the Character class. To check that the character in position loc of a string ess is a letter, use Character.isLetter(ess.charAt(loc)).

Recall that Java numbers its characters starting at zero while English refers to that character as the first character.

8. Canadians have a national identification number called the Social Insurance Number or SIN. The SIN is a nine-digit number but it’s not just any nine-digit number. Not only does the first digit indicate the region of Canada in which the number was issued, but the SIN can be validated through the use of the Luhn Algorithm, described at http://en.wikipedia.org/wiki/Luhn_algorithm.

The SIN itself is described at http://en.wikipedia.org/wiki/Social_Insurance_Number.

Create a method, isValidSIN, which accepts a String as a parameter and decides if it represents a valid SIN, returning the appropriate boolean value.

Assume that the input contains only digits. There are two possible approaches to solving this problem. In the first, you extract single-character substrings from the SIN using the substring method in the String class, and then use the Integer.valueOf method to convert the single-character substring to an int.

In the second approach, you convert the nine-character string into an int using Integer.valueOf and then use the modulus and division operators to extract individual digits. Either technique will work.
Chapter 4 – Inheritance

Learning objectives

By the end of this chapter, you will be able to:
- Define abstract class and inheritance
- Use abstract classes to implement inheritance

Abstract classes

Before we go any further with the Student class, we should think back to a comment made earlier. We noticed that the Professor and Student classes looked very similar.

After you gain more experience with object-oriented programming (and object-oriented analysis and design, the steps you should do before you start writing object-oriented code) your ears will perk up at a statement that says “classes are similar” and intuition will kick in whenever you see such commonality.

Commonality usually implies that there is a better way to deal with the classes than what you have first decided. In this case, we divided the people in the classroom into two groups, the students and the professors. But we are all people!

Why not create a People class? That class can contain all the common fields in Professor and Student.

What common fields are there? Well, right now there are none since the attribute names are different, but think a little deeper.

Professor has a professorNumber. Student has a studentNumber. Are they different in structure? Not in this example; both are nine-digit numbers, entered as Strings. Since number is not a very descriptive variable name, I will use identifier instead, for both.

Professor has professorFirstName. Student has studentFirstName. Are they really different? No, so I will use firstName instead, for both.

Professor and Student have several common fields, common on the basis of function if not name.

So how can we use this information? We can create a new class, called Person and derive the Professor and Student classes from it. (The name of a class is usually a singular noun. Thus we
used Person rather than the initial suggestion of People.) This is shown in the following class diagram. BlueJ will draw something similar, but omitting the instance variables.

```
Person
    identifier : String
    firstName : String
    middleName : String
    lastName : String
    fullName : String
    preferredName : String;

Student

Professor

Note the arrow from the Student class to the Person class. This is read as “a Student is a Person” or “Student is derived from Person.” Similarly, there is an arrow from Professor to Person. One diagramming style, which I have used here, is that the two arrows share a common path whenever it is reasonable to do so. BlueJ does not do this.

Person is referred to as a base class or a parent class or a superclass for both Student and Professor. Student and Professor are both derived from Person or are child classes of Person or are subclasses of Person.

Note that I will use the names parent class and base class interchangeably. In the same way I will use derived class and child class interchangeably.

Derived classes have features (instance variables and methods) in common with the base class, but the derived classes may have features (additional instance variables, additional methods, replacements for existing methods, or enhancements of existing methods) which make them different. We’ll see this in a few moments.

The Person class

Create a Person class within the College project. Enter the appropriate instance variables and constructor. You can copy the code from your Student or Professor class, remembering to change the variable names (studentFirstName, for example, becomes firstName). This is a good way to learn the Replace function in BlueJ. Note that the parameters and body of the constructor for
Person are identical to the parameters and constructors for Student and Professor, with the exception of the changed variable names.

But wait a moment! Do we wish to be able to create instances of the Person class which are neither students nor professors?

If you respond that “No, we don’t wish this.” (That’s the way I responded.), we must declare the class as abstract, a word which appears in the class header and signals to Java that we won’t be instantiating (creating instances of) the class. If we try to instantiate an abstract class we will get an error.

If you respond that “Yes, there are other types of people we wish to model besides students and professors. Perhaps there are the administrative deans. Perhaps we wish to model alumni and the recipients of honorary degrees.” then you do not wish to declare Person as an abstract class.

I can see the arguments behind both answers but, for the model we are creating (and it’s my model!), we won’t have anyone other than a professor or a student. So I will declare Person as an abstract class.

```
public abstract class Person
```

Compile the Person class and see what happens when you attempt to instantiate it. That is, try and use its constructor.

Right-click the class and select the constructor from the menu.

But the constructor is not exposed! That is, it does not appear on the menu. Thus we cannot create a Person object. That is because the class is abstract. We cannot create instances of an abstract class directly.

Open Student and Professor, changing the header of each to include the words extends Person.

That is, Student begins with the statement

```
public class Student extends Person
```

and Professor begins with the statement

```
public class Professor extends Person
```
Notice how the BlueJ class diagram shows that Person is an abstract class and adds arrows to indicate that Student and Professor are derived from it.

How do you create a Student object? That is, what should the constructor contain, now that all the instance variables are in the parent class?

A Student object needs a simplified constructor; it can use the constructor of its base class, simply by calling super as the first step in the Student constructor. Java uses the word super to refer to the parent class. Without a method after the word super, you are executing the constructor in the base class.

```java
Student(final String identifier, final String firstName, final String middleName, final String lastName, final String fullName, final String preferredName) {
    super(identifier, firstName, middleName, lastName, fullName, preferredName);
}
```

Attempt to compile Student. It will not compile, since the toString method in the derived class is trying to use private instance variables of the base class. private instance variables (and methods, for that matter) may be accessed directly only within the class in which they are declared. They may not be accessed from derived classes.
protected instance variables (and methods) may be accessed directly within the class in which they are declared as well as in any derived classes. Change the visibility of the instance variables of Person to protected.

**toString in an abstract class**

We have placed the common instance variables in the abstract class. We have placed common methods (getters and setters) in the abstract class. We have left the toString method in the derived class. Is this correct?

Both Student and Professor contain that method, and, as a general design rule, we should place methods as high up the inheritance tree (as close to the base class) as we can. Examine the two methods you created and you’ll see that they are almost identical. The only difference is that one toString method places the word Student at the beginning of its result, the other places the word Professor.

Thus the methods are not identical, so you could argue they should remain where they are. But you could also argue that the following solution works as well or better.

Create a toString method in the Person class. Its code is below.

```java
public String toString() {
    String result = new String();
    result = "number: " + identifier +
        "  First name: " + firstName +
        "  Middle name: " + middleName +
        "  Last name: " + lastName +
        "  Full name: " + fullName;
    if (preferredName.length() != 0)
        result = result + " (" + preferredName + ")";
    return result;
}
```

You have seen similar methods in a previous chapter. To calculate and save a string, I need a variable of type String, which I have named result. (A variable is a piece of memory with which I associate a datatype and name. In this case, the area of memory will contain a reference to a String, and when I need the contents of that area of memory, I’ll use the name result.)

Whether or not there is a preferred name, I need to concatenate the identifier and the first, last, middle, and full names. So I concatenate them, and save the result. The saving is done in the statement beginning result =. The way to interpret this statement is to say “Calculate the value to the right of the equals sign. Since it is a String, use result to remember where the value is stored.” This is exactly what we have been doing in our constructors and previous toString methods.

Now I need to check whether there is a preferred name. When there is no preferred name, there is nothing to be done, and that could cause some strange code.
if (preferredName.length() == 0)  
     // do nothing
else
     result = result + " (" + preferredName + ")";

There are no executable statements between the if and the else. Most people would wonder what is missing so we place a comment there to indicate we have not forgotten something. But the resulting code looks strange to my eyes.

So we do not test for equality, we test for inequality. As we have seen earlier the exclamation point is used to indicate not. Thus == means equals, and != means not equal. Note that we could have used the symbol > (or greater than) instead of != since the length cannot be negative.

if (preferredName.length() > 0)  
     result = result + " (" + preferredName + ")";

Tests for inequality or greater than are both acceptable and are preferable, to my eyes, to testing for equality but having an empty section of code.

In the Student class, include the following method.

public String toString()
{
     return "Student " + super.toString();
}

In the Professor class, include the following method.

public String toString()
{
     return "Professor " + super.toString();
}

As noted earlier, the word super refers to the base or parent class. super.toString() asks the parent to represent itself as a String, and then, as appropriate, we prepend a word to that String (add a word to the beginning of the string the parent calculates for us.)

Note that we are **overriding** the toString method in the parent class by creating a toString method in the derived class.

**Hint:** If you are unsure which classes you have changed and thus should be compiled, look at the class diagram. You can tell which classes need to be compiled since they appear striped in the diagram. Simply click the Compile button beside the class diagram and all will be compiled in the appropriate order. That is, if you have changed both Student and Person, the parent class, Person, will be compiled first.
The Person class, continued

Do you need to create getters and setters in the Student class? No. Those methods, and the setters should be in the Person class. If we use the expression s.getFirstName(), where s is an instance of Student, the effect is that the Student object essentially says “I don’t recognize that message. I’ll ask my parent to tell me how to respond.” When the parent responds, the Student can respond.

Set up your Student unit tests to ensure that they still work. You have changed the names of the getters and setters, haven’t you?

The Student class – a derived class

My implementation of the Student class is shown below.

```java
public class Student extends Person {
    Student(final String identifier, final String firstName,
            final String middleName, final String lastName,
            final String fullName, final String preferredName) {
        super(identifier, firstName, middleName, lastName, fullName, preferredName);
    }

    public String toString() {
        return "Student " + super.toString();
    }

    public boolean isInternational() {
        char firstDigit = identifier.charAt(0);
        if (firstDigit == '8')
            return true;
        if (firstDigit == '9')
            return true;
        return false;
    }
```
Modify the unit tests for Student to take into account the different instance variable names we are using.

The Professor class

Now that we have the Student class created and tested, we can create and test the Professor class. It is currently similar to the Student class, but does not need the isInternational method. Like the Student class, it is a derived class.

However, the Professor class has an additional instance variable, the number of the office in which the professor is available for consultation. In addition to the instance variable office, a Professor object has a getter and a setter. Make it so.

The unit tests here are simple. We know that the getters and setters for parts of the name work, since we have tested them as part of the Student class. The methods we need to test are toString and the office getter and setter.

Garbage in, garbage out

What happens to your computer programs when you provide input that breaks the rules?

For example, the identifier we are using for both students and professors is a nine-digit number, which we represent as a String. What happens if we provide an eight-digit number? A ten-digit number? Something which contains letters or punctuation?

You may provide bad data, but your program should reject it. The expression “garbage in, garbage out” is well-explained in the Wikipedia article at http://en.wikipedia.org/wiki/Garbage_In,_Garbage_Out.

The way to prevent garbage getting into our program is by diligent editing of the data. Part of editing is providing notification that an error has been detected. In Java this is done through throwing exceptions, the topic of the next chapter.

In Summary

In this chapter we have had an introduction to inheritance. But it is just an introduction. In particular, we have seen how to create classes which inherit instance variables and methods from their parents. We have seen how to override methods. We have seen how to add instance variables to derived classes.
We will see inheritance again in later chapters.

We have also gained more experience with unit testing.

Now that we have created well-designed Student and Professor classes, albeit simplified classes, we can explore them in more detail, identifying missing instance variables and behaviours. This exploration begins in chapter six. First, though, we must explore exceptions.

Note the phrase “simplified classes” in the previous paragraph. This simplification is the essence of abstraction and model making. A model is a representation of something “real.” Of course, our class and its instances do not represent actual, living, students. It is just a model and omits many aspects of real students. It represents a simplified student, an abstraction.

When we speak of abstraction, we are referring to the process of eliminating non-essential aspects of objects which model real things, students in this case.
Exercises

1. In a previous chapter, I discussed the use of a bank account as a common class for modellers. Part of its attraction is that there are so many types of bank accounts. Savings accounts don’t allow cheques. High interest accounts allow only a few withdrawals in a month. Extend your model of the bank account to accommodate some of these variations. An interesting collection of bank accounts are described at http://cse.stanford.edu/class/cs108/982handouts/14%20Inheritance%20Examples.pdf. I don’t think a real bank would have account types called Nickel ‘n Dime, and the Gambler. The programming in this example is in C++ rather than Java, but you may be able to understand much of it.

A second section of the page referred to earlier contains an interesting perspective on the role of Instructors at a teaching institution.

2. In a previous chapter, I discussed birding. Birds and all other living things provide interesting examples of inheritance.

A gull is a type of bird, and there are many types of Gulls, including Herring, Mew, and California.

An albatross is a type of bird, and there are many types of albatross, including the Black-browed and the Black-footed.

Sparrows come in many varieties, including Harris’, Song, and House.

Warblers too come in many varieties, including Yellow, Swainson’s, and Kirtland’s.

Model these in the simplest way possible.

Note that the biological classification into kingdoms, orders, families, etc. is a fertile source of modelling exercises.

3. At Okanagan College there are not just students and professors. There are also administrators, who provide management and direction, and support staff, the people without whom the institution would not run. Objects of the Administrator class need to know their salary. Support staff are unionized. For historical reasons, there are three separate locals of the union on campus. Objects of the SupportStaff class need to know the local to which they belong.

Implement Administrator and SupportStaff classes using inheritance from Person.

4. Sun (and now Oracle) provides a Java tutorial on its website. One of its inheritance examples is the bicycle and the different types of bicycle. http://download.oracle.com/javase/tutorial/java/concepts/inheritance.html. Model the
inheritance described.

How would you change the model to accommodate unicycles? Is a unicycle a bicycle?

5. Okanagan College is a four-year college so has no graduate students, people who have already completed one degree and are studying for a higher degree. A GraduateStudent is a Student. How would you model a graduate student?

6. In a previous chapter, I discussed modelling a Room at a college. There are actually several types of rooms, including lecture rooms and laboratories. Use inheritance to model these three room types.

7. In a previous chapter, I discussed modelling Furniture at a college. Assuming that the different types of furniture include tables, desks, chairs, and filing cabinets, use inheritance to model these different types of furniture.
Chapter 5 – Exceptions

Learning objectives

By the end of this chapter you will be able to:

- Define exception
- Describe a few exceptions from Java libraries
- Define catch and try
- Identify when exceptions are thrown and describe how to catch them
- Create, throw, and catch your own exceptions
- Use exceptions to create programs which gracefully handle erroneous data and mistakes in computation

Definition

Exceptions are an important topic in object-oriented programming. We will see them throughout most of the rest of this book. Before we see them, we should first answer the question “What is an exception?”

There are many sources that will answer the question, but one of my favourites is Oracle's Java tutorial, available at http://download.oracle.com/javase/tutorial/. One of the trails through the tutorial takes you through essential classes and features. Exceptions are essential, and are described at http://download.oracle.com/javase/tutorial/essential/exceptions/definition.html

This section of the tutorial contains a number of quotations from the tutorial, as viewed on 2006-05-19. For example, we have this introduction:

“What Is an Exception?

“The term exception is shorthand for the phrase "exceptional event."

“Definition: An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

“When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when...
the error occurred. Creating an exception object and handing it to the runtime system is called *throwing an exception*.

“After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible “somethings” to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the *call stack*.

“The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an *exception handler*. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

“The exception handler chosen is said to *catch the exception*. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, the runtime system (and, consequently, the program) terminates.”

**Examples**

Where might we want to throw an exception? The best example is the one mentioned at the end of the previous chapter: what happens if the identifier provided is not a String which contains nine digits?

**Runtime and nonruntime exceptions**

As exceptions in Java are very important, there is a class named Exception and a number of subclasses derived from it. The documentation on Exception and its subclasses (derived classes) lists its subclasses.

*AclNotFoundException, ActivationException, AlreadyBoundException, ApplicationException, AWTException, BackingStoreException, BadAttributeValueExpException, BadBinaryOpValueExpException, BadLocationException, BadStringOperationException, BrokenBarrierException, CertificateException, CloneNotSupportedException, DataFormatException, DatatypeConfigurationException, DestroyFailedException, ExecutionException, ExpandVetoException, FontFormatException, GeneralSecurityException, GSSException, IllegalAccessException, IllegalClassFormatException, InstantiationException, InterruptedException, InvocationTargetException, IOException, JAXBException,
KeySelectorException, LastOwnerException, LineUnavailableException, MarshallException, MidiUnavailableException, MimeParseException, MimeParseException, NamingException, NoninvertibleTransformException, NoSuchFieldException, NoSuchMethodException, NotBoundException, NotOwnerException, ParserConfigurationException, PrinterException, PrintException, PrivilegedActionException, PropertyVetoException, RefreshFailedException, RemarshalException, RuntimeException, SAXException, ScriptException, ServerNotActiveException, SOAPException, SQLException, TimeoutException, TooManyListenersException, TransformerException, TransformException, UnmodifiableClassException, UnsupportedAudioFileException, UnsupportedCallbackException, UnsupportedFlavorException, UnsupportedLookAndFeelException, URIReferenceException, URISyntaxException, UserException, XAException, XMLParseException, XMLSignatureException, XMLStreamException, XPathException

That documentation also contains the sentence “The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch.”

Reasonable, indeed.

By the way, you may wish to look at the class Throwable. “The Throwable class is the superclass (or parent class) of all errors and exceptions in the Java language.” Many of the Exception methods we use are actually derived from Throwable.

Exceptions come in two flavours: runtime and nonruntime. The Java tutorial states

“Runtime exceptions occur within the Java runtime system: arithmetic exceptions, such as dividing by zero; pointer exceptions, such as trying to access an object’s members through a null reference; and indexing exceptions, such as trying to access an array element with an index that is too large or too small. A method does not have to catch or specify runtime exceptions, although it may.

“Nonruntime exceptions are exceptions that occur in code outside of the Java runtime system. For example, exceptions that occur during I/O are nonruntime exceptions. The compiler ensures that nonruntime exceptions are caught or specified; thus, they are also called checked exceptions.”

The exceptions reporting inappropriate identifiers will be runtime exceptions, once we create them ourselves.

Creating and throwing exceptions

Where should we validate an identifier? We could do it within the Professor and Student constructors, but the reason for our use of inheritance has been to push all common features into the highest class possible. Following that logic, we should validate the identifier within the Professor class.
Since there is one set of rules for validating an identifier, the method to do the validation should not be attached to individual objects; it should be attached to the Person class itself.

The method to do the validation should throw an exception, if the validation method determines there is a problem.

To use an exception, we have two choices.

- When there is an appropriate type of exception already available, we create an object of that type.
- When there is not an appropriate type of exception, we create our own exception.

### DataFormatException

It happens that there is an appropriate type of exception available, DataFormatException, an existing type of exception, from the java.util.zip package.

Thus we can use the following statements in the validation method.

```java
if (identifier.length() < 9)
    throw new DataFormatException("Identifier should contain 9 digits. " + identifier + " only contains " + identifier.length() + ".");
if (identifier.length() > 9)
    throw new DataFormatException("Identifier should contain 9 digits. " + identifier + " contains " + identifier.length() + ".");
```

along with the appropriate import statement.

```java
import java.util.zip.DataFormatException;
```

What are the other statements that make up the validation method?

Not only could we use a DataFormatException, we could also use a NumberFormatException.

The Java documentation indicates NumberFormatException is “Thrown to indicate that the application has attempted to convert a string to one of the numeric types, but that the string does not have the appropriate format.” That is close to what we are trying to do.

You are free to use whichever pre-defined exception you wish. But there is a better solution.

### A better solution

This may appear to be the correct approach, but it is probably a better decision to have separate exceptions for the too short, too long, and not numeric cases. We therefore need to create three exception classes, ShortIdentifierException, LongIdentifierException, and NonnumericIdentifierException.
### ShortIdentifierException and LongIdentifierException

The code to create an exception is very simple.

```java
/**
 * Exception for short identifiers.
 */
public class ShortIdentifierException extends Exception {
    /**
     * Constructor for objects of class ShortIdentifierException.
     * @param msg The message associated with the exception
     */
    public ShortIdentifierException(final String msg) {
        super(msg);
    }
}
```

Where do we import `Exception`? We don't since it is part of the `java.lang` library and that library is imported for us, automatically.

Create a similar class for `LongIdentifierException`.

### Validation using exceptions

Normally you don’t generate an exception and do nothing with it; you either handle it or you throw it to some other section of your program which acknowledges the problem and deals with it. We will create a method to validate the identifier we propose using. If there is a problem, the method will throw the appropriate exception or exceptions.

Thus the validation method begins as follows.

```java
/**
 * Method to validate an identifier.
 * @param identifier The String which you want to use as an identifier
 * for a member of the college community.
 * @return the valid form of the identifier. Normally this is the value
 * provided as a parameter but it may have the hyphens and dashes
 * removed.
 * @throws ShortIdentifierException if the identifier is too short.
 * @throws LongIdentifierException if the identifier is too long.
 */
public static String validateIdentifier(final String identifier)
    throws ShortIdentifierException, LongIdentifierException {
```
Note the use of the word static. This indicates that the method is associated with the class itself, not with instances of the class. When we call this method, we will use Person.validateIdentifier instead of the name of an object followed by the period and the name validateIdentifier.

The body of validateIdentifier

The word final in the list of parameters is our promise that we will not change the parameter within the method. But the documentation implies that we may need to modify the parameter. To resolve this problem, we will use the following as the first statement of the method:

```java
String result = new String(identifier);
```

This makes a copy of the parameter; that is what we will change.

The last two lines of the method will be

```java
return result;
}
```

In between those statements, the method needs to first examine the identifier to see if it is too short. If so, it should throw an exception.

```java
if (result.length() < 9)
    throw new ShortIdentifierException("Identifier should contain 9 digits. " + identifier + " only contains " + identifier.length() + ".");
```

When a method throws an exception the normal flow through the method stops. If the current method cannot handle the exception, the method is terminated and the calling method is passed the exception. Since we are using the exception to send a message outside this method, the calling method must do something with the exception or must pass it back up the call stack.

If the identifier is not too short, the method needs to examine the identifier to see if perhaps it is too long. If so, it should throw an exception.

```java
if (identifier.length() > 9)
    throw new LongIdentifierException("Identifier should contain 9 digits. " + identifier + " contains " + identifier.length() + ".");
```

A cultural problem comes up here. People usually cannot remember nine digit numbers but they can remember three three-digit numbers. Thus many people will recite their identifier as 300 106 765 rather than 300106765. The spaces in the number represent pauses. When people write their number, they may write it as 300 106 765 or they may write it as 300-106-765. Hence a user may provide an identifier which contains punctuation. Once the punctuation is removed, the identifier is acceptable. The statements we have provided so far would throw an exception for 300-106-765 since it contains 11 characters.
But it would be polite to quietly remove the spaces and/or punctuation and see if what remains is an acceptable identifier.

**while loops**

We can do this! After the statement

```java
String result = new String(identifier);
```

insert the following statements.

```java
// remove hyphens
char target = '-';
int i = result.indexOf(target);
while (i != -1) {
    result = result.substring(0, i) + result.substring(i + 1);
    i = result.indexOf(target);
}

// remove spaces
target = ' ';
i = result.indexOf(target);
while (i != -1) {
    result = result.substring(0, i) + result.substring(i + 1);
    i = result.indexOf(target);
}
```

A while loop provides a way for your program to repeat portions of itself as many times as necessary. When I use the term “many”, I mean zero or more times. A while loop repeats as long as the condition, the expression within the parentheses, is true. Note that the condition may initially be false; in that case the body of the while loop is not executed at all.

What do these while loops actually do? The first loop looks at a copy of the identifier we are validating and asks “does it contain a dash (a hyphen)”? The second loop looks at the copy of the identifier and asks “does it contain any spaces?”

The indexOf method is the key, since it tells where the target is found within the String variable named result. If the target is not found, indexOf returns the value -1.

If the target is found (that is, the identifier contains a dash (hyphen)), the portion of the identifier before the target and the portion of the identifier after the target are combined to form a new identifier. In essence, we have deleted the position of the identifier in which we found the target. We need to do this roundabout calculation since there is no method to delete a character from a String. That is another way of saying that String objects are immutable.

If that bothers you, perhaps you should investigate the StringBuffer class.
Suppose the identifier is 300-106-785. When we seek a dash, the indexOf method finds a dash at position 3. Positions within a String start at 0; that is why the value of -1 is used when the target is not found.

Since we have a dash at position 3, we break the identifier in two pieces, one consisting of positions 0 through 2, the other consisting of positions 4 through 10, and combine them into a shorter String, 300106-785. Note that the parameters to the substring method are the first position to keep and the first position to not keep. That is, the parameters are the first position to keep and one more than the last position to keep. If there is only one parameter, substring returns everything from that position to the end of the string.

Within the body of the loop, indexOf looks at 300106-785 and finds a dash in position 6. That is done by the last statement in the while loop, so the program loops back to the while condition and asks “is there still a dash in the number?” Since the indexOf method has returned the value 6, the loop knows there is another dash. The body of the loop removes the dash by breaking the String into two pieces, one for positions 0 through 5, and one for positions 7 through 9, and then combining the pieces to give 300106785.

The second statement in the body of the loop again looks to see if there is a dash in the resulting string. The answer is no, so indexOf returns -1.

The while condition sees that indexOf has returned -1, so terminates, leaving us with the String 300106785.

What happens if the identifier provided was 300106785? That contains no dashes or hyphens, so indexOf returns -1 before the loop begins, the condition is false, and the body of the loop is not executed at all.

In a similar way, the second while loop removes spaces from the identifier. It may be difficult to see but target = ' '; contains a space between the two single quotation marks.

Once we have removed the dashes and spaces, then we can continue and check if the identifier is too short or too long.

Thus, the general structure of our validation method can be described as “Clean up the data if possible and then check for problems.”

**Testing the exceptions**

To test these two exceptions, we must write some unit tests, in a ShortIdentifierExceptionTest class, and in a LongIdentifierExceptionTest class.

Let's begin with the tests for ShortIdentifierException.
ShortIdentifierExceptionTest

What test do we need? For thorough testing I suggest we need the following tests.

- Validate a good identifier and ensure that the validation method does not throw an exception.
- Validate a short identifier and ensure that the validation method does throw an exception, and that the message associated with the exception is the one we expected.
- Validate an identifier which contains dashes or spaces, and ensure that the validation does throw an exception if the identifier without the dashes or spaces is too short and that the message associated with the exception is the one we expected.

Test – good identifier

Here is a possible first test.

```java
@Test
public void testNoProblem() {
    try {
        String identifier = "300106785";
        assertEquals(identifier, Person.validateIdentifier(identifier));
    }
    catch (ShortIdentifierException e) {
        fail("Unexpected ShortIdentifierException: " + e.getMessage());
    }
    catch (LongIdentifierException e) {
        fail("Unexpected LongIdentifierException: " + e.getMessage());
    }
    catch (Exception e) {
        fail("Unexpected exception: of type " + e.getClass().getName() + " with message " + e.getMessage());
    }
}
```

All the calls to the fail method have a String parameter we construct that identifies the type of exception which has occurred and displays the message associated with the exception. In the first two calls we know the type of exception and we hardcode the type in the message. In the third call to fail, we know there is an exception, but we have to ask the exception (via the expression e.getClass().getName()) what its type is. Hopefully we never see the result of this third call, since it should never be called. That is, the only exceptions which occur should be ShortIdentifierException and LongIdentifierException; no other exceptions should occur.

In all the calls to fail, e.getMessage() returns the message associated with the exception.

Try and catch blocks

Before we examine the test in detail, note the use of the try and catch blocks.
Any time you write statements which might throw an exception, you need to enclose those statements in a try block, even when you know that the exception should not be thrown because nothing can go wrong this time. Following the try block you need one or more catch blocks. The catch blocks catch the exceptions thrown and react properly.

Now consider this test. We are executing the Person.validateIdentifier method, which we know from its header throws two types of exceptions. In this test, we know that the identifier is good, so there should be no exceptions thrown.

But Person.validateIdentifier says it may throw exceptions, so you need to handle the possibility of an exception. If there is an exception of any kind, whether the two we expect or some other one, the test fails. We indicate that by calling the fail method with a message that is part ours and part from the exception itself.

Note that there are several catch blocks all associated with the same try. This is very common.

We know that the test is successful if the value returned by the validateIdentifier method matches what we predict it should. If it doesn't match, that is another way for the test to fail.

**Test – short identifier**

Now consider a test for an identifier which is too short, and does not contain dashes and spaces.

```java
@Test
going public void testIdentifierTooShort1() {
    String identifier = "1234567";
    try {
        String result = Person.validateIdentifier(identifier);
        fail("should not be created");
    }
    catch (ShortIdentifierException ie) {
        assertTrue("Identifier should contain 9 characters. " + identifier + " only contains " + identifier.length() + "," + ie.getMessage());
    }
    catch (Exception e) {
        fail("Unexpected exception: of type " + e.getClass().getName() + " with message " + e.getMessage());
    }
}
```

Since the identifier is too short, validateIdentifier should throw an exception. The statement after the method call should not be executed; if it is, then the test fails.

The validateIdentifier method should throw a ShortIdentifierException. By examining validateIdentifier, we can determine the exact contents of the exception's message. By comparing
the expected value of the exception's message to what we receive, we can see if the test passes or fails.

If we find another, unexpected, exception thrown, the test fails. We catch this unexpected exception in a second catch block.

**Test – short identifier, part 2**

Now consider a test for an identifier which contains dashes and spaces. The problem we are dealing with is an identifier which becomes too short when it is cleaned up.

Note that this test is based on a real case. A program I use allows you to paste in an identifier, but it only accepts nine characters. Many people provided the identifiers with dashes, so what was pasted in was, for example, 300-106-7.

```java
@Test
public void testIdentifierTooShort2() {
    String identifier = "300-106-7";
    try {
        String result = Person.validateIdentifier(identifier);
        fail("should not be created");
    } catch (ShortIdentifierException ie) {
        assertTrue("Identifier should contain 9 characters. " + identifier + " only " + "contains " + identifier.length() + ",", ie.getMessage());
    } catch (Exception e) {
        fail("Unexpected exception: of type " + e.getClass().getName() + " with message " + e.getMessage());
    }
}
```

This test will succeed, but the message displayed will cause confusion. 300-106-7 contains nine characters, including the dashes. But the error message says it contains seven. To clarify this, I would add the phrase “, excluding punctuation” to the error messages produced.

That is, the two if statements in validateIdentifier should change to the following.

```java
if (result.length() < 9)
    throw new ShortIdentifierException("Identifier should contain 9 digits. " + identifier + " only contains " + identifier.length() + ", excluding punctuation.");

if (identifier.length() > 9)
    throw new LongIdentifierException("Identifier should contain 9 digits. " + identifier + " contains " + identifier.length() + ", excluding punctuation.");
```
LongIdentifierExceptionTest

In the same way we created a ShortIdentifierException, we can create a LongIdentifierException and its tests.

The tests will be similar to those for a short identifier.

- Validate a good identifier and ensure that the validation method does not throw an exception.
- Validate a long identifier and ensure that the validation method does throw an exception, and that the message associated with the exception is the one we expected.
- Validate an identifier which contains dashes or spaces, and ensure that the validation does throw an exception if the identifier without the dashes or spaces is too long and that the message associated with the exception is the one we expected.

Make it so.

NonnumericIdentifierException

We can now be sure that the identifier contains exactly nine characters. But how do we ensure that those nine characters are all digits. We want to ensure that a user does not give us an identifier like 3OO106785 where the second and third characters are the letter O, not the digit 0.

As we have done twice already, create a new exception class, NonnumericIdentifierException.

Then we need to add some statements to validateIdentifier which will ensure that the nine characters are all digits. There are two ways to do this. The first involves a loop, the second involves exceptions.

Using a loop

```java
int j = 0;
while (j < 9) {
    if (!Character.isDigit(result.charAt(j)))
        throw new NonnumericIdentifierException(result + " contains a character at position " +
            j + " which is not a digit.");
    j++;
}
```

How does this loop work? And how does it decide to throw an exception?

Notice that result must contain nine characters, numbered zero through nine, since this statement comes about those checking the length. We need a variable to represent the position of the character; that variable is j. It starts at zero. As long as it is less than nine, the body of the loop is executed. The final statement within the body changes j, by adding one to its value. The condition tests if the value of j is still less than nine. If so, the body is executed. If not, the loop terminates.
While the loop is executing, we need a way to access the individual characters. When you examine the documentation for String, you will notice a charAt method. This returns the character at the specified position in the String. charAt(0) is the first character in the String, charAt(1) is the second. charAt(8) is the final character in a nine-character String.

If the parameter you provide for charAt is negative or greater than the number of characters in the String, charAt will throw an IndexOutOfBoundsException. This is a runtime exception.

Now that we have a way to extract individual characters, we need a method to determine if a character is a digit. Looking in the Character class (This class is a wrapper class, one which contains within it a primitive datatype. I don't know why the class is called Character while the primitive type is char.), we find the isDigit method. It is a static method so must be referred to as Character.isDigit with a character as a parameter.

!Character.isDigit(result.charAt(j)) is the way we say that the character at position j of the String named result is not a digit. Recall that ! is the symbol for not. Whenever we find a non-digit, we stop processing and throw the NonnumericIdentifierException.

Note that we are not determining all of the non-digits, nor are we determining how many non-digits there are. We are simply indicating that we have found a non-digit.

Here is a suitable unit test.

```java
@Test
public void testBadCharacters(){
    String identifier = "3oo106785";
    try {
        String result = Person.validateIdentifier(identifier);
        fail("should not be created");
    } catch (NonnumericIdentifierException ie) {
        assertEquals(identifier + " contains a character at " +
                        "position 1 which is not a digit.", ie.getMessage());
    } catch (Exception e) {
        fail("Unexpected exception: of type " + e.getClass().getName() +
                        " with message " + e.getMessage());
    }
}
```

An alternative loop is called the do-while loop. It is used when you know that you must perform the loop at least once.

```java
int j = 0;
do {
    if (!Character.isDigit(result.charAt(j)))
        throw new NonnumericIdentifierException(result + " contains a character at position " +
                                                j + " which is not a digit.");
```
j++;  
} while(j <9);

Recall that I mentioned there are two ways to determine if the nine-character identifier is numeric. Two different loops structures were not what I had in mind.

**Using an exception to throw an exception**

Since we are dealing with exceptions, it may be appropriate to use a technique which throws an exception to determine if we need to throw a NonnumericIdentifierException.

A nine-digit number is small enough to be considered an integer. Consider the Integer class. Specifically, consider the constructor for Integer, the one which accepts a String as a parameter and throws a NumberFormatException.

Consider the following statements.

```java
try {
    Integer temp = new Integer(result);
} catch (NumberFormatException nfe) {
    throw new IdentifierException("Identifier " + identifier + " is not numeric.");
}
```

We attempt, okay “we try”, to create an Integer but we do nothing with it if we succeed. But if we fail, we know there is a problem with the identifier we were provided and thus we can throw our own exception. Note that we convert identifier, not result. Why is that?

You can replace the while loop we just looked at with this try/catch block. Note that if you do this, you'll need to revise the unit test since the error message is different.

**Finally**

In addition to the catch block, there is also a finally block. As described in the Java tutorial, http://download.oracle.com/javase/tutorial/essential/exceptions/finally.html,

> “The finally block *always* executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break. Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.”

So far, we have seen no cases where there is “cleanup code” necessary, but we may see some eventually.
Note that a finally block will be executed unless the try block contains `System.exit` or the Java virtual machine crashes.

Consider http://download.oracle.com/javase/tutorial/java/nutsandbolts/branch.html for a broader discussion of the `continue`, `return`, and `break` statements.

**Summary**

Exceptions are the powerful technique Java uses to handle errors. We will see many more exceptions in the chapters that follow.
Exercises

1. Create validateName, a static method within the Person class to determine that both a first and a last name are provided. The method throws a NameException if either name is missing.

In Indonesia, many people go by just one name. How could you determine that you are in a country where people use only one name?

There are many stories about programs that have difficulty with names. One story is about a payroll system in which last names with only one character were used for testing purposes. That worked well until a person with the last name A joined the organization.

A second story was about the person who didn’t have a long first and middle name. His name was something like R B Jones. The payroll system threw exceptions for one-character names. So the data entry people entered the name as R(only) B(only) Jones. The people who wrote the system stripped out unusual characters from the input. Thus R B Jones became Ronly Bonly Jones.

2. There is yet another way to check that the identifier in numeric. Explore the Java documentation to explain how the following statements work.

```java
// Regular expression borrowed from
// regexlib.com/DisplayPatterns.aspx?cattabindex=2&categoryId=3
Pattern p = Pattern.compile("^\d+$");
Matcher m = p.matcher(str);
if (!m.matches())
    throw new IdentifierException("Identifier " + identifier + " is not numeric.");
```

3. Explore the Exception class to see what other types of exceptions are available to you.
Chapter 6 – An Address class

Learning objectives

By the end of this chapter, you will be able to:

- Create more-complicated classes
- Explain when and why to use cloning
- Use the online Java documentation to find out about classes.

Introduction

A college or university needs to know the mailing address of its students, so the Registrar’s Office can send written communications, including transcripts. (A transcript is an official record of the marks a student earns in the courses she takes.) The Library will use that address to send notices of library fines.

Of course, the college may need to have other ways of communicating with students. Given the problems with violence on campuses, many institutions want email and/or cell phone (mobile phone) numbers they can use to contact students in case of emergencies.

Professors also need to provide an address and contact information, this time to the Human Resources department.

In addition, an institution needs to know its own address. It may have a mailing address and a separate street address, or they may be the same.

An Address class is the focus of this chapter. A phone number is not so interesting. It’s just a String. Email addresses are not so interesting. They too are just Strings.

The interesting discussion is around a person’s mailing address. Let’s begin there.

Adding an address

Let’s now add an address to the Student and Professor classes. Stop! Do not pass go! What is wrong with the statement?

We do not need to add an address to both classes. We can add an address to the Person class and both Student and Professor will have access to it.
So what is an Address class? What makes up its state and its behaviour?

The Address class

What fields do we need to form an address?

You could use one long String, but, as we have seen earlier when we considered names, it may be more efficient to decompose the address into smaller fields and combine them to form larger combinations as needed. I would suggest the following fields:

- Number.
- Number suffix. Perhaps you live in a suite in a house and have an A after your address. 1703A Yates Street is an example.
- Name.
- Type. Road, Street, Parkway, or Crescent, are examples. This reminds me of the old question “Why do we drive on a parkway and park on a driveway?” or “If a train stops at a train station, what happens at a workstation?” Isn’t the English language wonderful?
- Direction. Some towns are divided into portions based on compass directions, giving addresses like 8743 12 St NE.
- City.
- Province (or state) or other subdivision of the country.
- Country.
- Postal Code. Different countries use different formats for their codes. We will not deal with that here.

To see where the ideas for these fields arose, consider the Canada Post postal code lookup page at http://www.mailposte.ca/tools/pcl/bin/advanced-e.asp

We could add all these fields to the Person class, but don’t you think it would be better to create an Address class (especially since it’s not just Person that may use this class) and then allow the Person class to contain an instance of the Address class? Yes, you could make the same argument about a Name class. In fact, if you have some spare time, it would be a very good idea to create a Name class.

For complete generality, you should also add an apartment (or unit) number to the address. My example will not do so. If you do include the apartment number, you may wish to have two constructors, one with an apartment number, and one without.

Return to BlueJ and create a new class, named Address. Double-click Address and enter the Java statements that it requires.

What statements are those?

In addition to the ones BlueJ suggest for you, you need private instance variables, the details of the constructor, getters and setters, and a toString method. Of course, you also need the comments that are appropriate.
The Address class – the code

My code follows.

/**
 * a class to contain a street address, excluding internationalized postal code
 * @author rick
 * @version 1 – april 2006
 */
public class Address {
    // instance variables
    private String number;
    private String suffix;  // the number followed by the suffix could give 1702A, for example
    private String name;
    private String type;
    private String direction;   // NW, for example
    private String city;
    private String province;
    private String country;
    private String postalCode;

    /**
     * Constructor for objects of class Address.
     * @param number - the number on the street
     * @param suffix - a suffix to the number (1702A)
     * @param name - the name of the street
     * @param type – road, street, crescent, etc.
     * @param direction - for a city divided into NW, SE.
     * @param city - the city, town, or village
     * @param province - two-character abbreviation
     * @param country – the country
     * @param postalCode - the postal code.
     */
    public Address(final String number, final String suffix,
                   final String name, final String type,
                   final String direction, final String city,
                   final String province, final String country,
                   final String postalCode)
    {
        this.number = new String(number);
        this.suffix = new String(suffix);
        this.name = new String(name);
        this.direction = new String(direction);
        this.type = new String(type);
        this.city = new String(city);
        this.province = new String(province);
        this.country = new String(country);
        this.postalCode = new String(postalCode);
    }

    /**
     * toString - convert the address into something suitable for a mailing label.
     */
public String toString() {
    String result = new String();
    result = number;
    if (suffix.length() > 0)
        result += suffix;
    result += " " + name;
    result += " " + type;
    if (direction.length() > 0)
        result += " " + direction;
    // end of line 1
    result += '\n';
    result +=city;
    // end of line 2
    result += '\n';
    result += province;
    result += ' ' + country;  // one space
    result += " " + postalCode;  // two spaces
    // end of line 3
    return result;
}

If you think documentation is not worth writing, press \[Ctrl]-J\] while viewing the Java code of the Address class. Isn’t that a nice piece of documentation that appears? And you didn’t have to do anything special to have it appear, other than provide a few statements in your code. Press \[Ctrl]-J\] again to return to your source code.

If you prefer not to use the keyboard to see the documentation, simply use the dropdown box at the top right corner and select between Source Code and Documentation.

The Address class – the code – in detail

Let’s look at my code in some detail. The constructor contains nothing new, but there are some interesting things in toString.

First, each person develops a different programming style. Despite what you see in some of my examples, I prefer to write many short statements while concatenating Strings. Some people write fewer and longer statements. For example, instead of

```
result += " " + name;
result += " " + type;
```

you could write

```
result = result + " " + name + " " + type;
```
Some people combine the two approaches and write

```
result += " " + name + " " + type;
```

Recall that \( x += y; \) is shorthand for \( x = x + y; \) There are other shorthand operations including -=, *=, and /=.

However you write it, the effect is the same, concatenating the current value of result, a blank, the current value of name, a blank, and the current value of type, and saving the resulting String in the variable named result, replacing its previous value.

Why the blanks? So the output is “1702A Fifth Street” (with a blank between A and Fifth, and between Fifth and Street), not “1702AFifthStreet.”

And what is this ‘\n’ that appears in two places in toString?

A String is delimited by double quotation marks. A String may contain zero or many characters. If you want a single character, a String may be overkill. (Yes, I have used a single-character String for the space between fields.) So Java contains a character datatype, called char. Each letter of the alphabet is a character; each digit of a number is a character; each punctuation mark is a character. Single characters are enclosed in single quotation marks. ‘ ‘ is a single character, a blank. ‘+’ is a single character, a plus sign.

A tab is a character, represented by ‘\t’, and a carriage return and linefeed combination is a character, represented by ‘\n’.

In the days of the typewriter, a carriage return moved you from the current position on a line to the beginning of the line. A linefeed moved you down one line. Since the carriage return didn’t move down a line, you could type over what you had typed. This allowed you to produce some interesting effects. See the Wikipedia article on ASCII art for some examples. [http://en.wikipedia.org/wiki/ASCII_art](http://en.wikipedia.org/wiki/ASCII_art)

You may even wish to see the article on typewriters if you don’t know about them. [http://en.wikipedia.org/wiki/Typewriter](http://en.wikipedia.org/wiki/Typewriter)

Some operating systems treat a carriage return and a linefeed as two separate characters, but that detail is hidden from us when we are using Java.

So ‘\n’ is the representation of the carriage return and linefeed combination. Despite appearances, ‘\n’ is treated as a single character.

Concatenating a char onto a String creates a new String, one which contains the command to display the rest of the String on a new line.

My toString method produces a three-line address that is suitable for printing on a mailing label.
Adopt what you like from my `toString` method, compile it, and test it. When you inspect an address, BlueJ shows the `\n` character instead of moving to a new line.

To see the output as it is meant to be seen, create an `Address` object on the workbench. Accept the suggested name of `address1` and provide the parameters you wish.

Then, from the main menu, select View, Show Code Pad. The workbench narrows and a small window appears in the space it vacated. In that window, you can type Java statements for immediate execution. Type `System.err.println(address1);` and press `Enter`.

A Terminal Window opens, showing the output of the `println` command, with different parts on different lines in the lower portion of the window. If we had used `System.out.println`, the output would have appeared in the upper portion of the window.

`System` is a class which allows your program to communicate with the keyboard and the screen. We will use it mainly for output.

Output to the screen is through either the `err` device or the `out` device. As noted earlier, `out` output appears in the top panel of the Terminal Window; `err` output appears in the bottom panel.

`println` is a method which displays the `String` it is given. If it is not given a `String`, it calls the `toString` method of the object it is given, in this case, `address1`, and displays its result on the requested device. This is the main reason that every class should have a `toString` method.

After displaying the output, the cursor in the Terminal Window moves to the following line, ready to display more output.

When I am testing a method which produces a long `String` as output, I will sometimes display the result in the Terminal Window rather than using an `assert` statement by using `System.out.println` within a unit test. I have to look at the output to decide whether it is correct, but appearance may be part of its correctness.

**The Address class – getters and setters**

Add the first of the getters and setters. I don’t care which ones they are.

Compile the class. Now create the unit test for the getter and setter. Hint: you can have many, here two, editor windows open on your screen at once. Use one to create the class and the other to create the unit tests for the class. Ensure you compile and run the tests after you create them.

Now create the other getters and setters, one pair at a time. Compile the class after you make each pair. Add, compile, and run unit tests as you do so.
Person uses Address

Now that the tests are complete and all are successful and we know that Address objects work properly, how do we add an instance of Address to Person? It’s simple; declare an instance variable.

protected Address homeAddress;

Recall that using the word protected means that derived classes may access this variable.

Did you notice how a “uses” arrow appeared on the class diagram when you insert that statement, and save the file?

A “uses” arrow is an indication that objects of one class (the one at the blunt end of the arrow) contain objects of another class (the one at the sharp end of the arrow.) This is quite different from the “extends” arrow.

We are not saying that an Address is a Person. We are saying that a Person object contains one or more Address objects. At the moment, a Person will contain only one Address object, a home address. But students often do not live at home, so may need another address.

Does everyone have an address? They should have an address, so we add the address as a parameter to the Student, Professor, and Person constructors, including documentation.

@param homeAddress the home address

Then we save the value provided to the instance variable.

this.homeAddress = homeAddress;

When we have saved String objects, we have made a copy of the object, by invoking a constructor within the String class, a constructor which accepts a String as its parameter. We don't have such a constructor for Address. Will this cause a problem?

Yes, it will. When an Object (and every parameter we have seen so far is an Object or is derived from an Object.) is passed to a method, the method does not receive a copy of the Object, it receives a reference to the Object (essentially an address in memory). When you have a reference to an Object, it is possible to use methods of the Object to change its contents.

In particular, when two Student objects refer to the same Address object, any change to the Address object will affect both Student objects, since they both contain references to the same Address object. Consider roommates, one of whom later moves.

Exercise 1 provides directions to see how this can happen. If you are going to do that exercise, do it now, rather than waiting until you have fixed the problem.
We could solve the problem by creating a constructor whose signature is `public Address(final Address a)` or we could implement a clone method. I know that you could create the constructor, so we'll look at cloning instead.

**Making copies (cloning)**

Note: This section (and others later) deals with an advanced topic. It is included here because this is the correct place to talk about it. But you may skip it should you wish. If you do so, be aware that your program will contain an error which should be fixed, perhaps when you are more comfortable with Java.

The parts of a person’s name (and the parts of a person's address) are Strings, and a String is a special type of `Object`, called *immutable*. As we have seen, that means it cannot be changed without totally rewriting it; it cannot be changed by adding or deleting a single character. It is a special kind of `Object` since most `Object`s can be changed.

The implication of this is that we do not need to clone the parts of a name, but we do need to clone an address.

Consider the following statements. You can enter them, and the tests I describe, in the CodePad window.

```java
String s1 = "here is a string";
String s2 = "and here is another string";

The two Strings are not equal when you compare them using the `equals` method (which compares contents of strings) since their contents are different, nor are they equal when you use the logical operator `==` (which compares hashcodes, often implemented as addresses of memory) since their contents are in different areas of memory.

You can see this by using `System.err.println(s1.equals(s2))`; and `System.err.println(s1 == s2)`;

The food called hash is a mix of a variety of different ingredients. Some people say it looks like pet food. The idea behind the computer technique called hashing is similar. Take the value of the `Object` and mix it up some way, producing a single number. That number is an indication of the contents of the `Object`. Two `Object`s with identical values should give the same hash code.

Now consider the statement

```java
String s3 = s1;
```

After this statement, `s1` and `s3` will compare as equal using the `equals` method (since their contents are identical) and when using the logical operator `==` (since they both refer to the same area in memory.)

Now consider the statement
s1 = "this is a different string";

s1 and s3 will compare as not equal when using the equals methods (since their contents are different) and when using the logical operator ==. Before executing the statement, s1 refers to an area of memory which contains “here is a string.” But s1 is immutable. When we change its contents, we do not change the contents of the area of memory associated with it, we allocate another area of memory, containing the different contents. Thus s1 and s3 refer to different areas of memory.

If this is unclear, draw a picture to help.

An Address is not an immutable Object; it can be changed quite simply. Thus, we should make a copy of it when we store it as part of a Person. Making an exact copy of an Object is called cloning the object.

Exercise 1 in this chapter asks you to explore the following scenario. Suppose a group of students share an apartment, and the same Address object is used to represent that address for each of the roommates.

Several Person objects now have a reference to the same Address object. If one of the roommates moves away, changing his/her address, that address change will be reflected for all the other roommates as well, even though they have not moved!

To clone an Address object, we must resort to a little deception. I’ll show you the code you need, but you must promise not to ask how it works, at least not for a while.

Open the Address class in the editor and modify the class declaration to read public class Address implements Cloneable

Then create a new method in the class.

```java
/**
 * @return a copy of the Address
 */
public final Object clone() {
    try {
        return super.clone();
    } catch (CloneNotSupportedException e) {
        return null; // should not happen
    }
}
```

This method works since all the instance variables of an Address are Strings and know how to clone themselves. If any instance variable did not know how to clone itself, this method would not work. In that case, of course, we would implement clone for the class to which those instance variables belong.
Note that if you do not make the class cloneable (by forgetting to write public class Address implements Cloneable), the CloneNotSupportedException will be thrown and you will not have a clone of the object.

There are two other ways to implement the clone method.

In the first, we create a nullary constructor, which we will see again later, a constructor which has no parameters. This will create an object whose instance variables are initialized to nulls or empty strings. You use a nullary constructor in conjunction with your setters.

Here is the nullary constructor.

```java
public Address () {
    this.number = "";
    this.suffix = "";
    this.name = "";
    this.direction = "";
    this.type = "";
    this.city = "";
    this.province = "";
    this.country = "";
    this.postalCode = "";
}
```

And here is the clone method that uses the nullary constructor.

```java
public final Object clone() {
    Address result = new Address();
    result.number = new String(this.number);
    result.suffix = new String(this.suffix);
    result.name = new String(this.name);
    result.type = new String(this.type);
    result.city = new String(this.city);
    result.province = new String(this.province);
    result.country = new String(this.country);
    result.postalCode = new String(this.postalCode);
    return result;
}
```

Now that we have the code to clone an Address, we can use the correct statement in the Person constructor.

```java
this.homeAddress = (Address) homeAddress.clone();
```

The word final in the method definition simply says that should any class use Address as a base class, it cannot create a clone method. This is a security issue. See more details at http://www.ftponline.com/javapro/2004_04/online/rule5_04_07_04/
If you are uncomfortable about using code which you do not understand, the following method also clones an Address object.

```java
public final Object clone() {
    return new Address(number, suffix, name, type, direction, city, province, country, postalCode);
} // end clone()
```

So now we have three ways to make a copy of an Address, or any other object we create. We can write a clone method in one of two ways or we can create a constructor whose parameter is an object of the class.

If we adopt the third route, we can use the statement

```java
homeAddress = new Address(homeAddress);
```

**The Address class – unit testing**

Create the getter and setter methods for Address. The getAddress method returns an Address. Create the unit test for the Address instance variable in the Student and Professor classes.

I had some difficulty with assertEquals when checking the equality of objects, like the address. You have to check that the corresponding parts are equal, thus creating a complicated test. One solution, which I do not like, is to place many assert statements in one test. The test passes if all pass. But I prefer to have one assert statement per test. Thus the solution is to use

```java
assertTrue(a.toString().equals(b.toString()));
```

where a is the variable containing the expected value and b is the variable containing the actual value.

Of course, you could write an equals method for an Address object, a method which returns true if all the parts match, false otherwise.

**Testing Address cloning**

Before we create the Address and Student objects, we should modify the toString method of the Person class to display the address. The simplest way to do this is to use the toString method in the Address class.

Before the last line in Person’s toString, place the line

```java
result += " Address: " + homeAddress.toString();
```

Recompile everything.

How do we create an Address object to pass into the Person constructor? One way is to use the Object Workbench.
Create an Address object by right-clicking the Address class and choosing the constructor.

Note the name of the object which is being created (probably the name is `address1`, but you can change the name if should you wish.), and provide values of its instance variables.

Create a Person object by right-clicking the Person class and choosing the constructor. Most of the values you provide are Strings, but the last is the name of the Address object you created earlier. Strings need to be surrounded in quotation marks but the address, the name of a variable, does not.

A second, in my opinion better, way is to modify the unit test for the `toString` method in Person. Once the Student objects have been created, check their addresses. In particular, what happens when there is no address available?

### The reserved word `null`

We have assumed that everyone has an address which they are willing to provide. What happens when someone does not have an address or does not wish to provide it? What do we provide to the constructor in that case?

Java contains the reserved word `null` to use in such cases. When we allow null addresses, we need to be sure that we neither clone a null object nor should we attempt to convert a null object to a String.

Thus we must make two changes to our code. First, before cloning the address, we must check it is not null. This applies in both the constructor and the `setAddress` method.

```java
if (homeAddress != null)
    this.homeAddress = (Address) homeAddress.clone();
else
    this.homeAddress = null;
```

Second, instead of simply calling the `toString` method of the instance variable `homeAddress`, we need to check whether `homeAddress` is null.

```java
if (homeAddress != null)
    result = result + '\n' + homeAddress.toString();
```

After making the changes to allow for a null address, recompile your classes and test your work by creating two Student objects, one with an address, and one with a null address. (Just provide the word `null` instead of name of the Address object.) Does your `toString` method work properly? If not, fix it.
The College class and addresses

As noted earlier, a college may have two addresses. One is its street address, providing the actual location of the college. The other is its mailing address, which may or may not be the same as its street address.

Modify the College class so that it has instance variables for both the street address and the mailing address. We will not add parameters to the constructor. The constructor will simply set both addresses to null. We will use the setStreetAddress and setMailingAddress methods to set the appropriate values.

A second constructor is one which provides both addresses.

A third constructor is one which provides only one address. You need to decide which is the most likely to be provided. I would suggest that the street address may be the one you wish to use.

Yes, a class may have several (more than one) constructors. We will see multiple constructors later when we explore ways to save the contents of objects.

Create unit tests for both addresses. You’ll need separate tests for addresses which are missing and for addresses which are provided, four unit tests in all.

Summary

While an address appeared simple, it turned out to be quite complex. And there are complexities we have omitted!

Note that everything we have said about a Student object so far applies to a Professor object. There is no difference in the structure of the two types of object. There certainly is a difference in the roles they play at the College. Let’s look at one more structure they have in common, and then we’ll start to look at their differences.
Exercises

1. Confirm the need for cloning by the following procedure.

   Create an Address object on the Object Bench.

   Create two Student objects on the Object Bench, both of which contain references to the same Address object. Use the getAddress methods in each Student object to see that the addresses are both the same.

   Use the setNumber method of the Address object to change its number.

   Then use the getAddress methods for both Student objects to see that both now contain the new number. The effect is that whenever one student moves and updates the address, the other student will appear to have moved as well.

2. Use the CodePad window to verify that Strings do not need cloning by repeating the tests described in this chapter. What happens when you do the tests?

   Then do another test. Add another statement.

   s1 = "here is a string";

   Is s1 equal to s3 or not?

3. Create a unit test that verifies that Strings do not need cloning.

4. The Person, Student, and Professor classes we have developed all use instance variables to represent the different parts of a name. Since we created a class for Address, perhaps we should do the same for Name. Design and implement such a class.

   If you do so, you should also move the method to ensure that at least one part of the name is provided is moved to the Name class.

   For an interesting perspective on including a title as part of a name, see this article from the March 2011 issue of Wired. http://www.wired.com/magazine/2011/02/st_dropdownmenus/

5. Internationalization is becoming more and more important. Suppose you are dealing with a culture in which not every home has an address. It may just have a name, or a description. How would you modify the Address class to support internationalization?

   Make it so.

6. Internationalization is becoming more and more important. That is, a program is not necessarily used in the country it is written. Identify some of the different ways in which
postal codes differ from country to country and write code to verify that the code entered matches the country. That is, create an AddressException class and use it to ensure the postal code matches the format of the country's postal code, and then use it to check for any peculiarities in the postal code. For example, the Canadian postal code does not use some letters of the alphabet. See http://www.canadapostalcode.info/canadian-postal-codes.html for details.

First, of course, you’ll need to add a country instance variable to the Address class.

Then, explore the postal codes for Canada, USA, United Kingdom, or some other countries which are different from your own.

7. Explore the difference between a shallow clone and a deep clone.
Chapter 7 – Dates

Learning objectives

By the end of the chapter, you will be able to:

- Create more-complicated classes
- Work with dates
- Use the online Java documentation to find out about classes.

Introduction

Many things at a college are based on seniority. Perhaps professors who have been there the longest have first choice on middle-of-the-day teaching times. Thus professors need to know the date on which they were hired.

Perhaps students who have been registered the longest are allowed to register first for a new semester, probably so they have a better chance of completing their program of study. You will explore that idea in the exercises at the end of the chapter.

Perhaps students who applied first will be allowed to register for classes first.

Another use of dates is that sections of a course have specific times for meeting. We will explore this in a later chapter.

All of these involve the use of dates and, perhaps, times. This chapter will explore dates.

How old is a person?

In many cases, you need information about the age of a person. At Okanagan College, for example, a professor is required to retire on the June 30 following his/her 65th birthday. In some other situation, an employee may not be required to pay a certain type of tax or make contributions towards a pension if he/she is too young or too old.

Mandatory retirement was in place when this chapter was first written. Since that time, it has been abolished. But the example is a good one, so I’ll continue to use it.

But you don’t want to store the age, because it changes every day. It would be better to store the birthday and calculate the age whenever it is required. To do that you need to be able to store a date. What facilities does Java offer to deal with dates?
When you look in the Java documentation, you will find two `Date` classes.

One, in the `package` named `java.sql.Date` appears to be used with databases. (Package is a Java term referring to a collection of classes. There is a hierarchical structure to these packages. Thus the `Date` class is within the `sql` package. Have you noticed that BlueJ displays the hint “Compile all uncompiled classes in this package” when you hover your mouse over the Compile button beside the class diagram? In this case, the term package refers to all files in your project.)

Many of these `Date`’s methods (including the constructor which accepts year, month, day) are deprecated. That means that they have been superseded by better methods in other classes; future releases of Java may not include deprecated classes or methods, so it is best not to use them. Reading the documentation closely, we find that this `Date` class is derived from the other `Date` class.
The other `Date` class, in the package `java.util.Date`, appears much more useful to us, although most of its methods (again including the constructor with year, month, and day arguments) have been deprecated.

Here, though, guidance is given on what to use in its place. There are suggestions to use the `Calendar` or `GregorianCalendar` classes. Would one of these be what we need, or are they more complicated than we need?

**Details about calendars**

`Calendar` is an abstract class (in the same way we have developed `Person` as an abstract class), used to derive other classes. Rather than doing that derivation ourselves, we should explore `GregorianCalendar`.

The documentation describing `GregorianCalendar` provides an interesting, but brief, review of the way dates have been changed in the past, moving from a Julian calendar to a Gregorian one (for some countries) in 1582 or some other year.

Did you ever wonder why some countries and religions celebrate Christmas on December 25 and others celebrate it on January 6? It’s a result of the change to the calendars in some cultures and the reluctance of people in others to switch.

The conventions in `GregorianCalendar` regarding values used for instance variables are as follows:

- A year `y` is represented by the integer `y - 1900`.
- A month is represented by an integer from 0 to 11; 0 is January, 1 is February, and so forth; thus 11 is December.
- A date (day of month) is represented by an integer from 1 to 31 in the usual manner.
- An hour is represented by an integer from 0 to 23. Thus, the hour from midnight to 1 a.m. is hour 0, and the hour from noon to 1 p.m. is hour 12.
- A minute is represented by an integer from 0 to 59 in the usual manner.
- A second is represented by an integer from 0 to 61; the values 60 and 61 occur only for leap seconds and even then only in Java implementations that actually track leap seconds correctly. Because of the manner in which leap seconds are currently introduced, it is extremely unlikely that two leap seconds will occur in the same minute, but the specification follows the date and time conventions for ISO (International Standards Organization) C.

I (and others) have to wonder why the day of the month starts at 1 but all other fields (including month) start at 0. Yes, I know January is the first month, but it is an interesting inconsistency that the month variable starts at 1 while everything else starts at 0.

In English, the names of some of the months represent older calendars. September, October, November, and December begin with the Latin words for seven, eight, nine, and ten, despite being the ninth, tenth, eleventh, and twelfth months of the current calendar.
Why might this be?

Who had a year that was 10 months long?

The documentation on the first Date class also includes an interesting note. “In all cases, arguments given to methods for these purposes need not fall within the indicated ranges; for example, a date may be specified as January 32 and is interpreted as meaning February 1.” That makes it easier to decide when 10 days in the future is, if you cross from one month to another.

How do we use so much information on dates and calendars?

Can we even use this information?

**The MyDate class – a simplification**

For now, we will use none of the provided classes; they are more powerful, hence complicated, than we need. Instead, we’ll create our own class, MyDate.

What is advertised as a better date is available in the open source project Joda-Time. It is described at http://joda-time.sourceforge.net/. Version 2.1 was released on 2012-02-22.

There is a proposal this project added to Java, where it would be used as a replacement for the existing Date classes.

Why not just create a BirthDate class, since that is what introduced this section?

Thinking ahead, we may be able to use other types of dates, including the date a student graduated, the date an employee was hired, and the date an employee was promoted. We can always create a birthDate instance variable of type MyDate but it would not be sensible to create a graduationDate instance variable of type BirthDate. Having a more generic type of date appears to offer more opportunities for reuse.

The MyDate class needs instance variables for year, month, and day, and needs a constructor and a toString method.

What data type should we use for the year, month, and day instance variables? The documentation quoted earlier gives limits on the values, and these values are small integers.

**Programming style – a skeleton for a class**

The MyDate class is like all other classes we have seen and like the vast majority of classes we will see.

- There are several instance variables. Instead of storing the year as the actual year – 1900, we’ll store the actual year. Instead of storing the month as a number between 0 and 11,
we’ll use a number between 1 and 12. We’ll store the day of the month as a number between 1 and the maximum number of days in the month.

- There is a constructor (and maybe more than one constructor.)
- There is a getter for each instance variable.
- There is a setter for each instance variable which it makes sense to be able to change. Note that you can make a good argument that dates should be immutable and thus there should be no setters.
- There is a toString method.
- There are unit tests.

**Primitive types**

Java supports many types of integers - byte, short, int, and long. They differ in the number of digits they can hold but programmers often select long, the longest of the four. It wastes a little memory but you won’t lose any digits, as long as your number is no larger than \(2^{63}-1\) and no smaller than \(-2^{63}\).

Recall that we talked about bits earlier.

In a computer, a bit is the smallest unit of memory. A bit has only two values, one or zero, or on and off, depending on how you look at it. Everything is stored as a collection of bits. When the collection contains two bits, it can represent four values, 00, 01, 10, and 11.

In many languages, characters (letters, punctuation marks, and numbers not used in calculations) are stored as eight bits (allowing 256 values), also known as a byte. Java stores its characters using the Unicode standard of 16 bits (allowing for 65536 values).

Representing numbers, a byte contains 8 bits (representing integers from -128 to 127), a short contains 16 bits (representing integers from -32768 to 32767), an int contains 32 bits (representing integers from -2147483648 to 2147483647), and a long contains 64 bits (representing integers from \(-2^{63}\) to \(2^{63} - 1\)).

If you wish to provide a long value as a literal or constant, you must follow it with the letter l. Due to possible confusion between 1 (one) and l (ell), you should use an uppercase L instead of a lowercase l. Thus you might see the following statement.

```java
long numberOfDays = 1000000000000000L;
```

But why do the names of these integer types not begin with a capital (uppercase) letter? The names of all the other classes we have seen have begun with a capital letter.

The answer is simple. There are some primitive types in Java which are not objects, and we have just found the names of four more of them. Of course, there is also a wrapper class called Long which encapsulates a long, and gives it some of the methods of other classes, including the toString method.
The phrase “encapsulates a long” means that a Long contains, as its sole instance variable, a long. As a class, though, Long supports additional useful methods, like toString. You will sometimes run across the word “wrapper” describing a class; this means we have a class encapsulating some value.

**Digression**

When you create a variable of a primitive type, you associate a location in memory with the name of the variable. When you inspect that location in memory, you will find the value of the variable.

But when you create an object and inspect the associated memory location, you find the address of another location in memory. If you go to that address, you will find the object, including its instance variables.

BlueJ shows an object's structure when you inspect an object. Right-click an object and you will see a list of all the instance variables. For those which are objects, there is another Inspect button which is enabled. For primitive instance variables, this second Inspect button is disabled.

We discussed some of this, but not in such detail, in a previous section in connection with cloning Address, Name, and String objects.

**The MyDate class – the code**

Take a shot at creating your own MyDate class before looking at mine, below. Create your unit tests as well.

```java
/**  
 * A date, including year, month, and day of month.  
 *  
 * @author rick gee  
 * @version 1 - april 2006  
 */
public class MyDate {
    // instance variables
    private long year;
    private long month;
    private long day;

    /**  
     * Constructor for objects of class MyDate  
     * @param year  
     * @param month  
     * @param day  
     */
    public MyDate(final long year, final long month, final long day) {
        this.year = year;
        this.month = month;
        this.day = day;
    }
}
```
public String toString() {
  return (new Long(year)).toString() + '/' +
      (new Long(month)).toString() + '/' +
      (new Long(day)).toString();
}

You can’t send a toString message to an instance of a primitive type; you can only send such a message to an Object. Thus, I have used the constructor of the Long class to convert a primitive type to an object, thus allowing me to use the toString method to display it.

What would happen if I used return year + '/' + month + '/' + day;

Try it, and explain what happens.

Hint: Characters are stored as bits, and so are numbers. How you interpret those bits depends on the datatype you have specified. In an expression which involves a character being added to a number, Java interprets the character as a number.

The bit pattern for a slash is 0000000000101111. That is, in hexadecimal (or base 16), 002F. You can see this at http://www.unicode.org/charts/PDF.

To interpret the table, find the symbol you wish and create its hexadecimal value by using the three-digit column heading followed by the digit at the left. (An uppercase “A” is 0041; a lowercase “a” is 0061.)

To convert from binary to hexadecimal, group the bits into groups of four (starting from the right), giving 0000 0000 0010 1111, and translate each group of four into one of the hexadecimal digits from 0 (0000) to 9 (1001) and A (1010) to F (1111).

When interpreted as a hexadecimal number, the pattern 000000000101111 is 2F, which translates into decimal as 2 * 16 + 15 * 1 or 47.. When interpreted as a character it is a slash.

Thus when you add a slash to a year you get (It is 2011 as I write this) 2011 + '/' is 2058.

This statement can be simplified a little, due to a new feature in Java called autoboxing. With this feature, long variables are automatically converted to Long variables, for example, when needed, particularly when you are concatenating longs and Strings. Thus, the following statement is also acceptable.

    return year + "/" + month + "/" + day;
For this to work, the slashes must be Strings, as they are in this statement, not chars, as they were in the previous statement.

You may wish to modify the separator. I chose to use a slash (/) but you may prefer to use a hyphen (-) or a space.

I have used the international ordering (year, month, and day) in this method.

You may wish to create two extra methods, one to return the date in the American format (month, day, year) and one to return the date in the British format (day, month, year).

These last two methods cannot both be named toString since there would be no way to distinguish between the three methods. They would all have the same name and the same parameters; their signatures (the name and the type and number of parameters) would be identical, a bad thing which would prevent your program from compiling.

Note that you could use the int datatype for year, month, and day if you wish. In that case, you will use Integer in place of Long in toString, although the alternative form of the statement will be unchanged. An int is large enough that your program won't be used when an int becomes too short.

**The MyDate class – unit tests**

Use your unit tests to see that MyDate creates and displays dates correctly. When you use its constructor, you are specifying integers, so you do not need to enclose them in quotation marks as you did for Strings.

What happens when you specify August as month 08 instead of just 8 (or September as 09), in the constructor?

Java allows you to specify integers as decimal numbers (base 10.) Such numbers are the one you use all the time – the non-negative ones are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, etc..

Java also allows you to specify integers as hexadecimal numbers (base 16) of which we just saw a few. Non-negative hexadecimal numbers begin 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, etc.

The numbers 0 through 9 are the same in the decimal system and in the hexadecimal system. I don't mean they look alike, I mean that they represent the same quantity.

Larger hexadecimal numbers are preceded by 0X. Thus the hexadecimal number A, representing the decimal value 10, is written as 0XA. 0XF is the hexadecimal equivalent of 15. 0X10 is the hexadecimal equivalent of the decimal value 16.
123 is a valid, but different, number in decimal and hexadecimal. If 123 is a hexadecimal value, enter it as 0X123.

Java also allows you to specify integers as octal numbers (base 8.) The non-negative numbers begin 0, 1, 2, 3, 4, 5, 6, 7, 10, etc. The decimal, hexadecimal, and octal numbers 0 through 7 are identical in appearance and value. Larger octal numbers are preceded by 0. Thus the octal number 10, representing the decimal value 8, is written as 010.

But what about 08? It begins with 0 but not 0X so it must be an octal number. But octal numbers go 0, 1, 2, 3, 4, 5, 6, 7, 10. So 08 is not an octal number and you receive an error when you attempt to use it. Similarly you get an error if you enter September as 09.

0123 is acceptable as an octal number, too. Its value is different from the decimal number 123 and the hexadecimal number 0X123.

When we translate 0123 as an octal number to a decimal number, we find it is equal to 1 \* 8^2 + 2 \* 8^1 + 3 \* 8^0 or 64 + 16 + 3 or 83. When we translate 0X123 to a decimal number, we find it is equal to 1 \* 16^2 + 2 \* 16^1 + 3 \* 16^0 or 256 + 32 + 3 or 291.

A mathematical joke is “There are 10 types of people in the world, those who understand binary and those who do not.”

In the toString method, do you want the first nine months of the year to appear as single digits (as happens with the example above) or as two digits? Do you want the first nine days of the month to appear as single digits or as two digits? If the answer is “two digits”, one solution is to modify toString as shown below.

```java
public String toString() {
    String result;
    result = (new Long(year)).toString();

    if (month < 10)
        result += "/0";
    else
        result += "/";
    result = result + (new Long(month)).toString();

    if (day < 10)
        result += "/0";
    else
        result += "/";
    result += (new Long(day)).toString();

    return result;
}
```
You should make the MyDate class cloneable, as we did for the Address class. This is because we need to keep dates with no possibility of changing them accidentally. Can you imagine an example where such changes would be very bad?

Seniority matters, so changing dates there could be a problem. In a different application, keeping track of when parcels arrived could be important.

**Using MyDate objects**

Create a birthDate instance variable in the Person class, and add it to the constructors for Person, Student, and Professor.

Create a getBirthDate method in the Person class, and test it. How? Since the Person class is abstract, you can’t create a unit test for it. But you can create unit tests for Professor and Student, checking that they process birthdates properly.

A reminder - Since the Person class is abstract, you can’t create a Person object and then test its methods.

While you should understand the sentiment of that statement, the statement is incorrect. Because a Student object is a Person object as well, whenever you create a Student object, you actually create an object which is both a Person object and a Student object.

What the reminder meant to say is that you cannot directly call the constructor of the Person class. You call it only from the constructor of its subclasses.

My getBirthDate method follows.

```java
/**
   * @return birthDate as a MyDate
   */
public MyDate getBirthDate() {
    return birthDate;
}
```

Now modify Professor to contain a dateHired. This is an instance variable which Student does not have, and thus it must be an instance variable in the Professor class.

This is the first time we have had different instance variables in Student and Professor. Both these classes are derived from Person, and thus contain all the Person instance variables, but now we see that derived classes may also have their own instance variables.

We will also see that derived classes may have their own methods, perhaps with the same names in the different classes, but different behaviours. As an example, getPhoneNumber may return the home phone number for a student but an office phone number for a professor.

How do you test that the birth date and hired date are handled correctly?
On the Object Workbench, create two MyDate objects, one for birth date and one for date hired. Create a Student object, using the birth date object you just created. Create a second Student object, specifying null for the birth date. Does toString display the birthdates properly?

Create a Professor object. Execute its toString method or its unit tests. While the constructor will accept a null date, it makes no sense to have a null date here. We will need to insert additional code at a later time to prevent this from happening.

Remember that it’s better to have unit tests, rather than these ad hoc tests. We won't be seeing any more ad hoc tests. I want you to become used to using unit tests.

**Simplifying the class diagram**

By the way, BlueJ may be creating a very complicated class diagram. It may be creating superfluous “uses” arrows whenever one class includes several instance variables of another class. This seems to be less of a problem with more-recent versions of BlueJ.

My diagram now has “uses” arrows from the Person, Student, and Professor classes to MyDate. The Student class does not directly use the MyDate class, so I deleted that “uses” arrow.

My diagram has “uses” arrows from the Person, Student, and Professor classes to the Address class. But Student and Professor do not directly use the Address class, so I deleted those “uses” arrows as well.

You may find that you can make the class diagram more readable by moving the classes around with your mouse. Here is my current diagram (omitting the unit tests for simplicity). You have created all your unit tests, haven’t you?
The diagonal shading indicates that I need to compile most of my classes. I’ve made some changes to them, so I can’t run them until I recompile them.

**Retirement of a professor**

Recall that I mentioned earlier that a professor at Okanagan College is expected to retire on the June 30 following his/her 65\(^{th}\) birthday.

Let’s create an instance variable for the retirement date, and a getter method. We don’t have a setter method, since we can calculate the retirement date once we are given the birth date.

When the month of birth is June or earlier, the professor retires on June 30 of the year in which he/she turns 65.

When the month of birth is July through December, the professor retires on June 30 of the following year.

Whenever we set the birth date, we calculate the retirement date; that is the only way the retirement date is calculated – it cannot be set through a public setter. Thus, we can create the following private method.

```java
private void setRetirementDate(final MyDate birthDate) {
    long retirementMonth = 6;
    long retirementDay = 30;
}```
long retirementYear = birthDate.getYear() + 65;
if (birthDate.getMonth() >= 7)
    retirementYear++;

dateRetirement = new MyDate (retirementYear, retirementMonth, retirementDay);
}

Notice some things about this method:

- It is private so that only another method within the Professor class may call it.
- We use the getters from the MyDate class to extract the year and month of the employee’s birth date.
- We use the ++ operation to increase the year when the birthday is in the second half of the year. ++ is shorthand for “increase the value of the variable specified by 1.”
- We use the constructor for MyDate to create the retirement date.

But which method calls this one?

The constructor for Professor will certainly need to call it.

So too will the setBirthDate method. But setBirthDate is inherited from the Person class. Student objects don’t have a retirement date; Professor objects do.

The best way to handle this quandary is to create another setBirthDate method in the Professor class. It will do everything the Person setBirthDate method does, and will also calculate the retirement date. Technically, the setBirthDate method in Professor overrides the setBirthDate method in Person.

public void setBirthDate(final MyDate birthDate) {
    super.setBirthDate(birthDate);
    setRetirementDate(birthDate);
}

The call to super.setBirthDate (the setBirthDate method in the parent class) ensures we do everything the Person setBirthDate method does. We do not want to cut and paste statements from that method into this one; if we did that, whenever we changed the Person setBirthDate we would need to remember to do the same for the Professor setBirthDate.

Once the processing in the setBirthDate method in the parent class is completed, we do the additional processing which the setBirthDate method in the derived class specifies.

**Retirement of a professor – unit tests**

Create two unit tests for getRetirementDate. The first is for a person whose birthday is in the first half of the year and the second is for a person with a birthday in the second half of the year. My tests follow.
Note that unit tests are very important here. The details of the retirement date are in a contract between faculty members and the college; we need to be able to confirm that the calculations are being done correctly.

```java
@Test
public void testgetRetirementDate1() {
    p.setBirthDate(new MyDate(1950, 11, 10));
    assertTrue("2016/06/30".equals(p.getRetirementDate().toString()));
}

@Test
public void testgetRetirementDate2() {
    p.setBirthDate(new MyDate(1949, 05, 10));
    assertTrue("2014/06/30".equals(p.getRetirementDate().toString()));
}
```

Many people will argue that unit tests for single-line setters and getters are a waste of time. But this test is definitely not a waste of time since we need to be 100% sure that the retirement date is correctly calculated.

Oh, the tests succeed, but there is a subtle error here.

What is the retirement date for a person whose birthday is exactly June 30? From carefully reading the contract, the answer appears to be, June 30 of the following year. Modify setRetirementDate and create a third unit tests to ensure this occurs correctly.

Here is my revised setRetirementDate method.

```java
public void setRetirementDate(final MyDate birthDate) {
    long retirementMonth = 6;
    long retirementDay = 30;
    long retirementYear = birthDate.getYear() + 65;
    if ((birthDate.getMonth() >= 7) ||
        (birthDate.getMonth() == 6) && (birthDate.getDay() == 30))
        retirementYear ++;
    retirementDate = new MyDate(retirementYear, retirementMonth, retirementDay);
}
```

Note the parentheses. Every if statement begins with if (some condition). Here, the condition consists of two parts (joined by the symbol for an or operation, ||), and one of the parts consists of two parts (joined by the symbol for an and operation, &&.)

To determine whether we need to adjust the calculated year of the retirement date, we need to determine whether the month of birth is greater than or equal to seven (a condition enclosed in parentheses) or the birthday is exactly June 30.

To test if the birthday is June 30, we need to test if the month is June (a condition enclosed in parentheses) and if the day is 30 (also enclosed in parentheses.)
To ensure there is no ambiguity, these last two conditions are combined using the `&&` operator and then the result is enclosed in parentheses. We finish by using the `||` operator to decide if either of the two conditions, the month is greater than or equal to seven or the birthday is June 30, is true.

Logical operators have a priority. In particular, `&&` is done before `||`. Thus many of the parentheses in the method above are superfluous. But I have put them there to clarify how the tests are being done.

Unfortunately, excess parentheses may make the statement harder to understand.

**Programming style – common errors**

A statement which I believe is “It is okay to make mistakes. It is good to admit them. It is best to learn from them. It is bad to repeat them.” This is by L. Bernstein from Software Engineering Notes, July 1993, and is still true today.

What are some of the common errors you have made? What are some of the common error message you have seen and what causes them to appear?

- **Spelling mistakes.** Spelling mistakes in comments are annoying. Spelling mistakes in variable names may cause your program not to compile. If the program compiles, the mistakes may cause it not to work, due to confusion between variable names.
- **Capitalization mistakes.** Class names are capitalized. Every word in a class name is capitalized. The first word in an object name begins with a lowercase letter but other words (if any) are capitalized.
- **Mismatched parentheses.** For every opening parenthesis there must be a corresponding closing parenthesis. BlueJ helps by indicating the matching opening parenthesis when you type the closing parenthesis. I find it helps to type the opening parenthesis, then the closing parenthesis, and then go back and fill in what should be between the parentheses.

![An unmatched left parenthesis creates an unresolved tension that will stay with you all day.](http://xkcd.com/859/)

- **Mismatched braces.** Missing braces is a more common problem than missing parentheses. For every opening brace there must be a corresponding closing brace. BlueJ helps by indicating the matching opening brace when you type the closing brace. I find it helps to type the opening brace, then the closing brace, and then go back and fill in what should be between the braces.
“Open brace, enter, enter, close brace, up arrow” is my mantra.

- Missing or superfluous semi-colons. A semi-colon marks the end of a statement, so don’t forget it, and don’t add unnecessary semi-colons. In many cases, unnecessary semi-colons don’t hurt; they just mark you as inexperienced or careless. Checkstyle catches this problem for you.

- Missing or superfluous commas. Commas are only used to separate items in a list, typically in a method header or in the call to a method. They also appear when you want them to appear in the output of a toString method.

- == or =. A single equals sign is used when you are assigning a value to a variable. A double equals sign is used when you are checking that two primitive variables have the same value. The double equals is also used when you are checking whether two objects both refer to the same area of memory, a rare test.

If you have installed the Checkstyle extension which I mentioned in an earlier chapter, now might be a good time to use it and check the code you have generated so far. Access it via the Tools menu.

Common problems Checkstyle finds are missing javadoc statements, problems with indentation, problems with parentheses and braces. The default style, one with which I agree, sometimes seems picky, but everything it flags detracts from the readability of your code.

**Bad dates - exceptions**

When you are entering dates, it is all too easy to enter an incorrect date. Incorrect, since you may accidentally enter June 31, or February 29 in a year which is not a leap year. We will prevent this by creating a MyDateException which can be thrown whenever we detect an incorrect date.

Like the other exceptions we have created, the MyDateException class is simple to write.

```java
/**
 * Problems with MyDate objects.
 * 
 * @author Rick
 * @version December 2010
 */
public class MyDateException extends Exception {
    /**
     * Constructor for objects of class MyDateException.
     * @param message The message to associate with the exception
     */
    public MyDateException(final String message)
    {
        super(message);
    }
}
```
To use this class properly we need a method to examine the parts of the date we provide and see if they are appropriate. Of course, the method will not detect a problem when you enter January 5 when you meant to enter February 5.

Consider the following method.

```java
/**
 * method to validate the three parts of a MyDate.
 * @param year  the year portion of the new date
 * @param month the month portion of the new date
 * @param day the day portion of the new date
 */
public static void validate(final int year, final int month, final int day)
throws MyDateException {
    if (month < 1)
        throw new MyDateException("Month " + month + " is too small.");
    if (month > 12)
        throw new MyDateException("Month " + month + " is too large.");
    if (day < 1)
        throw new MyDateException("Day " + day + " is too small.");

    // seven months have 31 days each
    int daysInMonth = 31;
    // four have 30 days
    if (month == 4 || month == 6 || month == 9 || month == 11)
        daysInMonth = 30;
    // and there is February. See Wikipedia entry on leap year for
    // description of algorithm
    if (month == 2) {
        if (year % 400 == 0)
            daysInMonth = 29;
        else if (year % 100 == 0)
            daysInMonth = 28;
        else if (year % 4 == 0)
            daysInMonth = 29;
        else
            daysInMonth = 28;
    }
    if (day > daysInMonth)
        throw new MyDateException("Day " + day + " is too large for " + year + " " + month + ".");
}
```

Like our other validation methods, this is a static method; there is only one method, shared by all MyDate objects, rather than a separate method for each MyDate object. When you think of it, the method must be static since we want to use it before we create a MyDate object.

The processing in this method follows the same pattern we have used earlier. Clean up the input if possible; in this case there is no cleanup. Then check the data for problems.
We check the easiest problems first. Is the month too large? Is the month too small? Is the day of the month too small? Note that the concepts of “too large” and “too small” are in relation to the twelve-month Gregorian calendar.

Then we have the harder problem. Is the day of the month too large? This is complicated since different months have different numbers of days, and February has different numbers of days in different years, depending on whether or not it is a leap year.

Perhaps you'll use the logic that is shown above, where we determine the number of days in each month, given the month and (in the case of February) the year.

Or perhaps you'll know that there is a class called GregorianCalendar which contains some useful methods. The getMaximum method, in particular, is very useful here. These few statements replace the last two-thirds of the method above.

```java
GregorianCalendar g = new GregorianCalendar(year, month, 1);
if (day > g.getMaximum(Calendar.DAY_OF_MONTH))
    throw new MyDateException("Day " + day +
        " is too large for " + year + " + month + ".");
```

Just remember to import java.util.GregorianCalendar.

What unit tests do you need? There are many.

- Month too large
- Month too small
- Day 0 or negative
- January and an acceptable day. I would test 0, 15, and 31.
- January and a day of 32 or more.
- February not in a leap year and an acceptable day. I would test 2013, and days of 1, 15, and 28. I would test 2000, and days of 1, 15, and 28.
- February in a leap year and an acceptable day. I would test a year of 2400, and days of 1, 15, and 29. I would test 2012 and days of 1, 15, and 29.
- February in a leap year and a day of 30 or more.
- February not in a leap year and a day of 29 or more.
- March and an acceptable day. I would test 1, 15, and 31.
- March and a day of 32 or more.
- April and an acceptable day. I would test 1, 15, and 30.
- April and a day of 31 or more.

Plus 16 other tests for the rest of the months of the year.

Note that the number of tests will illustrate a limitation of BlueJ.

If a test class has too many tests, the menu showing them will expand beyond the top and bottom of your screen. There is no way to scroll the menu up and down, nor is there a way to have the menu appear in two columns. If you wish to run a test which is not visible, you need to run all the tests.
As an alternative you can create a second test class and transfer some of the tests to it. The difficulty, albeit a minor one, is that when you move the MyDate class in your class diagram, the first test class will move too, but this second one will not.

As a third possibility, you can do as we have done with our constructor unit tests and put many assert statements within one method. You could have a dayTooLargeForMonth unit test which checks 12 months via 12 assert statements. Here is the beginning of my test.

```java
@Test
public void dayTooLargeForMonth() {
    try {
        MyDate.validate(2012, 1, 32);
        fail("2012/01/32 is not an acceptable date");
    }
    catch (MyDateException e) {
        assertEquals("Day 32 is too large for 2012 1.",
                e.getMessage());
    }
    catch (Exception e) {
        fail("Unexpected exception of type " +
                 e.getClass().getName() + " with message " +
                 e.getMessage());
    }
}
```

In this test we are looking at days that are too large for the month. I begin with January, which always contains 31 days. Since validate is a static method with MyDate, we access it by using the name of the class followed by the name of the method.

Since the day provided is too large, we expect validate to throw an exception. If the exception is not thrown, the test fails.

If the exception is thrown, we have two possibilities. Is it the correct type of exception, or is it not? If the exception is the right type of exception, we check that its message is correct. If the exception is not the correct type of exception, we display the type of the exception and its message.

Not that in this case, throwing a MyDateException is the correct behaviour and the test succeeds. Normally throwing an exception signifies a problem, but not in this unit test.

Once January is working we test February.

```java
try {
    MyDate.validate(2013, 2, 29);
    fail("2013/02/29 is not an acceptable date");
} catch (MyDateException e) {
    assertEquals("Day 29 is too large for 2013 2.",
                 e.getMessage());
}
```
catch (Exception e) {
    fail("Unexpected exception of type " +
        e.getClass().getName() + " with message " +
        e.getMessage());
}

try {
    MyDate.validate(2012, 2, 30);
    fail("2011/02/30 is not an acceptable date");
} catch (MyDateException e) {
    assertEquals("Day 30 is too large for 2012 2.",
        e.getMessage());
} catch (Exception e) {
    fail("Unexpected exception of type " +
        e.getClass().getName() + " with message " +
        e.getMessage());
}

Note that I have shown two tests for February, the first for a year which is not a leap year and the second for a year which is a leap year. Add the missing February tests, for years like 2000 (a century which is a leap year) and for 2100 (a century which is not a leap year.)

Then it is an exercise for the reader to test all the other months.

**Summary**

In this chapter we have created two more classes (MyDate and MyDateException), and tested them extensively.

Creating classes and testing them fully will be crucial to your success as a programmer, and as a designer.

But we have reached a plateau in what we can do with single-valued variables. We need to consider collections, the topic of the next chapters.
Exercises

1. The Java libraries contain a `DateFormat` class. Use it to display a date with two-digit months and days. That is, use it to eliminate the comparisons in the `toString` method of `MyDate`.

2. The Java libraries contain the `GregorianCalendar` class which we briefly considered. Explore that class more fully and use `GregorianCalendar` objects to replace `MyDate` objects.

3. Modify the `MyDate` class to display the name of the day of the week. For now, simply use the English-language names of the days of the week.

4. In a previous chapter, we talked about birders and the sightings they make. Part of a sighting is the date on which it occurred. Using either `MyDate` or `GregorianCalendar`, create a `Sighting` class which contains the species of bird, the date and time of the sighting, and a note about the sighting. This note may include location, weather, bird behaviour, number of males and females, and the number of young.

   Are you thinking that the note would best be represented by several separate instance variables? Good.

   If you use `MyDate`, you will need to add the time of day to the class, or you will need to derive a `MyDateAndTime` class from `MyDate`. You could even create a `MyTime` class.

5. Rewrite the `setRetirementDate` method to eliminate some of the parentheses, but do not eliminate the clarity.

6. Explore the Joda-Time project as an alternative to the `MyDate` class. Joda-Time allows much more complicated date manipulation as well as many different calendars, particularly non-Western calendars.
Chapter 8 – Collections, part 1

Learning objectives

By the end of the chapter you will be able to:

- Define the general term collection
- Define and describe the capabilities of the set, list, and map collections
- Compare and contrast the capabilities of set, list, and map
- Describe implementations of set, list, and map
- Use sets, lists, and maps to represent complicated data
- Define iterator
- Use an iterator and a for-each loop to process elements in a collection
- Use a while statement

Life is complicated

You may have noticed that you are taking a number of courses. You may have noticed that some of your courses have one section but others have multiple sections. You may have noticed that a professor generally teaches more than one course. You may have noticed that your studies extend over more than one semester. You may have noticed that a department at your school offers many courses. You may have noticed that a department at your school consists of many professors.

All of these remind us that the world contains objects in multiples, not as singles.

Up to now, we have been able to create two professors, by giving each Professor object a separate name. How do we accommodate varying numbers of professors in a department? Behind that question are two different situations. First, different departments have different numbers of professors. Second, the same department may have a different number of professors at different times as professors retire, resign, or go on leave, and new professors are hired.

Similarly, how do we accommodate students taking varying numbers of courses? Some students may elect to take only three courses at one time. Others may take four, five, or more.

We need to look at data structures, commonly called collections. A collection of X is an object, containing within itself a number of different objects of type X.

Thus you can talk (in other models) about a collection of automobiles, a collection of houses, a collection of vacation destinations, or a collection of occupations. We can even have a collection
of collections. The college model we are examining contains many collections, many of which will be in the College class.

Collections the college model contains

There are many possible ways to model the collections a college contains. What follows is my opinion. For each collection, I’ve included a brief discussion of why I believe that collection is appropriate.

A college contains collections of employees; it is better is there is a different collection for each type of employee. The only employees we have implemented so far are professors; all the professors at the college form a collection. This collection of professors (in the College class) will be the only collection containing Professor objects. All other collections “of professors” will contain only employee identifiers.

If you have an identifier, you can always find the object associated with that identifier, provided you have the appropriate getter method for the collection. That is, you will have a getter which accepts an identifier and returns the object which has that identifier.

A note regarding the decision to have many collections of employees: At my college, there are at least four kinds of employees, three of which are based on the unions and the locals of the unions to which they belong. The fourth kind is the administrators who are not in a union but are in an association, a different type of organization. Each kind of employee corresponds to a different collection.

Thinking like this comes from courses in systems analysis and design. Those courses teach you the questions you should ask before you start to code. I have done analysis (to understand the problem) and design (to find a good solution) in order to decide what I am asking you to implement.

A college contains a collection of students. When you register as a student, you do not register as a student in a department or as a student in a course, you register as a student of the college. This collection of students (in the College class) will be the only collection containing Student objects. All other collections “of students” will contain only student identifiers.

A college contains a collection of departments. When we implement this collection in an exercise, we’ll see that it is actually a collection of Department objects. This collection of departments (in the College class) will be the only collection containing Department objects. All other collections (faculty, for example. We call these unit portfolios at my school.) “of departments” will contain only department identifiers.

Each department contains a collection of employees. When we implement this collection in an exercise, we’ll see that it is best to implement a department as a collection of employee numbers, rather than as a collection of Employee (or Person) objects. When we need Employee (or Person) objects, we’ll be able to retrieve them, from the college’s collection of employees, as long as we know the employee identifier.
A college contains a collection of courses. While a course may appear to be offered by a department, it is actually offered by the department on behalf of the college. When we implement this collection, we’ll see that it is actually a collection of Course objects. This collection of courses (in the College class) will be the only collection containing Course objects. All other collections “of courses” will contain only course identifiers.

Next we have an interesting collection, the collection of sections. One possibility is that a course contains a collection of sections. While that may be reasonable, it will lead to problems later when we attempt to model the associations between students and sections. There are several such associations; one which represents the students in a section, one which represents the sections in which a student is currently enrolled, and one which represents the sections in which a student was enrolled in the past.

Rather than have the course contain a collection of sections, a better design is to have the college contain a collection of sections. When we implement this collection, we’ll see that it is actually a collection of Section objects. This collection of sections (in the College class) will be the only collection containing Section objects. I'll probably call this collection a Semester or a Term. All other collections “of sections” will contain only section identifiers.

Each section contains a collection of students. When we implement this collection, we’ll see that it is best to implement a collection of student identifiers, rather than a collection of Student objects.

Each student contains two collections of sections, one consisting of sections in which the student is currently enrolled, and one consisting of sections in which the student was enrolled in the past. For past sections, we remember the marks the student earned at that time. When we implement these collections, we’ll see that it is best to implement a collection of section identifiers, rather than a collection of Section objects.

Each section contains a collection of its meeting times. When we implement this collection, we’ll see that it is actually a collection of Meeting objects. This collection of meetings (in the Section class) will be the only collection containing Meeting objects. All other collections “of meetings” will contain only meeting identifiers.

Why will some collections contain objects and some the identifiers of objects?

The prime reason is that we wish to have only one copy of an object. That is, there will be only one Professor object representing each professor at the college. There will be only one Student object representing each student at the college. By having only one object corresponding to each item in the real world, we will eliminate duplication of data. We will also ensure that when an object is updated, the update needs to happen in only one place.

Whenever any method needs information about a student, for example, it will need the student identifier and can use that identifier to ask (by calling a method in) the College class to search its collections and return the appropriate data.
These ideas are behind the practice of “database normalization.” Any course in database management will explain normalization in more detail.

Now that we have an idea of what collections our model needs, let’s see what Java provides to implement the collections.

**The Java Collections Framework**

Collections may be categorized in many ways.

- Is the size of the collection fixed, or may it vary? The number of months in the Gregorian year is fixed at 12, but the number of courses a student takes may vary from student to student, and from semester to semester.
- Does the collection allow duplicate values or does it not? A rack of Scrabble® tiles can contain duplicate tiles but a hand of playing cards drawn from a 52-card deck cannot.
- To check that an item is in the collection, how many other items do we need to check? Is the collection designed to allow speedy searching? This matters when we are searching through large collections.
- Are the items in the collection sorted in some order, or are they unsorted? If they are in one order, can they be placed in a different order?
- Does an item in the collection consist of a single value or is the item a pair, consisting of a key and a value? If you are creating a list of the words used in a document, the collection need only contain the words. But when you are creating a dictionary, the collection needs to contain both the word and a definition. Many words have multiple definitions, so each item of the collection may itself contain a collection. Think also of the index of this book. There are very few words in the index that appear on only one page. One such word, chosen for no particular reason, is hippopotamus.

By asking questions such as these, we can determine the type of collection we should use in a particular circumstance.

Sets and lists are collections of objects. The difference between them is that a set does not allow duplicates, but a list does. (Another difference is that a set is usually unordered, while a list has an order, the order in which elements are added to the list. There are different implementations of set, some of which may create an order. There are different implementations of list, which may allow a different order. We’ll see these implementations in examples later.)

Consider the students in a section of a course or the sections of a course a professor teaches. These would be represented by sets, since students are unique within a section and sections within a course are unique. The uniqueness is enforced by having unique identifiers.

Consider the coins in your pocket, or the list of items you need to pick up from the grocery store on your way home. These are lists as you may have two coins of the same denomination or two loaves of bread on your shopping list.
A map uses a key/value metaphor to store information. Think of a table with two columns and many rows. The first column represents the key; the second column represents the value associated with the key. For example, the key might be a student identifier and the value might be a Student object. Think again of the list mentioned above, the things you must pick up from the grocery store on your way home. The key may be “loaf of bread” and the value may be two. This last statement shows that choosing the type of collection is not simple.

A multimap is a map in which the values are themselves collections. For example, the key might be a course identifier and the value might be the professors teaching the course. A multimap is also called a dictionary, since a word (the key) may have many meanings (the values).

Rather than reinventing these collections, we will use the Java Collections Framework in this textbook.

However, there are other collections available. One interesting collection (of collections!) is available at http://code. This is a project that Google has developed, containing many of the core libraries used by Google projects. Guava contains many libraries, including the Collections library, which may be available at http://code.google.com/p/google-collections/ if the previous URL is broken. We’ll use some other Google libraries in a later chapter.

The Java Collections Framework provides collections via interfaces. An interface describes the characteristics of a collection without providing implementation details. We have seen this aspect of interfaces, but not with collections, when we explored Cloneable. The Java Collections Framework also provides classes which implement each interface in several different ways.

One collection type, and hence interface, is Set. Implementations of the interface include HashSet and TreeSet. HashSet is designed to allow speedy retrieval and insertion. TreeSet provides a sorting mechanism. But both allow an element to occur only once in the collection.

Another collection type (and interface) is List. Implementations of the interface include ArrayList and Vector.

A third collection type (and interface) is Map. An implementation of the interface is HashMap.

We will look at some of the aspects of collections in the current and following chapters. More details on the Java Collections Framework are available in the Java tutorial, at http://download.oracle.com/javase/tutorial/collections/index.

We will begin our discussion of collections by considering the simplest collection, a set, a collection of data items with no duplicates and usually no order.

A set

In mathematical terms, a set is an unordered collection of items, with no duplicates. \{1, 2, 3\} is a set, the same set as \{3, 1, 2\}. \{1, 2, 4, 5, 6, 4\} is not a set because of the duplication of the number 4.
There is a collection called a multiset (or a bag) which allows multiple occurrences of an item, but remains unordered. There is no Java class for a multiset available as part of the Java Collections Framework.

The Google project mentioned earlier does contain a multiset.

As noted earlier, we can use sets in our model of the college. After all, a person will not be registered twice in the same section of a course, nor will a professor teach the same section twice.

At my college, a student may be registered in one lecture section, one lab section, and one seminar section of the same course in a semester. This is very common and our model supports it since each section, regardless of its type, will have its own identifier.

At my college a student may be registered in two lecture sections of the same course at the same time only when the course is a Special Topics course and the two sections focus on different topics. Even in this special case, a student is not registered twice in the same section.

Does the order in which we store the courses in which a student is registered matter? Does the order in which we store the professors in a department matter?

In general, the order doesn’t matter. When it does, we will place the data structure in the correct order before processing it. What matters is that there are no duplicates in the collection.

**A collection of professors**

Let’s consider how to implement a collection of professors, all of whom are employees of the college. Recall that the collection will be an instance variable of the College class.

The collection of professors will be represented as a set since there are no duplicate professors. This is because every professor has a unique employee number or identifier. That ensures there are no duplicates.

We will assume no ordering on the elements.

More correctly, there are many possible orderings of professors, but none stands out as the predominant one. Professors might be ordered by hiring date (and this would be useful when producing seniority lists.) They might be ordered by employee number. They might be ordered by name (and this would be useful when producing phone directories.) They might be ordered by name within department.

Given these possibilities, we will not impose an order until the time we need to process the collection. Then, should it be necessary, we will impose an order.
To implement the collection, we need to import an interface into the College class. Use this import statement

```java
import java.util.Set;
```

Place that statement at line 1 of the class. BlueJ will use javadoc to create the documentation for a class, but one of javadoc’s restrictions is that the comments describing the class must appear immediately before the class header, with no blank lines between the */ and the class header. Thus, any import statements must appear at the beginning of the file describing the class. BlueJ politely leaves a blank line at the beginning of the file when it creates the skeleton of a class for us.

If you look at the documentation describing java.util.Set, you’ll find, as expected, that it is an interface. In the list of classes and interfaces (lower left frame) interfaces are italicized. In the main frame, the heading is Interface Set.

Recall that an interface describes the capabilities of a data structure, but it does not describe how these capabilities are implemented. The documentation lists All Known Implementing Classes. These are the classes which implement the capabilities of the Set interface. Of the several choices presented, we’ll select the HashSet. Why we select HashSet is discussed below.

**HashSet**

A HashSet is a very efficient way of implementing a set. That is, when we need to retrieve an element from the set, that retrieval is very quick. When we need to place an element in the set, that operation too is very quick. We pay for speed by using more memory for the implementation than we may use in other implementations.

The word “hash” refers to how the set keeps track of its members, but one of the beauties of object-oriented programming is that we don’t need to know the details of how a HashSet works; we just need to know how to use it. That is, we need to know how to import it into our project, what methods it supports, and how to use those methods. Such information is described in the online documentation.

Time complexity is a way of measuring how quickly a task is completed. Space complexity is a way of measuring how much space a task requires. Often we trade one for the other since we can't have great performance in both. Fast (less time) and cheap (less memory) are usually incompatible.

The food called hash is a mix of a variety of different ingredients, often chopped up. Some people say it looks like pet food. Some hash recipes are available at http://homecooking.about.com/library/archive/blhash.htm.

The idea behind the computer technique called hashing is similar. Take the value of the Object and mix it up some way, producing a single number, the hashcode. That number is an indication of the contents of the Object. Two Objects with identical values have the same hashcode.
An interesting application of hashing is [www.tinyurl.com].

Yes, there are other meanings for the word “hash.”

Java is actually a very simple language. Most of its complexity and power comes from its many libraries. When you wish to use a class from one of the libraries (or packages), you must import that library or package. At least, you must import the portion of it which you need.

The Java documentation tells us that HashSet is in the java.util.HashSet package, so we need to import that package as well.

Now College contains two import statements.

    import java.util.Set;
    import java.util.HashSet;

**Generic collections**

Java collections are *generic*. That is, the collection, and its capabilities, is independent of the type of data it contains. This is a very powerful feature and has only recently come to Java.

While we could create a collection of Object, and thus place any type of object in it, that is not a good idea since we might have to check the type of each object as we retrieved it. It is much better to identify the type of element in the collection as precisely as we can when we first declare the collection. By writing HashSet<Professor> when we declare a variable we are saying we will be using a HashSet all of whose elements are Professor objects.

We are also asking the Java compiler and interpreter to let us know should we ever attempt to place some other type of object in the collection. The compiler will display an error message when we attempt to compile a program including statements to add an object of an incorrect type.

At least it will as long as we have gone through the BlueJ menu choices Tools, Preferences, Miscellaneous, and checked “Show compiler warnings when unsafe collections are used.”
Note that this will also cause the compiler to tell us whenever we declare a variable as just a `HashSet` without specifying the type of its elements.

We need to name instance variable for the collection of professors at the college. Let’s use the name `theProfessors`.

```java
private Set<Professor> theProfessors;
```

By declaring `theProfessors` as a `Set`, we are describing its general structure and capabilities. We have not yet indicated the specific implementation of `Set` we would like to use. We do that in the constructor for the class.

```java
theProfessors = new HashSet<Professor>();
```

This statement creates a collection which is initially empty. The documentation tells us that while the collection is empty, there is room for 16 elements in the collection. That size will expand when the collection is 75% full. When we add elements to the collection, all of whose elements must be instances of the `Professor` class.

We do not need to use the name `this.theProfessors` as there is no confusion possible between an instance variable and a parameter since there is no parameter with the same name. But there is nothing wrong with using it.
**Programming style**

When declaring variables which are collections, declare the type as an interface, and then use a specific implementation when the variable is instantiated.

How do you name a collection?

Some people will often name a collection after the class of object it contains, changing the first letter to lowercase and adding an “s” to the end of the class. Thus a collection of Student objects would be named students. A collection of Course objects would be named courses. A collection of Meeting objects would be named meetings.

A different naming convention, the one I’m using here, prefixes the name of the class with the word “the” and adds an “s” to the end of the class name. This leads to variables named theProfessors, theStudents, and theMeetings.

**HashSet<Professor>**

Note the details in the documentation that says that any Object placed in a HashSet must implement the equals and hashCode methods.

hashCode is used to determine the location in the set which the Object will occupy, and equals is used to detect collisions, when two Objects have the same hashcode.

I have said that a set is unordered. That is correct, but the HashSet implementation needs to determine where in the set to place a particular object. It uses the hashcode of the object to do so.

Note that the order of elements using the hashcode may have no connection to the order you might expect. To see this, suppose the hashcode is the number formed from digits two, four, and seven of the professor identifier.

If you have professors whose identifiers are 193456789, 487654271, and 777234567, their hashcodes are 947, 862, and 725.

They will be placed in the set in the order of the hashcodes, so 777234567 comes first, and 193456789 comes last, exactly the reverse of the order you might have expected.

A different way of calculating the hashcode would result in a different order.

Collisions occur. To prevent the collisions having a degrading effect, the HashSet should contain some unused space. That is why a HashSet will expand when it is more than 75% full.
The equals method

The equals method provides a way to signal when two objects have identical content. In the case of people, we could define equality to mean that the names are identical.

Of course, this definition wouldn’t work in the John Smith Society. Some stories say this society is limited to descendants of John Smith, an American pioneer; others say it is limited to members named John Smith.

In the latter case, the definition of equality using names we have proposed won’t be appropriate because it would say everyone is identical. In that case, perhaps we should use an alternative form of equals, one which examines the identifier. We show that possibility later in this chapter.

Since a professor is a person, we can place the equals method in Person and it will be available to professors, students, and any other type of person we create. If we compare an object of one class to an object of another class (using instanceof) the result is false. If we compare two objects of the same class then we compare the names.

```java
/*
 * test for Person object equality using names.
 * @param o The Object to be compared against the current Object
 * @return true if both objects are Person objects and the names are the same,
 * false otherwise
 */
public boolean equals(Object o) {
    boolean result = false;
    if (o instanceof Person) {
        Person p = (Person) o;
        result = this.familyName.equals(p.getLastName()) &&
                this.givenName.equals(p.getFirstName()) &&
                this.otherName.equals(p.getMiddleName());
    }
    return result;
}
```

This method overrides the equals method in Object. That's the reason the parameter is an Object, to match the signature of equals in Object.

Some sources recommend a longer but faster version of an equals method, checking for comparisons to null (in which case the answer is false) and itself (in which case the answer is true). That method looks like this.

```java
public boolean equals(Object o) {
    boolean result = false;
    if (o == null)
        result = false;
    if (o == this)
        result = true;
```
if (o instanceof Person) {
    Person p = (Person) o;
    result = this.familyName.equals(p.getLastName()) &&
            this.givenName.equals(p.getFirstName()) &&
            this.otherName.equals(p.getMiddleName());
}
return result;
}

That version uses only one return statement per method. If you allow more than one return statement per method then here is a third version of equals.

public boolean equals(Object o) {
    if (o == null)
        return false;
    if (o == this)
        return true;
    if (o instanceof Person) {
        Person p = (Person) o;
        return this.familyName.equals(p.getLastName()) &&
                this.givenName.equals(p.getFirstName()) &&
                this.otherName.equals(p.getMiddleName());
    }
    return false;
}

If we assume we have two Person objects (p1 and p2), to check them for equality we use the expression p1.equals(p2). We have seen a similar use of the equals method while comparing Strings in the unit tests.

This method examines two Objects, one provided via the reserved word this (in the expression above that object will be p1) and the other provided as a parameter (in the expression above that will be p2). This method returns a boolean whose value is true when the objects are both Person objects with the same name and false when either the objects are not both Person objects or they are both Person objects, but some part of the names are different.

We cannot compare apples and oranges, and we know that p1 is a Person object, so the method first checks that the Object (p2) provided for comparison is a Person object. If not, the comparison fails, and the method returns the value false. If p2 is a Person object, we cast it to a Person object so we can extract its instance variables; the objects are equal when the given names, other names, and family names are identical.

Alternatively, as noted above, we could check that the identifiers are equal. This is probably safer since schools have unique identifiers for Persons, both students and professors.

What happens when a professor takes some courses, thus becoming a student?

At my college, there is no problem since the identifiers are the same structure. Several recent graduates from a degree program were working at the college while completing the degree. Their
classification at the college was support staff, a type of person we are not modelling here. But we could.

Thus a better equals method is:

```java
/*
 * test for Person equality
 * @return true if both objects are Person objects and the identifiers are the same, false otherwise
 */
public boolean equals(Object o) {
    if (o == null)
        return false;
    if (o == this)
        return true;
    if (o instanceof Person) {
        Person p = (Person) o;
        return this.identifier.equals(p.getIdentifier());
    }
    return false;
}
```

Recall our discussion of George Boole earlier. boolean variables have the value true or false.

**The hashCode method**

Recall the details in the documentation that mention every object added to a HashSet must have a hashCode method.

This method manipulates the contents of the Object to determine an integer which characterizes the Object. It is possible that two Objects may hash to the same number. In that case, we will have a collision (That’s the technical term.) when adding the Objects to a HashSet (or a HashMap.) Collisions are dealt with by the collection in one of several possible ways, but collisions are an advanced topic which we will not pursue here.

Every Object has a hashcode. Since the identifier is an Object, we will simply use its hashcode as the hashcode representing the Person object rather than inventing our own hashing algorithm.

```java
/*
 * @return hash code for the Person object
 */
public int hashCode() {
    return this.identifier.hashCode();
}
```

More generally, we could use an alternative hashCode method.

```java
/*
 * @return hash code for the Person object
 */
public int hashCode() {
```
How many elements are in a set?

Now we have a collection in which we can place Professor objects (but we don’t know how to place them there yet.) We may find it useful to know how many objects are in the set at any time. Yes, at the moment there are none.

How many elements are in the collection is independent of the way the collection is implemented. Thus we would expect to look in the Set documentation (rather than the HashSet documentation) and find a size method which will give us that information.

Actually, HashSet does include a size method of its own.

```java
/**
 * How many professors are there?
 * @return the number of Professor objects teaching at the college
 */
public int getProfessorCount() {
    return theProfessors.size();
}
```

There is no setter for the size. The size is automatically incremented when an element is added and decremented when an element is removed. We saw a getter with no corresponding setter earlier when we discussed a professor’s retirement date.

Create a test for the collection now, even though the collection contains no professors.

```java
@Test
public void testGetProfessorCountZero() {
    assertEquals(0, c.getProfessorCount());
}
```

What are the elements in the set?

I realize that we don’t have any way of adding professors yet, but I find it a good practice, or style, to be able to display a collection, even when the collection is empty. But first, we need a way to visit all the elements in a collection.

Traversing a collection with a for loop

Recent versions of Java provide a very useful statement, the for-each loop.

```java
for (Professor p : theProfessors) {
    // do something with the contents of each p
}
```
In English, the statement means visit all the elements of the collection in some order; we have no control over the order in which these visits take place. As you visit an object, copy it into a temporary variable whose name is the one we have provided, \( p \). Then do something with \( p \).

Note that you cannot update or delete \( p \) this way, but you can display it or count it or examine its contents. The name \( p \) is just an abbreviation of Professor. I use single- or double-letter names to represent temporary variables which are used only within a small piece of code, or which have very limited scope, like this for-each loop.

Note the opening and closing braces. If you omit them, the code will not compile even when there is only one statement which you execute for each element of the collection.

Now that we can visit the elements in the collection, we can list the elements in the collection. Here is a method to create a String which contains all the professors in the collection, one per line.

```java
public String professorList() {
    String result = "Professors :";
    for (Professor p : theProfessors) {
        result += ‘\n’ + p.toString();
    }
    return result;
}
```

Since the collection we are listing is an instance variable of the College class, professorList is a method in the College class.

**Adding a professor**

Java libraries have generally been well-designed (but see the earlier discussion about dates.) We will find, by looking further at the online documentation, that collections support an add method which will, obviously, add an object to the collection. For a Set, the addition will take place only when the element is not already present, since sets do not allow duplicates.

```java
/**
 * Add a Professor as a new employee.
 * @param the Professor object to be added
 */
public void addProfessor(Professor p) {
    theProfessors.add(p);
}
```

Create a unit test which adds some professors to the collection and then use professorList to display the list of professors.

You may add professors within your unit tests in several ways. One, admittedly clunky solution, is
assertTrue(c.professorList(), false);

where c is a variable of type College.

This is a form of the assert statement we haven’t seen before. In it we provide a message (the first parameter) which will appear should the assertion (the second parameter) be false, which it always does in this case.

When JUnit tells us the test has failed, by placing an X beside the name of the test (and displaying the dreaded red line), you should click the name of the test and see its more-detailed explanation. The explanation is the list of professors working at the college.

But there are other solutions. One is to replace the call to assertTrue with System.out.println(c.professorList());

As we have seen in many places so far, there is often more than one way to produce a result. Which is better (or best) is sometimes a judgement call, a matter of personal preference.

System is a package which the Java libraries provide us. It is always imported without us providing an import statement. Within that package is an object named out. This object exists to display messages on the standard output device, the screen or monitor you are using, in BlueJ’s case the upper portion of the Terminal window. The out object supports a method called println which displays the value we pass to it, here the representation of a professor. The name println is an abbreviation for “print line” which itself is an abbreviation for “print some text and then move down a line.”

Compile CollegeTest. Before testing professorList, select View, Show Terminal from BlueJ’s main menu. A window appears in which you will be able to see the output of the test. Then test professorList.

In what order are the professors listed?

Run the test again. The order may change as you run and rerun the unit test. Is there a problem with our program?

No, the order in which elements of a set are listed does not matter so it may change from test to test. To clarify a statement I made earlier, the hashcode is used to help determine the order in the set; it does not determine the order. The order does matter, of course, when we want the professors listed in alphabetical order.

When you attempt to add the same professor a second time, the addition is silently rejected. That is, you do not receive an error message, but the professor is not added. After all, the definition of a set says there will only be one occurrence of each value in the set.

Actually, when you check the documentation closely, you will see the add method returns true when the addition is successful. It returns false when the addition is not successful. The
addProfessor method we have created discards the value returned. Hence you can say it silently rejects duplicates.

It is usually not a good idea to ignore values returned by a method, but it seems to be acceptable here. If you wish to pay attention to the value returned, then addProfessor should return a boolean, and addProfessor should become.

```java
public boolean addProfessor(Professor p) {
    return theProfessors.add(p);
}
```

When should you add a Professor object to the collection?

I would argue that it should be done when the object is created, as the last statement in your constructor for Professor. Modify the constructor to make it so. Modify the unit test as well.

Do your unit tests still work? If not, fix the problem.

**Cloning a professor**

Note that we have not cloned the Professor object when we added it to the collection. We cloned addresses. Why should we not clone professors?

Actually, we should clone it, but we don’t yet have a clone method in the Professor class. Let’s remedy that right now.

```java
/**
 * clone.
 * @return a copy of the Professor
 */
public final Professor clone() {
    return new Professor(identifier, firstName, middleName, lastName, preferredName, fullName, addr, birthDate, dateHired);
} // end clone()
```

Note that as we create more instance variables in the Professor class, we will need to modify the clone method to ensure all the instance variables are cloned. Some of these instance variables may be primitive datatypes, some may be collections.

Compare this clone method to the one we created in the Address class. The signatures are quite different. In Address, the signature is public final Object clone() and here it is public final Professor clone(). What is the difference between the two and why does that difference exist? Does the difference matter?

Now clone the Professor object when you add it to the collection.
Removing a professor

By looking at the online documentation we see that many collections support a remove method to remove an object from the collection. This method assumes that we know (have a reference to) the object we wish to remove. But, since we cloned the object, we do not have a reference to the object. However, we do know its identifier.

Thus, to remove a professor, we need to look through all the elements in the collection until we find the correct one, the one with the identifier we are seeking, and then we remove it.

If we were developing a more sophisticated model, we would probably not actually remove the professor. We would probably mark her as inactive.

There are at least two types of inactive; she could have resigned and gone elsewhere, or she could have retired. If we wish to mark a professor as inactive, removeProfessor would not actually remove the professor. It would set a flag (an instance variable with a limited set of special values, possibly true or false, possibly an enum type. The enum type is not covered in this book. Look at http://download.oracle.com/javase/tutorial/java/javaOO/enum.html for details.) to show that the professor is inactive.

The for-each statement is very handy when we wish to just display the elements in a collection. But we wish to delete an element from the collection and the for-each statement does not allow that. We need an iterator.

Traversing a collection with an iterator and a while loop

An iterator is an object which allows us to visit all of the objects in a collection. As we visit the objects we may change them (including delete them) should we wish to do so. An iterator is a concept; an Iterator is an interface for the iterator concept.

Every Iterator object has two methods.

- hasNext returns the value true when there are more unvisited items in the collection and returns the value false once you have visited all the items.
- next returns the next unvisited item in the collection, should there be one. Before using the next method, you should check the result of the hasNext method to be sure there is an unvisited item.

Depending on how the collection is implemented, an iterator may be implemented in many different ways. But we won’t worry about its details! It is sufficient to know that we import the Iterator interface from java.util.Iterator, associate it with a collection, and then use the hasNext and next methods. For more details than we have seen here, see the Java documentation on Iterator.

We have already created a HashSet instance variable theProfessors. To allow us to create its iterator, we place another import statement in the College class

import java.util.Iterator;
In any method in Professor which requires an iterator, place a statement to create a variable which is an iterator over Professor objects.

```java
Iterator<Professor> it = theProfessors.iterator();
```

When you are using an iterator, you often need a while loop as well.

A while loop (or while statement) uses a condition to determine whether a statement (or a block of statements, enclosed in braces) should be executed and then whether the statement (or statements) should be repeated.

While the condition remains true, the loop continues processing data. This means that something inside the loop must change the value of the condition. Otherwise you have created what is called an infinite loop. We know no way to get out of an infinite loop yet.

```
By the way, Apple Computer has its headquarters located at 1 Infinite Loop, Cupertino, California.
```

Here is a version of professorList which uses an iterator and a while loop.

```java
public String professorList() {
    Iterator<Professor> it = theProfessors.iterator();
    String result = "Professors";
    while (it.hasNext()) {
        result = result + 
        "n" + it.next().toString();
    }
    return result;
}
```

Recall our discussions of George Boole earlier. boolean variables have the value true or false. The hasNext method returns a boolean value. When it returns true there are more elements in the HashSet beyond those, if any, we have processed. If it is false, there are no more elements to process.

Recall our discussions of George Boole earlier. boolean variables have the value true or false. The hasNext method returns a boolean value. When it returns true there are more elements in the HashSet beyond those, if any, we have processed. If it is false, there are no more elements to process.

In English, the while loop does the following.

- **Step 1** – check whether there are more elements in the HashSet.
- **Step 2** – If so, process the element which is available, and repeat Step 1.
- **Step 3** – If there are no more elements, the loop is completed, so resume execution of the method with the statement following the closing brace at the end of the while.

In a similar manner we use an iterator and a while loop to create a removeProfessor method.

```java
/**
 * remove a Professor from the collection.
 * @param the identifier of the Professor object to
 * be removed from the collection
 */
```
public void removeProfessor(final String target) {
    Iterator<Professor> it = theProfessors.iterator();
    while (it.hasNext()) {
        Professor p = it.next();
        if (target.equals(p.getIdentifier())){
            theProfessors.remove(p);
            break;
        }
    }
}

The iterator provides the elements in the collection one element at a time. We compare the identifier (the employee number) of the element to the target, the identifier of the object we wish to delete. Once we have found the appropriate Professor object, we use the collection’s remove method to remove the object, and then we use a break statement to terminate the loop immediately. Since there is only one professor with the specified identifier, we need look no further.

The break statement is equivalent to “terminate the loop immediately!”

Some would say that a break statement is a harsh way to terminate a loop. Is this a better way?

```java
public void removeProfessor(final String target) {
    Iterator<Professor> it = theProfessors.iterator();
    boolean notFinished = it.hasNext();
    while (notFinished) {
        Professor p = it.next();
        if (target.equals(p.getIdentifier())){
            theProfessors.remove(p);
            notFinished = false;
        } else
            notFinished = it.hasNext();
    }
}
```

Does the method work when you attempt to remove a Professor object which does not exist?

**Exceptions**

If you attempt to retrieve a Professor object which does not exist, or you try to delete a Professor object which does not exist, how do you indicate this?

Returning a null object if the Professor object you tried to retrieve does not exist is a possibility. Throwing an exception is perhaps a more common method.
The remove method we see above could be modified to return a boolean, whose value would be true if the removal was successful, or false if it was not. But again, throwing an exception would be a more common method.

Make it so.

**Another collection of professors – the department**

A college contains a collection of departments. Each department contains a collection of professors. When we implement a department, it is a collection of professor identifiers, rather than a collection of Professor objects.

Here is a portion of the code to implement a department.

```java
import java.util.Set;
import java.util.HashSet;
import java.util.Iterator;

public class Department {
    // instance variables
    private String departmentName;
    private String chairPerson;
    private Set<String> theMembers;

    public Department(final String departmentName, final String chairPerson) {
        this.departmentName = new String(departmentName);
        this.chairPerson = new String(chairPerson);
        theMembers = new HashSet<String>();
    }

    public String toString() {
        String result = "Name: " + departmentName + " Chair: " + chairPerson;
        return result;
    }

    public void addMember(final String identifier) {
        theMembers.add(new String(identifier));
    }

    public void removeMember(final String identifier) {
        Iterator<String> it = theMembers.iterator();
        while (it.hasNext()) {
            String s = it.next();
            if (identifier.equals(s)) {
                theMembers.remove(s);
                break;
            }
        }
    }
}
```
Listing all the professors

Now let’s consider the problem of listing all the members in the department. It is easy to list just the employee identifiers.

```java
public String departmentList() {
    String result = "Department:" + departmentName;
    for (String id : theMembers) {
        result += '\n' + id;
    }
    return result;
}
```

But how do we list the names of the employees?

The only objects which know the names of the employees are the elements of the `theProfessors` collection within the `College` class. How do we access that collection?

**The Singleton pattern**

There are several solutions to accessing a collection, but the best is to recognize that we will probably be modelling only one college at a time. Thus it may be possible to determine the `College` object which we are using by simply asking the `College` class itself. Consider these modifications to the `College` class.

First, create a `College` object.

```java
private static College theCollege = null;
```

It is `private` so that it is accessible only through a method, which we will create in a moment. It is `static` so that there is only one such object. It is `null` as it is not associated with any data yet.

Second, make the constructor `private`. It will have no parameters, so you will need to use the setters to specify its name, phone number, etc.

```java
private College() {
    this.name = null;
    this.phoneNumber = null;
    this.homePage = null;
    this.streetAddress = null;
    this.mailingAddress = null;
    theProfessors = new HashSet<Professor>();
    // any other collections
}
```

Did you import all the correct collection classes?
Third, create a getInstance method which returns theCollege. If that variable does not yet exist, the method should create it and then return it.

```java
public static synchronized College getInstance() {
    if (theCollege == null)
        theCollege = new College();
    return theCollege;
}
```

This method is the access point for theCollege instance variable. Whenever your program needs information about the college, it first uses College.getInstance().

Your program will contain many statements that say College c = College.getInstance(); but in every case the object c will refer to the same College object. That is the one place which contains a collection of the objects in the model, so we can be sure that any updates happen in only one place.

The word synchronized is used to ensure that only one copy of the method is executing at one time. This is important to ensure that there is truly only one copy of the object.

Consider the situation where two different parts of the program call (or invoke) the method at the same time. One must go first. It is given a small amount of time, so it executes the first statement. Then the second invocation of the method gets its turn and it is allocated a slightly larger amount of time, sufficient to complete the first two statements of the methods. Then the first invocation gets its turn and executes a statement.

Both invocations have now executed the constructor and thus there are two copies of the College object.

By declaring the method synchronized, one invocation of the method may not begin until another has finished. Thus the first invocation creates the College object, and the second invocation returns the College object which was already created.

This technique is one which has been used successfully by many writers in many circumstances. The word pattern is used to describe such generally-useful techniques. Patterns will be covered in more detail in a later chapter. This particular pattern is called Singleton.

**Listing all the professors, continued**

Now that we have access to the College object, we can use it to determine the name of an employee, given the employee’s identifier. This method, placed in the College class, will do it.

```java
public String getProfessorName(final String identifier) {
    String result = "";
    for(Professor p : theProfessors) {
        if (identifier.equals(p.getIdentifier())) {
            result = p.getFullName();
        }
    }
    return result;
}
```
When we find the professor with the correct employee number we return the name. If we do not find the professor, we return an empty String.

Would it be better to throw an exception?

Now we have all the pieces to produce a list of the members of a department, showing the names of all the members.

```java
public String departmentList() {
    College c = College.getInstance();
    String result = "Department:" + departmentName;
    for (String id : theMembers) {
        result = result + 
        c.getProfessorName(id);
    }
    return result;
}
```

The unit test involves creating a college, creating some professors who work at the college, creating a department, placing some professors in the department, and then listing the members of the department.

```java
@Test
public void testDepartmentList() {
    // sample dates
    MyDate date1 = new MyDate(1999, 10, 12);
    MyDate date2 = new MyDate(2004, 12, 8);

    // sample professors
    Professor p1 = new Professor("111", "F1", "M1",
        "L1", "p1", "F1 M1 L1", null,
        date1, date2);
    Professor p2 = new Professor("222", "F2", "M2",
        "L2", "p2", "F2 M2 L2", null,
        date1, date2);
    Professor p3 = new Professor("333", "F3", "M3",
        "L3", "p3", "F3 M3 L3", null,
        date1, date2);

    // professors work at the college. Use these statements only if your constructor does
    // not add the professors to the collection
    c.addProfessor(p1);
    c.addProfessor(p2);
    c.addProfessor(p3);

    // the department
    Department d1 = new Department("Gaelic Studies", p2.getFullName());
}```
// professors are in the department
d1.addMember("111");
d1.addMember("333");

System.out.println(d1.departmentList());
}

The department list appears in the Terminal window since the test uses System.out.println instead of assert.

By the way, the unit test appears to show that the department chair is not a member of the department. Ask your professor to see if this is the situation at your college or university.

If it is not, how would you modify the Department class?

Collection of departments

Now that we have a functioning Department class, we can modify College to contain a collection of Department objects. You will need the statement

theDepartments = new HashSet<Department>();

We will leave it as an exercise for you to implement the collection of departments, and the addDepartment, removeDepartment, and departmentList methods.

Is there a clone method for departments? There should be one. Make it so.

Collection of students

As mentioned earlier, a college contains a collection of students too. Everything we have said about the collection of professors, and much of what we have said about departments, also applies to the collection of students. You will need the statement

theStudents = new HashSet<Student>();

We will leave it as an exercise for you to implement the collection of students, and the addStudent, removeStudent, and studentList methods.

Is there a clone method for students? There should be one. Make it so.
Summary

Most things in the world come in collections, they do not come as singletons.

We have seen one type of collection, the set, and its various implementation, and how to use it. We noted that there are other kinds of collection, including lists and maps, which we will see in subsequent chapters.
Exercises

1. Implement the collection of students referred to in the chapter.

   That is, the College class will contain a collection of Student objects. You should be able to produce a list of all the students registered at the college.

2. Implement the collection of departments referred to in the chapter.

   That is, the College class will contain a collection of Department objects. You should be able to produce a list of all the departments at the college.

3. Modify the college model so that a professor is not actually removed from theProfessors.

   To do so, create an instance variable which contains the professor’s status. Possible values are active, resigned, and retired. Only active professors should appear in the output of professorList.

4. In a previous chapter, we discussed modelling a bank account.

   Bank accounts have transactions associated with them. The transactions come in many forms – withdrawals, deposits, interest payments, preauthorized deductions, etc. Model a Transaction class.

   Then modify your BankAccount class to contain a collection of Transaction objects. Should you use a Set or a List? Why?

   Should the BankAccount class contain the collection of Transaction objects, or should there be a Bank class which contains the collection of Transaction objects? Why?

5. In a previous chapter we discussed modelling the sightings a birder makes.

   Create a YardList class which contains a collection of all the sightings a birder has made in his/her yard. If you imagine the birder does not live in a house, but lives in a flat (or an apartment), create a class which contains a collection of all sightings made at some specific location.

   I have heard of birders who maintain lists of the birds seen from work, or seen in the movies, or seen on the trip to and from work. The most-unusual was the birder who kept a list of all species seen reflected in the monitor of his computer, which faced out the window. With LCD monitors, I imagine that list doesn’t get much use now.

6. In a previous chapter we discussed modelling playing cards.

   Having a PlayingCard class allows you to model any number of card games. Select a card
game with which you are familiar and create a class which contains a collection representing the cards you may hold at any time.

In bridge, you may be holding 13 cards. In poker you may be holding five. In Go Fish, you may be holding many.

How would you model a game which uses several 52-card decks at once?

7. In a previous chapter we discussed modelling a die. Many games use several dice.

![Image of dice](image.png)

Create a class, the skeleton for a game, which contains a collection of dice. For some possible games, consult http://en.wikipedia.org/wiki/Dice_game.

8. In the chapter we referred to a multiset.

Implement a Multiset class. Examine the Google MultiSet class and see how your code differs.
Chapter 9 – Collections, part 2

Learning objectives

By the end of the chapter you will be able to:

- Describe the use of a comparator
- Implement a comparator and use it with ordered collections
- Use a TreeSet
- Format numbers
- Produce reports from collections

Introduction

Here, we will examine more of the collections a college contains, and then focus on tools and techniques we need to be able to process collections. This includes the production of complicated reports, including formatting numbers.

A collection of courses

Not only does a college contain a collection of students, a collection of employees, and a collection of departments, it also contains a collection of courses. What is a course? What data do we need to remember about courses?

The Course class

At my college, a Course object needs a subject abbreviation (String) and a course number (int). It also needs a course title (String), a course description (String), the number of credits a student earns by passing the course (int), and the number of hours each section of the class meets (lecture (double), laboratory (double), and seminar (double)).

The course title is a short phrase; the course description is a longer statement about the course.

For example, this textbook was written to be used in COSC (the subject abbreviation) 111 (the course number), whose title is “Computer Programming I”, and whose description is “This course is an introduction to the design, implementation, and understanding of computer programs. Topics include problem solving, algorithm design, and abstraction, with the emphasis on the development of working problems.”
The hours will appear in the calendar as a triplet, in the order lectures, labs, seminars. For this course, the triplet is (4, 2, 0).

You can see this by visiting [http://webapps-1.okanagan.bc.ca/ok/calendar/calendar.aspx](http://webapps-1.okanagan.bc.ca/ok/calendar/calendar.aspx).

A `double` is a primitive datatype (hence begins with a lowercase letter) which stores numbers containing a decimal point.

In our model, usually the lecture hours will be no more than five; the lab hours will be no more than five; the seminar hours will be no more than two; the credit hours will be no more than four. You'll need such information if you create exceptions for incorrect hours. Larger hours will occur for courses which are taught in an intensive manner (all day for a week, for example) before a semester begins or after it has ended. In this example, the values will be no more than 50.

The `double` datatype allows numbers much larger than these, but we will use `double` rather than the smaller `float` datatype, if only so that we do not need to put the letter `f` after numbers. That is, 1.0f is a `float` and 1.0 is a `double`. As memory is cheap, most programmers waste a few bits and use `double` instead of `float`. We will adopt that practice.

### Course – unit tests

Create the `Course` class, including the constructor and the `toString` method. Create unit tests to ensure a few courses are created correctly. Sample courses could include:

- A course worth one credit, with three lecture hours, two lab hours, and no seminar hours.
- A course worth two credits, with zero lecture hours, four lab hours, and one seminar hour.
- A course worth three credits, with 3.5 lecture hours, 2.5 lab hours, and no seminar hours.

What happens when you run the unit tests? Are they successful?

### Formatting the output

Examine the output. Everything looks fine until you notice that the triplet containing the lecture hours, lab hours, and seminar hours is slightly incorrect, at least according to the standards at my college. Your `toString` method did display the hours, didn’t it?

Different colleges display their hours differently. My college standard is that numbers with a zero after the decimal point should be displayed as integers. Thus (3.5, 2.5, 0.0) is two-thirds correct and (3.0, 2.0, 0.0) is totally incorrect. The outputs should be (3.5, 2.5, 0) and (3, 2, 0). We need a way to format the numbers as they are converted to `String`s.

Using the idea of “formatting” numbers, you look in the Java documentation, and find the `Format` class.
This is “an abstract base class for formatting locale-sensitive information such as dates, messages, and numbers.” That sounds like what we want to do, but as we have seen we can’t instantiate objects of an abstract class.

In the “see also:” section of the documentation, you’ll see NumberFormat which is also an abstract base class.

The “see also:” section of NumberFormat includes a link to DecimalFormat, “a concrete subclass of NumberFormat that formats decimal numbers.” Exploring this class eventually leads you to a chapter in the Java tutorial, entitled Formatting. http://download.oracle.com/javase/tutorial/i18n.

After reading this portion of the tutorial, you realize that this class will solve our problem, and that formatting consists of two steps.

- First, you create a formatting object using a specific pattern which shows how the output is to be formatted.
- Second, you provide a number to that formatting object, which returns it, formatted according to the pattern you specified.

In our case, specifying a pattern of "#.##" will provide the output in the correct format. It will display the digit before the decimal point and will display the decimal point and digit following it only when that digit is not zero.

Thus, we add the statement

import java.text.DecimalFormat;

to Course. Notice that we are using a different Java package for this class. Previously we have imported classes only from java.util.

Modify toString so it appears as follows. Comments have been omitted in the interest of saving trees.

```java
public String toString() {
    DecimalFormat myFormatter = new DecimalFormat("#.##");
    String result;
    double totalHours;
totalHours = lectureHours + laboratoryHours +
    seminarHours;
    result = subjectAbbreviation + " " + courseNumber;
    result = result + "." + credits + "."
    + myFormatter.format(hours);
    result = result + 
    title + 
    description
    + "(" + myFormatter.format(lectureHours)
    + "," + myFormatter.format(laboratoryHours)
    + "," + myFormatter.format(seminarHours)
    + ")";
    return result;
}
```
Create the unit tests for the getters and setters; then create the getters and setters themselves.

What happens if you create a class with 10 or more hours of lectures per week? Do you need to change myFormatter?

**printf**

A recent addition to Java is the printf method, adopted from C and C++.


**A collection of courses, continued**

Now that we have a Course class, we can resume discussions on a collection of Course objects.

The collection will be an instance variable of the College class, and we can base its implementation on our implementation of the theProfessors collection.

Make it so, except that we do not want to remove courses, ever. This is so that a student who graduated many years ago can still ask for details on the course.

**The Section class**

A course is offered in one or more sections. A first-year English course may have many sections; a first-year calculus course may also have many sections. At a small college or university, a third-year course in any subject may have only one section.

In the previous chapter, we discussed which class should contain the collection of sections. I decided that it would be the College class which contains a collection of sections. What type of collection will we use?

**Set? List?**

Rather than using a Set as we did earlier, let’s look at another type of data structure, a List. Note that there is nothing wrong with a set. A set would be an acceptable implementation of this collection; I simply would like to look at another type of collection.

A list is a collection of elements, in which there is some ordering of the elements based on position in the list. That is, there is a first element and there is a last element. For each element (except the first) there is a previous element. For each element (except the last) there is a next
element. You may insert new elements at the end of the list, or in the middle of the list, moving other elements to subsequent positions.

For a circular list, the first element in the list has a previous element, the last one in the list. For the last element in the list, there is a next element, the first element in the list.

An array is a type of primitive data structure found in almost all programming languages, recent or ancient. An array is a collection of elements, each of which is associated with a position number or index. To retrieve an element from an array, you specify the name of the array and the index. The index usually starts at 0.

An ArrayList is an implementation of the List interface, whose underlying implementation is an array.

To College, add the import statements

```java
import java.util.List;
import java.util.ArrayList;
```

And then declare the data structure.

```java
private List<Section> theSections;
```

In the constructor, allocate some memory to the data structure

```java
this.theSections = new ArrayList<Section>();
```

When you read the documentation describing ArrayList, you’ll see that you have created a data structure with room for 10 elements. If you add more than 10 elements to the collection, the collection will expand to hold the extra values. By what amount does it expand?

Create a method, addSection, which will add a section to the Course.

```java
/**
 * add a section to the offerings of a course.
 * @param s the Section to be added
 */
public void addSection(final Section s) {
    theSections.add(s);
}
```

The online documentation tells us that the method adds the section after already existing elements. If you wish to add it elsewhere, at the beginning of the list for example, use an alternative form of the add method, one which has two parameters, the position (or index) at which the object is to be added and the object itself. If that position is zero, the object is added at the beginning of the collection rather than at the end.

Do we need to clone the Section when we add it?
Yes, we should. We want only one copy of the Section available so that when we change a meeting time of the Section, we need only change it in one place.

You said the Section doesn’t contain a meeting time? Oops, we made an error in our original analysis of the problem and designed a class which omits an important feature. Correcting the omission is another example of refactoring. We will correct this omission in the next chapter, where we create a Meeting class.

The code in the last few pages looks good, but we don’t have a Section class yet, so none of the code will even compile.

**What is a section?**

What information do we really need to model a Section?

We have used unique identifiers for professors (employee numbers) and for students (student numbers). The combination of subject abbreviation and course number ensures uniqueness for courses so we don’t need a unique identifier. For sections, one possible unique identifier is subject abbreviation, course number, plus something to indicate when the section is offered. That sounds complicated. Let’s create a unique identifier (a number) instead.

We still need the course with which a section is associated. We will ensure this by having a Section object contain the subjectAbbreviation and the courseNumber.

Thus we have the beginning of a Section class.

```java
private int identifier;
private String subjectAbbreviation;
private int courseNumber;
```

For our reports, we need to know whether we are dealing with a lecture section, a lab section, or a seminar section. This gives us another instance variable.

```java
private String sectionNumber;
```

At my college, lecture sections will have a numeric section number; lab section numbers begin with the letter L, and seminar sections begin with the letter S. So, even though we use the term “section number” we will not represent it with a number.

We need to know when and where a Section object meets.

There are two aspects to “when.”
- The semester in which the section is offered, and
- The day of the week, the start time on that day, and the end time on that day during that semester.
We will cover the semester in this chapter, the meeting details in the next.

A section of a course offered in the current year is different from a section of the course offered last year. Thus a Section object also needs to contain instance variables defining the semester when it is offered. One possibility is to store W2008, for example, to indicate the winter semester of 2008. F2008 and S2008 might also be used. But what is “winter”?

My college uses something similar. It has three semesters in the year, numbered 10, 20, and 30. The semester beginning 2012 January 1 is 201210, the semester beginning 2012 May 1 is 201220, and the semester beginning September 2012 is 201230.

We have defined Winter as 10, Summer as 20 and Fall as 30.

A better, that is, more-generally applicable, solution is to remember the start date (year, stored as at least four digits, and month) as well as the end date (year, also stored as at least four digits, and month).

When discussing birthdates, we mentioned the Calendar and GregorianCalendar classes. The Calendar class has some constants representing the months of the year. A section of a course is different when it is offered in the fall semester (At my college that is September through December, or should that be Calendar.SEPTEMBER through Calendar.DECEMBER?), the winter semester (Calendar.JANUARY through Calendar.APRIL), or the summer semester (Calendar.MAY through Calendar.AUGUST).

We represent the day of the week (needed in the next chapter) similarly. The Calendar class provides us with Calendar.SUNDAY, Calendar.MONDAY, etc.

But remember that the month and day-of-the-week constants in the Calendar class all begin at zero. Thus, when you display Calendar.JANUARY, you’ll see zero rather than one. So long as we can remember to increment (increase by one) the value before we display it, we will be okay.

We need to add some instance variables to the Section class.

```java
private int startYear;
private int startMonth;
private int endYear;
private int endMonth;
```

Why did I make these variables ints?

One reason is that the values these instance variables will contain are small. Another is that the Field Summary portion of the documentation describing Calendar says the values for the months and days of the week are ints and we are using its constants. We need to modify the constructor to accept the values of those instance variables and we need setters and getters as well as creating and/or modifying the appropriate unit tests (an exercise left to the reader).
Warning – just because unit tests are “left to the reader” doesn’t mean they are unimportant. I
refuse to evaluate any assignments which my students submit without a complete set of unit
tests.
We need to modify the toString method so it shows the start and end dates. We need to deal with
two subtleties in toString.
First, the months need to be increased by one before printing. We can do that.
Second, recall the problems we had displaying birth dates, with single integers less than 10
displaying with no leading zero. As a solution, we used an if statement to display a zero if
necessary. But now we know how to use the DecimalFormat class to print that leading zero, if
necessary.
DecimalFormat actually lets us solve both problems simultaneously. The following statements are
the body of the toString method.
DecimalFormat myFormatter = new DecimalFormat(″00″);
String result;
// details of the section
result = identifier + ″ ″ + departmentAbbreviation
+ courseNumber + ″ ″ + sectionNumber;
+ ″ (″ + startYear + '-' + myFormatter.format(startMonth +1)
+ ″ to ″ + endYear + '-'+ myFormatter.format(endMonth + 1)
+ ')';

The pattern we have used (in the first statement) causes two digits to be displayed, with the first
one being zero if necessary.
What would the output be if we used
result = result + ″ (″ + startYear + '-' + startMonth + 1
+ ″ to ″ + endYear + '-' + endMonth + 1
+ ')';

Can you explain why the output is not what you expected?

The Singleton pattern revisited
The College class is acting as a gatekeeper to the individual Course objects, as it does for
Professor objects. Sometimes we need to retrieve single Course objects, as when we wish to add,
or remove, a section. That is, we need a method in the College class which returns a Course
object, when we provide its subject abbreviation and course number.
public Course getCourse(final String subjectAbbreviation,
int courseNumber) {
for (Course c : theCourses) {
if (subjectAbbreviation.equals(
c.getSubjectAbbreviation()) &&

194


Using the object returned by the method, we are able to add sections to the courses the college offers.

Here is another place where we may wish to throw an exception rather than returning a null object.

In the same way, we can create a getSection method, by providing the subject abbreviation, course number, and section number. Make it so.

**Listing a collection in order**

Whenever we talk about listing a collection in order, we must have some way to decide whether one element of the collection precedes another. To produce a list of the professors employed by the college in some special order, we need to be able to describe how to compare one professor to another to determine which precedes the other. In the context of a college, there are several orders which matter for professors, mentioned earlier (name, employee number, or hiring date).

To produce a list of the courses offered by the college in alphabetical order by subject abbreviation, we need to be able to describe how to compare two Course objects.

To produce a list of the students registered at the college, we need to be able to compare two Student objects. In the context of a college, there are several orders which matter for students, mentioned earlier (by identifier, name, or average mark. The mark may be for a semester, to determine who is on the Dean’s List or receives scholarships, or it may determine which of the graduates receive medals or scholarships.)

To produce a list of the sections offered at the college, we need to be able to compare two Section objects. They should be listed by starting date, subject abbreviation, course number, and section number.

Maybe we should have a Semester class, with the collection of sections within it.

We see that we need a technique which allows us to specify several different types of ordering for a class.

In all comparisons, some values are more important than others, so we make sure we compare them first. For example, when sorting a group of people by name, family name is more important than given name which is more important than middle name. When sorting by date, the year is probably more important than the month which is probably more important than the day.
The international standards for dates, year/month/day reflects my view of importance. See [http://www.iso.org/iso/date_and_time_format](http://www.iso.org/iso/date_and_time_format) for more than you wanted to know about dates.

Of course, when you are producing a list of people in order by their birthday, you may want only the month and the day; the year may not matter.

Let’s begin by seeing how to compare Professor objects.

**Comparable**

Since we have used the word “compare” several times, you may have gone to the online documentation and looked up “compare”. The closest match is “Comparable”.

The Comparable interface is interesting, but that is not a solution to our problem, since Comparable only allows one way of sorting, specified in a compareTo method. This one way of sorting is sometimes called the natural ordering for the data.

The natural order for numbers is numeric. The natural order for Strings is alphabetic, using the Unicode coding scheme. But what is the natural order for professors - by name, by identifier, by salary, or perhaps by hiring date? That's an impossible question to answer unless you say "It depends."

What entry comes immediately after Comparable in the online listing of classes and interfaces? The Comparator interface! This is what we need.

**Comparators**

The Comparator interface allows us to have many compare methods associated with one class.

We begin by having the Person class (after all, a professor is a person) create a Comparator object.

All Comparator objects contain a method named compare.

All compare methods behave in the same way: they accept two parameters and return a negative integer when the first is less than the second, zero when their values are equal, and a positive integer when the first is the larger.

To use Comparator objects you need the statement

import java.util.Comparator;

at the beginning of the Person class and any other classes which contain comparators.
The logic of an alphabetical comparison is: Compare the family (last) names. When they are not the same, you know which is less. When they are the same, compare the given names. If the family and given names are the same, compare the middle names.

You could implement the logic using a series of if statements, or you could notice that the String class has a compareTo method which returns the same values as a Comparator.

The String class also has a compareToIgnoreCase method which you may elect to use if you wish to ignore the difference between uppercase and lowercase letters.

We have used the terms uppercase and lowercase several times now.

Uppercase letters are the capital letters. They are named “uppercase” from the olden days when printers had to make up lines of text by placing individual letters into trays. The capital letters were kept in the upper part of the storage area, hence the term uppercase. Similarly, the other letters were kept in the lower part of the storage area, hence the term lowercase.

How are uppercase and lowercase letters stored in a computer? They are all just bit patterns.

When you look at the Unicode description for the Latin alphabet, the one commonly used in English-speaking countries, at http://www.unicode.org/charts/PDF, you’ll find that the uppercase letters we need are represented by hexadecimal values between 0041 (the first three digits are the column heading on page 2 of that document) and 005A, and the lowercase letters are represented by hexadecimal values between 0061 and 007A. That is, the uppercase letters have a lower hexadecimal representation than the lowercase letters. Interesting.

But it is easy to convert from a lowercase letter to the corresponding uppercase letter; treat the characters as numbers, and simply subtract the hexadecimal number 0020 from the numeric equivalent of the lowercase character. Fortunately there is a method, Character.toUpperCase, to do this automatically for us.

Why would you want to ignore case as you compare names? Perhaps you want to have Von Smitz and von Smitz appear together on the list.

When comparing characters, note that a blank is less than a digit, which is less than any uppercase letter, which is less than any lowercase letter. Within the uppercase (and lowercase) category, letters compare in normal alphabetical order.

```java
/**
 * Alphabetic comparison. The compare method
 * returns negative, 0, positive if Person p1 is <, =, or > Person p2
 * based on the family name, given name, and other name.
 * Both objects must be Person objects.
 */
public static final Comparator<Person> ALPHABETIC_ORDER = new Comparator<Person>() {
    public int compare(Person p1, Person p2) {
```
int result = 0;
// compare family names
result = p1.getLastName().compareTo(p2.getLastName());
if (result == 0){
    // family names are the same
    result = p1.getFirstName().compareTo(p2.getFirstName());
    if (result == 0)
        // family and first names are the same
        result = p1.getMiddleName().compareTo(p2.getMiddleName());
    }
return result;
};

Note that the method assumes both objects being compared are of type Person. It will not work if this assumption is not valid.

Any statement about what must be true before a method is invoked is known as a precondition, and should be documented in the method.

Any statement about what must be true after a method completes its processing is known as a postcondition. It too should be documented.

```java
javadoc does not have tags for preconditions and postconditions. Yet.
```

How would you change the method to return false if the two objects are not both Person objects?

Particularly note the semi-colon after the final brace. That semi-colon must be there as the statement, lengthy though it is, is just declaring an object (called ALPHABETIC_ORDER), and every declaration statement must end with a semi-colon.

This Comparator is declared to be static. This means there is only one method with the name (ALPHABETIC_ORDER) within the Person class. There may be many Person objects, but they all use the same method when comparing two Persons alphabetically. This is only sensible.

The name of the comparator is capitalized, following the convention that constants (declared using the reserved word final) are capitalized.

Compile the Person, Student, and Professor classes.

**Unit testing the comparator**

How do you create a unit test for a comparator? This is particularly difficult with the comparator we have written, since it is in an abstract class, and we can’t create unit tests for abstract classes.

But we note that both Student and Professor are derived from Person, so we can place our unit tests for the comparator in the tests for either of those classes. Here is one such test, from the Student class.
@Test
public void testComparator1() {
    Student s1 = new Student("123", "Richard", "Dennis", "Gee", "Rick", "Richard D. Gee", a, d);
    Student s2 = new Student("125", "Rick", "", "Gee", "Rick", "Richard D. Gee", a, d);
    assertTrue(Person.ALPHABETIC_ORDER.compare(s1, s2) < 0);
}

The variables a and d are the names of Address and MyDate objects. For the unit test, they could have any value, including null.

The test will succeed. The family names of the two students are the same but the given (or first) name of s1 is less than the given name of s2. The first three letters of the given name are the same, but the fourth letters are different, and “h” is less than “k”. For characters, alphabetical order translates into “less than.”

Note the name of the unit test. The digit on the end of it is meant to imply there will be many such tests, one for each possible path through the comparator. To fully test the comparator, there should be many unit tests. How many?

Each comparison can have three outcomes, and there are three comparisons. Thus there are theoretically three (the number of outcomes per comparison) raised to the power three (the number of comparisons), or 27, unit tests. But some of the tests are unnecessary. There is no need to test the first name when we know that the family names are different. Thus we need three tests to deal with the family name. We need three more to deal with the given name where the family name is the same. We need three more to deal with the middle name if the family and given names are the same. Thus we only need nine tests.

Later in the chapter we see an alternative form of the comparator, one which requires only three comparisons and hence only three unit tests.

In subsequent sections, we will see comparators for different classes, which will involve four or more comparisons, each of which may have three outcomes, thus requiring three raised to the power four, or 81, unit tests.

Alternative versions of the comparator

Note the comment earlier about the distinction between lowercase and uppercase letters. If we wish to ignore the case of the names (so that we treat Von Smitz and von Smitz as identical), we replace the compareTo methods in the comparator above with compareToIgnoreCase.

You could also combine the parts of the name into one string and compare those. That gives the following comparator.

    public static final Comparator<Person>
    ALPHABETIC_ORDER_ALTERNATIVE = new Comparator<Person>() {
        public int compare(Person p1, Person p2) {
            String name1 = p1.getLastName() + ' ' +
p1.getFirstName() + ' ' + p1.getMiddleName();
String name2 = p2.getLastName() + ' ' +
    p2.getFirstName() + ' ' + p2.getMiddleName();
return name1.compareToIgnoreCase(name2);
}
);

This is yet another example that there are often many ways to accomplish the same result.

When I first wrote that method I did not include the single characters between the parts of the
name. But then I realized that there could be examples in which the space would matter.
Consider the two names Pauly Ian Smith and Paul Yian Smith.

Definitions – class and object - revisited

Do you recall our circular definitions of class and object? We said that a class is a blueprint for
objects and an object is an instance of a class.

The classes we have seen so far are models (simplifications) of something in the real world. But
now we have a class (actually, an interface, but it's close enough to a class), Comparator, which is
not a model of something in the real world. You might walk down a street and see a Student
object coming towards you but you will never walk down a street and see a Comparator object
coming towards you.

Thus we should restate our definition of a class. How about “A class is a model of something in
the real world or it is a way of implementing a concept.”

Our definition of an object is still appropriate. “An object is an instance of a class.” Notice that
our Comparator object is created by using the constructor of the Comparator class.

Some programming languages allow a class to be derived from several classes. This is called
multiple inheritance. In Java only single inheritance is possible, but the same goal is achieved by
having a class extend one class and implement one or more interfaces.

Producing an alphabetical list of professors

How do we combine our knowledge of collections and comparators so that we can produce a list
of professors at the college in alphabetical order by name?

We find that we have a problem. We have implemented theProfessors as a HashSet because of its
speed. But now we need a different implementation since a HashSet is not ordered and, more
importantly, cannot be ordered.

So we select a different implementation of the Set interface, the TreeSet, which can be ordered.
We will transfer the data from the HashSet to the TreeSet, creating a new set ordered as the Comparator prescribes, but we will do the transfer only when we need the data in order. Most of the time, we leave the set in its unordered state.

To use a TreeSet requires the following two changes in the College class. First, we add an import statement.

```java
import java.util.TreeSet;
```

Then we create a new method, `alphaProfessorList`.

```java
/**
 * produce a professor list, in alphabetical order.
 */
public String alphaProfessorList()
{
    // create the TreeSet
    Set<Professor> alphaProfessors = new TreeSet<Professor>(Person.ALPHABETIC_ORDER);
    alphaProfessors.addAll(theProfessors);

    // traverse the Set (in alphabetical order)
    String result = "";
    String prefix = "\n";
    for (Professor p: alphaProfessors) {
        result += prefix + p.toString();
        prefix = "\n";
    }
    return result;
}
```

The statement that creates the TreeSet indicates what type of objects will be in the set and what comparator is used to compare two such objects.

The `addAll` method transfers all the elements of the (unordered) HashSet into the (ordered) TreeSet. Once that is complete, we simply need to visit the elements of the TreeSet one after the other and we will visit them in alphabetical order.

How a TreeSet represents its elements in order is of no concern to us right now. If you want more details, consider taking a data structures course, the normal follow-up to an introductory programming course.

We have used a TreeSet because most of our collections are Sets and a TreeSet is ordered by the comparator. But our collections could have been Lists, using the ArrayList implementation. To sort the elements in a List using a comparator, use the sort method from the Collections class.

```java
Collections.sort(theList, theComparator);
```
The for-each statement, revisited

Note that we have again used the for-each statement to visit all the elements in a collection. We are not altering those elements so we do not need an iterator.

```java
String result = "";
String prefix = "";
for (Professor p:alphaProfessors) {
    result += prefix + p.toString();
    prefix = "\n";
}
```

We could use an iterator to process all objects in the collection but the for statement provides a shorthand way of referring to individual elements in the collection.

By the way, what is the purpose of that prefix variable?

Sometimes I ask questions without answering them. The reason is to get you reading code and understanding it.

The ability to read and understand code is crucial in your career as a programmer. Many times you will see code which is undocumented, but you need to understand what it does.

BlueJ revisited

This may be an appropriate point to mention the Test Fixture to Object Bench menu option in BlueJ.

When you right-click a unit test on your class diagram, one of the options on the menu that appears is Test Fixture to Object Bench. This creates all the objects you need to run your tests, and places them on the Object Bench, which we used in the beginning of the book but haven’t used lately.

We can run individual methods of these objects, independent of the unit tests. This can be useful when we are trying to debug (fix the errors in) an object or class.

Debug is a term that dates back to the beginning of computers. At that time, the memory was switches which actually moved. The story is that one of these switches was behaving erratically. When someone went inside the computer (Yes, they were very large!) to check out why the switch was behaving erratically, they found a moth (colloquially called a bug) had become caught in the switch. To fix the problem, the moth was removed. Hence, the switch was de-bugged.

In any case, I prefer not to use the word debug as it has somewhat humorous connotations. You are finding and removing the errors in your program, you are not debugging it. Having errors in your program is not funny!

To get an idea of the problems that errors cause, I suggest you become a faithful reader of the Risks Digest. It is available online at [http://catless.ncl.ac.uk/risks](http://catless.ncl.ac.uk/risks) or via email. That link provides a link to subscribe.

The errors are not just small, either. Perhaps the largest and most expensive was the spaceship that crashed into the Martian surface due to confusion between metric and Imperial measurements.

**Producing a professor list in numeric order**

But let’s go back to the problem of modelling a college, its people, and its courses.

Sometimes it is more appropriate to have a list of employees ordered by identifier (employee number) rather than by name. To make this possible, we need a second Comparator, another method in College, and another unit test.

This Comparator, in the Person class of courses, is much simpler than the first one we created, since now we are comparing only the identifier values.

```java
/**
 * numeric comparison. The compare method
 * returns negative, 0, positive if Person p1 is <, =, or > Person p2, based on the identifier
 */
public static final Comparator<Person> NUMERIC_ORDER = new Comparator<Person>() {
    public int compare(Person p1, Person p2) {
        return (p1.getIdentifier().compareTo(p2.getIdentifier()));
    }
};
```

You will probably consider copying and modifying `alphaProfessorList` to create `numericProfessorList`. You will probably consider doing the same for the unit test.

That would be the easy way, but it has its problems. You may forget to make a change in the copy, for example. Or, even worse, the code you copy may contain an error, and now that error is in two places.

Can we avoid having the same code in two places? Of course we can, or I wouldn’t ask the question!

**Passing a Comparator to a method**
Did I hear you say “Why can’t we just have one professorList method and tell it which Comparator to use?” You can, and you should.

`/**`
`* produce an employee list, in the order specified by the Comparator.`
`* @param theComparator The Comparator to use in sorting the employees`
`*/`
`public String professorList(final Comparator<Person> c) {`
`   // create the TreeSet
`   Set<Professor> ordered = new TreeSet<Professor>(c);
`   ordered.addAll(theProfessors);
`
`   // traverse the Set
`   // and compute a String
`   String result = "";
`   boolean needsReturn = false;
`   for (Professor p: ordered) {
`      if (needsReturn) {
`         result = result + '\n';
`         result += p.toString() + '\n';
`         needsReturn = true;
`      }
`   }
`   return result;
`}
`

Don’t forget the statement

`import java.util.Comparators;
`

The coding in the method above is slightly different from that in alphaProfessorList. Once again we see that there are many ways to accomplish the same task. If you didn’t figure out the use of the variable prefix in the previous version, perhaps you can do so now.

A unit test for the alphabetic Comparator could look like this.

`@Test`
`public void testAlphaProfessorList1() {`
`   College c = College.getInstance();
`   // create some professors (code not shown here)
`
`   // and remember them
`   c.addProfessor(professor1);
`   c.addProfessor(professor4);
`   c.addProfessor(professor3);
`   c.addProfessor(professor2);
`
`   System.out.println(s.professorList(Person.ALPHABETIC_ORDER));
`}
`
Remember to use View, Show Terminal so there is a place on the screen to display the class list. Have you found out that you can display the terminal window after putting something in it? The output will still be available.

The unit test for the numeric Comparator would be the same, replacing ALPHABETIC_ORDER with NUMERIC_ORDER.

So we are able to remove the alphaProfessorList and numericProfessorList methods from College, and to remove their unit tests from CollegeTest.

**Registering students in sections**

But let’s get back to the Section class.

A section contains students. That is the English description. In our implementation, sections do not contain Student objects, they contain student identifiers.

We designed them that way, so that we only have one instance of a Student object for each student (in the theStudents collection in College) and we gain access to that instance by providing the only instance of the College class with the student number.

How do we create the collection of students in a section?

First, we need to decide the type of the collection we will use. We can use either a set or a list. There is no compelling reason for one over the other. Flip a coin. There it goes. Up, and up, over, and over, and it comes down … set.

Thus we need the following statements.

```java
private Set<String> theStudents;
```

In the constructor, we need

```java
this.theStudents = new HashSet<String>();
```

In the clone method we need

```java
s.theStudents.addAll(this.theStudents);
```

We need to add a student’s identifier to the collection. If we have a Student object, this method works.

```java
/**
 * add a Student object to the section.
 * @param s Student object to be added
 */
```
public void addStudent(final Student s) {
    theStudents.add(s.getIdentifier());
}

Sometimes we may have the Student object to be added to the collection in a Section object. Sometimes we may have the identifier. That allows us to have a second addStudent method.

/**
 * add a Student object to the section.
 * @param sn the student identifier to be added
 */
public void addStudent(final String sn) {
    theStudents.add(sn);
}

We have overloaded the addStudent method. We have two methods both with the same name but the parameter(s) differs. This is also an example of polymorphism, where two or more method have the same name but different signatures.

In a similar way we can overload a method by having many methods with the same name but with different numbers of parameters, or the same number of parameters but of different types.

Soon we will need to access all the students in the collection. Here’s a method to do that.

/**
 * @return collection of Student objects.
 */
public Set<Student> getStudents() {
    return theStudents;
}

But this is not a good method (In fact, it's a bad, bad method!) since it gives everyone access to all the elements in the collection and tells how the collection is implemented.

A much better way to accomplish a similar task is to give everyone access to an iterator, which provides one Student object at a time.

/**
 * @return iterator over a collection of Student objects.
 */
public Iterator<Student> iterator() {
    return theStudents.iterator();
}

**Producing an alphabetic class list**

How do we combine all our knowledge of collections so that we can produce a list of students in a Section, in alphabetical order by name?
We find that we have a problem, the same problem we found earlier. We have implemented a set as a HashSet because of its speed. But now we need a different implementation since a HashSet is not sorted.

Instead, we select the TreeSet again. As we did before, we will transfer the data from the HashSet to the TreeSet, creating a new set ordered as described by a Comparator, but only when we need the data in order. Most of the time, we leave the set in its unordered state.

Recall that the Section collection contains student numbers. Once we extract an element from that collection, we must retrieve the corresponding object from the College collection of students. This calls for a getStudent method within the College class.

```java
/**
 * given a student number, return the student object.
 * @param a student number
 */
public Student getStudent(final String identifier) {
    for (Student s:theStudents) {
        if (identifier.equals(s.getIdentifier())) {
            return s;
        }
    }
    return null;
}
```

To use a TreeSet requires two changes in the Section class. First, we add an import statement.

```java
import java.util.TreeSet;
```

Then we create a new method, alphaClassList.

```java
public String alphaClassList() {
    College c = College.getInstance();
    // create the TreeSet
    Set<Student> orderedStudents = new TreeSet<Student>(Person.ALPHABETIC_ORDER);
    // for all student identifiers in the section, retrieve the Student objects
    // and arrange them in alphabetic order
    for (String sn:theStudents) {
        orderedStudents.add(c.getStudent(sn));
    }
    // traverse the Set and compute a String
    String result = "";
    boolean needsReturn = false;
    for (Student st:orderedStudents) {
        if (needsReturn)
            result += '\n';
        result += st.getIdentifier() 't' + st.getFullName();
        needsReturn = true;
    }
    return result;
}
```
The statement that creates the TreeSet indicates what type of objects will be in the set and which comparator is used to compare two such objects.

We visit each element of the collection of student numbers, representing the students who are registered in the section. For each, we ask the college singleton to give us the appropriate Student object. We insert each into the TreeSet, thus ordering them alphabetically. Once we have visited all elements in the HashSet, we simply need to visit the elements of the TreeSet one after the other and we will visit them in alphabetical order.

**The for-each statement, revisited**

Note that we have again used the for-each statement, twice in fact, to visit all the elements in a collection.

The first time, we visit the elements in the theStudents collection.

```java
for (Student sn:theStudents) {
    orderedStudents.add(c.getStudent(sn));
}
```

Then we visit all the elements in the orderedStudents collection.

```java
String result = "";
for (String st:orderedStudents) {
    if (needsReturn)
        result = result + \
    result = result + st.getIdentifier
        + \
        needsReturn = true;
}
```

We could use an iterator to process all objects in the collection but the for statement provides a shorthand way of referring to individual elements in the collection.

**Producing a class list in numeric order**

Sometimes it is more appropriate to have a class list of students in a section, ordered by student number rather than by name. As we saw earlier, we can provide a comparator as a parameter to a method.

This Comparator is much simpler than the first one we created, since we are comparing only the identifier values.

```java
/**
 * numeric comparison. The compare method returns negative, 0, positive if Person p1 is <, =, or > Person p2 based on the identifier */
```
public static final Comparator<Person> NUMERIC_ORDER = new Comparator<Person>() {
    public int compare(Person p1, Person p2) {
        return p1.getIdentifier().compareTo(p2.getIdentifier());
    }
};

Modify alphaClassList so its name is simply classList, and so that it accepts a Comparator as a parameter. Create unit tests for both comparators.

**Students enrol in sections**

Now that we have created a Section object which contains a collection of students (that is, a collection of student identifiers), we are able to see how to model a college in much more detail. In particular, we can model the association between Student and Section where a student registers in a section of a course. We can model that by having a Student object contain a collection of section identifiers.

The other side of the association is that a Section knows which students are registered in it by maintaining a collection of student identifiers.

Make it so.

**SortedSet**

What is so special about a TreeSet? Is there some other implementation of Set which ensures its elements are in order?

Yes. Explore SortedSet.

**Summary**

Now that a section knows the semester in which it is taught, we can focus on the daily meetings of the class, the details of which include a room number, the start time, and the end time. We will do that in the following chapter.

The challenges of producing a schedule will introduce us to the complexities and fun of string manipulation.
Exercises

1. A student enrols in many courses, usually in one section of each, in a semester. Using the techniques from the chapter, model that.

That is, a Student object needs a collection of the section identifiers of the sections in which he/she is currently enrolled. Each Student object should be able to produce a currentEnrollment report listing all these sections.

2. A professor teaches many sections in a semester. Using the techniques from the chapter, model that.

That is, a Professor object needs a collection of the section identifiers of the sections he/she currently teaches. Each Professor object should be able to produce a currentTeachingLoad report listing all these sections.

3. A professor who has been at the college for more than one semester has a history of the courses which he/she has taught. Model that.

That is, a Professor object needs a collection of the subject abbreviations and course numbers of previously-taught courses. Each Professor object should be able to produce a previouslyTaught report listing all these courses.

How can that report show when the professor taught the sections?

4. In a previous chapter, we discussed modelling a bank account, with a collection of transactions. In what order should the transactions be stored?

Why do you suggest that order?

How would you implement that?

5. In a previous chapter, we discussed modelling the sightings a birder makes. In what order should the sightings be recorded?

Why do you suggest that order?

Note that a birder will sometimes wish to have a list in order by species and sometimes in order by date and time. Sometimes a birder will only want sightings for a particular time period.

How do those ideas affect your earlier answer?

6. In a previous chapter, we discussed modelling playing cards. In what order should the playing cards be stored in a collection?
Why do you suggest that order?

7. Explore the DecimalFormat class to see what other capabilities it contains.

8. Explore the printf method.

9. Describe how you produce a class list where the students are in a random order.
Chapter 10 – Collections, part 3

Learning objectives

By the end of the chapter you will be able to:

- Produce reports from collections
- Describe the characteristics of the StringBuffer class, including how it differs from String
- Use and manipulate the contents of a StringBuffer
- Describe and illustrate the difference between aggregation and composition

Introduction

In the previous chapter, we mentioned that we wish to model when a Section is offered and we mentioned there are two aspects to “when”; the semester (which we have modelled by providing start and end dates), and the days and times within the semester. We begin the chapter by modelling the days and times within the semester.

The Meeting class

Whenever the students and professor for a section meet, we will call that a meeting. We will indicate the date and time of each meeting, along with the location (room number) of the meeting. A Section object will contain a collection of Meeting objects.

We will use a 24-hour clock to represent the start and end times. Thus the values range from 0000 to 2359 and the corresponding instance variables will be integers.

We will use the constants from the Calendar class to represent the days of the week. Thus, the corresponding instance variables will be integers.

Can a section meet twice on the same day? That would be unusual, but it is possible. We must make sure we do not make that impossible. The section in which this book was first used originally met in a 120-minute block, followed by a recess. We changed that to three 40-minute blocks, each separated with a 10-minute recess. This is an example of a section meeting several times in one day.

To simplify the model a little, we will assume the room number of the meeting is simply a String.
Instance variables

Since a section meets many times a week, we should create another class, one called Meeting, and a Section object should contain a collection of Meeting objects. An object has identity, state, and behaviour, as we have already seen many times.

Identity is the name of the object, and we create the name through a declaration statement, so there is no problem there.

What is the state of a Meeting object? That is, what are its instance variables and how do they receive their values? We've discussed them in the previous paragraphs.

private int dayOfWeek;
private int startTime;
private int endTime;
private String roomNumber;

Behaviours

And what are the behaviours of a Meeting object? Aside from the obvious getters and setters, it may be useful to have a method that computes the duration of a meeting in minutes.

A meeting may begin at 1000 and end at 1050, a duration of 50 minutes. But it may begin at 0830 and end at 0920, also a duration of 50 minutes. In the first case, you can subtract the start time from the end time and calculate the length immediately, but you cannot do this in the second case.

How do you calculate the length of a meeting?

Time arithmetic

There are many ways to calculate the duration of a meeting, but all require you to have both the minutes and hours of both the beginning and ending times available. How do you separate a number like 0830 (actually stored as 830) into its hours (8) and its minutes (30)? Probably the easiest way is as follows.

int startHours = startTime / 100;
int startMinutes = startTime % 100;

In each of those statements we declare a variable and give it a value at the same time.

The first statement uses division (division is represented by the /, or slash) to isolate the hours. When you divide an integer by an integer, the result is an integer; any remainder in the division is dropped and no rounding takes place. For example, 830 / 100 is 8, 1020 / 100 is 10, 1050 / 100 is 10, and 955 / 100 is 9.
To determine the remainder discarded in the integer division, use the modulus operator (represented by %). 830 % 100 is 30. That is, the remainder when you divide 830 by 100 is 30.

1020 % 100 is 20; the remainder when you divide 1020 by 100 is 20.

1050 % 100 is 50; the remainder when you divide 1050 by 100 is 50.

955 % 100 is 55; the remainder when you divide 955 by 100 is 55.

Remember that the modulus operator is used only when you are doing integer division.

Similarly, we can calculate when the meeting ends.

```java
int endHours = endTime / 100;
int endMinutes = endTime % 100;
```

Once you have the hour and minute when the meeting begins and ends, it is a simple matter to calculate its duration, in minutes.

```java
return (endHours - startHours) * 60 + (endMinutes - startMinutes);
```

This is one of the more complicated arithmetic calculations we have seen. It involves the idea that some calculations have priority over others. That is, some calculations must be done before others. Parentheses indicate that the calculations within the parentheses should be done before any other calculations. In our case, there are two parenthesized calculations which are performed (the two subtractions) and the results saved in temporary storage, which we never see.

To decide what is done with those results, you need to know that multiplication (indicated by an asterisk, *) has a higher priority than addition. Thus the first saved result is multiplied by 60 and the result of that calculation is added to the second saved result.

For example, when a meeting runs from 0830 to 0950, the calculation computes (9 – 8) * 60 + (50 – 30) or 1 * 60 + 20, or 60 + 20, or 80 minutes.

If a meeting runs from 1130 to 1220, the calculation computes (12 – 11) * 60 + (20 – 30), or 1 * 60 + (–10), or 60 + (–10), or 50 minutes.

Should a calculation use more than one multiplication operation, they would be evaluated from left to right. So too would division and so too would a mix of multiplication and division operations.

More-complicated calculations could use more than one addition operation; they would be evaluated left to right as well, but only after the multiplication and division operations had been completed.

Subtraction and addition have the same priority, but lower than multiplication and division, and are done left to right.
Exceptions

Note that these calculations assume there are no errors (intentional or otherwise) in the times provided. There is an expression “Garbage in, garbage out.”

For example, due to sloppy input handling, a meeting may have a start time of 3456 and an end time of 6578. It may have a start time of 1200 and an end time of 1030. In both cases, we will calculate an incorrect value for the duration of the meeting.

To prevent these errors entering our system, we should throw an exception whenever we detect a bad piece of data. Bad pieces of data include hours too large, minutes too large, and a negative value for the time.

Make it so.

Section uses Meeting

Now that we have a Meeting class, we can attach a collection of Meeting objects to a Section object. What type of collection should we use?

We have seen a List (of section identifiers added to a Course object, for example) and the ArrayList implementation. We have seen a Set (of student identifiers representing students in a section) and we have seen two implementations (HashSet and TreeSet.)

When you check the online documentation, you’ll find there are other types of collections as well. For now, though, I think a TreeSet is the most appropriate implementation to select.

Why would I suggest that?

A Set does not allow duplicates, and two meetings of a section cannot occur at the same time. Thus a set is a suitable collection. But which implementation should we select?

Either implementation might work. But a TreeSet is ordered and we may want to retrieve the Meeting objects in order by day of the week.

In particular, we will produce a schedule showing when each section of each course is offered. The individual meetings will be in order from Sunday through Saturday, following the normal North American week.

Why is that the normal order, when Saturday and Sunday form the weekend? An “end” implies it is after something.

But the English language is infuriating in many ways. There is the old question “Why do you drive on a parkway and park on a driveway?” And there is the other question, “If a train stops at a train station, what happens at a workstation?”
Many religious groups and calendars from Europe seem to have it right, with the week beginning on Monday and ending on Sunday.

Since we are going to use a TreeSet, we need to specify a Comparator<Meeting> which describes how to compare one Meeting object to another. (And we need an equals method for Meeting objects. That one is easy. They are the same when they take place in the same room on the same day at the same time.)

/**
 * is one meeting equal to another?
 * @return true if day, times, and room match
 * @return false if not both Meetings, or if something is different between the two Meetings
 */
public boolean equals(Object o) {
    if (o == null)
        return false;
    if (o == this)
        return true;
    if (o instanceof Meeting){
        Meeting m = (Meeting) o;
        return (this.roomNumber.equals(m.getRoomNumber())) &&
                (this.dayOfWeek == m.getDayOfWeek()) &&
                (this.startTime == m.getStartTime()) &&
                (this.endTime == m.getEndTime());
    }
    return false;
}

Note that we need to use the equals method with the roomNumber, since it is an Object, a String; we use == with the day of week and the times, since they are primitive datatypes, ints.

If you don’t like using both equals and ==, you can convert an int to an Integer, and then use its equals method. But that makes code that is overly complicated and hard to read.

I suppose you could use the hashcode method to determine the hashcodes for each String and then compare the hashcodes using ==.

You could compute the hashcodes for the two Meeting objects and compare them using ==. That would be an elegant solution! What is a suitable way to calculate the hashcode for a Meeting object?

Just because there are many ways to accomplish the same task, they are not all sensible.

We combine four boolean values, using the “and” operation (&&). The only time this combination will evaluate to true is when all four boolean values are true; that is, when the day, room number, start time, and end time all match.

On to the comparator.
What does it mean that one meeting is less than another? Here is my understanding.

- When the first is on an earlier day of the week than the second, the first is less.
- If both are on the same day of the week, the one with the earlier start time is less.
- If both are on the same day of the week and have the same start time, then the meeting which ends first is less.

Based on that understanding, here is my Comparator<Meeting>.

```java
/**
* which meeting is less? The compare method returns negative1, 0, positive
* if Meeting m1 is <, =, or > Meeting m2 based on the day of week, start time, and end time
*/
public static final Comparator<Meeting> TIME_ORDER = new Comparator<Meeting>() {
    public int compare(Meeting m1, Meeting m2) {
        int result = 0;
        int d1 = m1.getDayOfWeek();
        int d2 = m2.getDayOfWeek();
        // compare day of week
        if (d1 < d2) // m1 is earlier in the week
            result = -1;
        else if (d1 > d2) // m2 is later in the week
            result = +1;
        else {
            // both on same day of the week
            int s1 = m1.getStartTime();
            int s2 = m2.getStartTime();
            if (s1 < s2)
                result = -1;
            else
                if (s1 > s2)
                    result = +1;
            else {
                // same day, same start time
                int e1 = m1.getEndTime();
                int e2 = m2.getEndTime();
                if (e1 < e2)
                    result = -1;
                else
                    if (e1 > e2)
                        result = +1;
                else
                    result = 0;
            } // compare end times
        } // compare start time
        return result;
    }
}; // end TIME_ORDER
```
This comparator is longer than the others we have seen. The reason is that all the values we are examining are primitive datatypes. Thus we cannot use the compareTo method we have used with Objects. We must use the less than (<) and greater than (>) operators instead.

In the old days, programmers were considered better when they could write shorter methods to accomplish a task. Many languages included special features to allow terse coding. Java still includes one, the use of the conditional operator, the question mark (?). I prefer not to use it, however. I feel that clarity is reduced by the use of that operator.

If you really want to use the conditional operator, explore its use on your own.

The if statement, revisited

This Comparator contains several if statements, nested within each other. The basic structure of the if statement is as follows.

```java
if (condition)
    Statement to do when the condition is true
else
    Statement to do when the condition is false
```

The “statement to do” needs to be enclosed in braces when it is a compound statement, containing several other statements. For example, two assignment statements make a compound statement, and would need to be enclosed in braces. But an if statement is not a compound statement, so an if within an if does not need braces, although you may wish to use them to increase clarity.

Thus some of the else clauses in the Comparator do not need braces (because the “statement to do” is a simple statement, or is another if statement.) Some do need braces because the “statement to do” combines several assignment statements and another if. Perhaps it is better to always use braces, so you eliminate the “does it or does it not” questioning.

Note that the Checkstyle extension we are using expects braces even when there is a single statement in the body of the if or the else. Despite that, the rules of the Java language are well-defined, so none of the if clauses in the example need braces because they all contain a simple statement, setting the variable result to an appropriate value.

Note that the braces help tell which else is associated with which if. Indentation does not do that and can imply associations which do not exist. Consider the following badly-indented statements.

```java
if (percentage > 90)
    grade = "A";
else
    grade = "F";
    message = "with distinction";
```
The third assignment statement will be executed regardless of which path your program takes through the if statement.

Consider the following statements.

```java
if (percentage > 90)
    grade = "A";
    message = "with distinction";
else
    grade = "F";
```

This will not compile since there are two statements following the if which are not enclosed in braces. The second statement is the first statement outside the if. Thus the else is not attached to an if.

**Creating the collection of Meeting objects**

How does a Section create its TreeSet of Meeting objects?

You might be inclined to add the Meeting objects to the constructor. But sections are often planned and created, without knowing the actual times the section will meet. Thus, I have not modified my constructor other than to create an empty collection of meetings.

```java
private Set<Meeting> theMeetings;
theMeetings = new TreeSet<Meeting>(Meeting.TIME_ORDER);
```

To add a meeting to the collection, I have created a new method `addMeeting`.

```java
/**
 * add a meeting time.
 */
public void addMeeting(final Meeting m) {
    theMeetings.add(m);
}
```

Stop! We have not cloned the Meeting object. Implement a clone method, or create a constructor whose parameter is a Meeting object.

Then `addMeeting` will become one of the following, depending on which solution you implemented.

```java
public void addMeeting(final Meeting m) {
    theMeetings.add((Meeting) m.clone());
}
```

or

```java
public void addMeeting(final Meeting m) {
```
theMeetings.add(m.clone());
}

or

public void addMeeting(final Meeting m) {
    theMeetings.add(new Meeting(m));
}

How do you remove a meeting from the collection? The following is the obvious answer, and is correct if you did not clone the Meeting object when you added it to the collection.

/**
 * remove a Meeting object from the section.
 * @param m the meeting to be removed.
 */
public void removeMeeting(final Meeting m) {
    if (theMeetings.contains(m))
        theMeetings.remove(m);
}

But you did clone the Meeting object so removeMeeting must search through the collection for the Meeting object to remove. Make it so.

How do you change the time of a meeting? Changing the time of a meeting may seem difficult, but it is actually easy. One solution is to simply remove the old meeting and replace it with a new.

/**
 * Change the time of a meeting.
 * @param m1 the meeting to be changed
 * @param m2 the replacement meeting
 */
public void changeMeeting(final Meeting m1, final Meeting m2) {
    theMeetings.remove(m1);
    theMeetings.add(m2);
}

It could be even easier by using the setters of the Meeting class. Explore that possibility.

Section uses Meeting – displaying the times

Now we proceed to the challenging part, how do you produce a neat summary of the meeting times? We’ll do that in a method named meetingsList.

To understand how the method works, consider a section which meets on Monday, Wednesday, and Friday, from 1030 to 1120 each day, in the same room. We have three separate Meeting objects, one for each day.
We wish to combine the information from these objects into a single String that reads M W F 1030 1120 XXX where XXX is the room number, assuming all three meetings are in the same room. This is the format used at Okanagan College; perhaps your college or university uses a different format.

Note the spaces (one after M, one after W, three after F, one after 1030, and one after 1120.) These spaces represent missing information (the one after M, the one after W, and the first two after F represent other days of the week) and whitespace to make the output clearer (the third one after F, the one after 1030, and the one after 1120.)

I don’t know about your school, but at my college, when classes meet at the same time on different days, they usually meet in the same rooms. But not always; I once almost had a class that met three times a week in a different room each day.

When the section meets at different times on the three days, or in different rooms, then we wish to produce separate lines of output. There may be two or more separate lines, depending on how many different times or rooms are used. Recently I have had classes that meet three times a week, once in one room and twice in a different room. The schedule will be two lines.

You should be thinking that a first step is to be able to extract individual Meeting objects from the collection. Note that these objects will be returned according to the days of the week, Sunday (represented by the number zero) through Saturday (represented by the number six), and two meetings on the same day will be in order by starting time, since we are using the TreeSet implementation and our comparator.

We don’t know in advance how many lines of output (Strings) we will need. Often that forces us into using a collection, which will expand as needed. This time, though, we can determine an upper bound providing we assume no section meets more than five times a week. Should a section meet five times a week, meeting at a different time (or in a different room), we might need five lines of output. We'll only need all five lines if the meeting times are all different, or the rooms are all different.

We will be writing a method which will create all the lines of output. At the end of the method, we can combine the (maximum) five Strings into one and return it.

Arrays

So what type of collection should we use to store five Strings? The collections we have seen all can vary in size. Using any of them is, to some degree, a matter of overkill. There is a tried-and-true collection type we can use, since we know that the size of the collection is small; we can use an array.

We have seen the idea of an array mentioned when we first saw the ArrayList class.
Recall that an array is a collection of data items, in which individual elements are referred to by means of an index. That is, when you have an array containing five elements, the first one has index zero, the second has index one, and the fifth has index four. This is the same zero-based numbering we found in the Calendar class, and it permeates computing. Why?

Remember that the smallest non-negative integer is zero. Back in the old days, when memory really mattered and machines were slow, it made sense to start numbering at zero.

So perhaps we should create an array of String objects. But wait; didn’t we say in an earlier chapter that a String is immutable? Yes, we did. That means that once it is given a value, it can’t be given a new value.

You may be wondering a little about right now. Haven’t we created a String by concatenating a number of values together? Haven’t we done that in a series of statements? Consider, for example the toString method in the Course class.

```java
public String toString() {
    DecimalFormat myFormatter = new DecimalFormat("#.#");
    String result;
    double totalHours;
    totalHours = lectureHours + laboratoryHours + seminarHours;
    result = departmentAbbreviation + " " + courseNumber;
    result += "-" + credits + ";-"
    result += myFormatter.format(hours);
    result += '\n' + title + '\n' + description;
    result += "(" + myFormatter.format(lectureHours)
    result += "," + myFormatter.format(laboratoryHours)
    result += "," + myFormatter.format(seminarHours)
    result += ");";
    return result;
}
```

Every time we add one String to the end of another we appear to be changing the value of result.

True, but we do by being wasteful of memory.

When we perform those statements in the method above, we use the value of result to create another variable. We then have result refer to that new variable. The original value is left in memory, but nothing is referring to it. At some later time, a process called “garbage collection” will be used to identify the memory which is not referred to, and return it to the pool of available memory.

Note that all the changes we made involved adding characters to the end of an existing value. While summarizing meeting times, I’m thinking of a process in which we construct a String as we read meeting details, perhaps modifying the String (by inserting new characters, or changing existing ones) when we read a different meeting detail. But a String won’t allow an insertion. We need something like a String but which one allows us to modify it.
How about a `StringBuffer`?
**StringBuffer**

The online documentation states that a `StringBuffer` is a “…mutable sequence of characters. A StringBuffer is like a String, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.” The word “mutable” is geek-speak for “changeable”. That’s what we want!

![Venn diagram](http://xkcd.com/747/)

In fact, when you read the documentation on StringBuffer you’ll find that the most recent versions of Java have included a new class, `StringBuilder`, which might be even better. We’ll use `StringBuffers` here and leave `StringBuilder`s as an exercise.

Let’s continue with the creation of `meetingsList`.

I am thinking we should create five `StringBuffers`, each of which will contain seven characters representing the days of the week (yes, classes may occur on Saturday and Sunday), a space (to make the result more readable), the four digit start time, a space (to make the result more readable), the four-digit end time, a space (to make the result more readable), and the room number for the meeting, as shown in the example above.

Thus each `StringBuffer` needs to contain 18 characters (seven plus one plus four plus one plus four plus one) plus the length of a room number (in the case of my college, four characters), a total of 22 characters.

The length of a room number may vary for a different college. For that matter, other values we have considered (the subject abbreviation and the course number) may be different for other colleges too.

Now we need an array which we can use to access the individual `StringBuffers`.

```java
// assumption - no more than five meetings a week
StringBuffer temp[] = new StringBuffer[5];
```
Then we need to initialize each element of the array to a string of 22 blanks.

```java
int row = 0;
for (row = 0; row < 5; row++)
    temp[row] = new StringBuffer("                      ");
```

There are 22 spaces (blanks) between the quotation marks in the last statement.

**Adjusting to different colleges**

But look, we have used the number five twice already (and the word five once.) It is better to avoid such use, and create a constant, and use it instead. What if you ever have to change the number? Will you remember that every five is the same or are some different from others?

In Canada, for a while we had two different sales taxes. The provincial rate where I live was 7%, and the federal (national) rate was also 7%. Then the federal rate dropped to 6%. Imagine how many programmers had to look at each use of 0.07 and see whether it was the provincial rate (which didn’t change) or the federal rate (which did.)

And then the federal rate changed again, to 5%!

And in the summer of 2010 the two rates were combined into one!

While we are thinking about avoiding repetition of numbers, let’s deal with the other constants we have.

We mentioned the length of the start and end time of a meeting, the length of a subject abbreviation, the length of a course number, and the length of a room number. Can we define those lengths as constants? If so, where?

Since they are college-wide limits, I would suggest we place these limits in the College class. Since we are using the singleton pattern and thus have only one instance of the College class, we can use getters as we have with all our classes. In an exercise at the end of the chapter, we will explore how to use resource bundles to customize these values for different colleges.

Here are the additions to the College class, with comments omitted to save trees.

```java
// college-wide constants
private final int MAX_MEETINGS_PER_WEEK = 5;
private final int MEETING_START_TIME_LENGTH = 4;
private final int MEETING_END_TIME_LENGTH = 4;
private final int ROOM_NUMBER_LENGTH = 4;
private final int DEPT_ABBREVIATION_LENGTH = 4;

public int getMaxMeetingsPerWeek() {
    return MAX_MEETINGS_PER_WEEK;
}
```
public int getMeetingStartTimeLength() {
    return MEETING_START_TIME_LENGTH;
}

public int getMeetingEndTimeLength() {
    return MEETING_END_TIME_LENGTH;
}

public int getRoomNumberLength() {
    return ROOM_NUMBER_LENGTH;
}

public int getDeptAbbreviationLength() {
    return DEPT_ABBREVIATION_LENGTH;
}

Of course the instance variables are private. As constants, which we know they are since they are declared to be final, their names are capitalized. Since long sequences of capital (uppercase) letters are hard to read, spaces within the names are replaced with underscore characters.

Now we can retrieve these constants wherever appropriate.

    // assumption - no more than maxMeetings meetings a week
    College c = College.getInstance();
    int maxMeetings = c.getMaxMeetingsPerWeek();
    StringBuffer temp[] = new StringBuffer[maxMeetings];
    // each element of the array is empty
    // create an array of StringBuffers of the correct length,
    // containing blanks. some of the blanks will be
    // replaced later
    int lengthNeeded = 7 + 1 +
        c.getMeetingStartTimeLength() +
        1 + c.getMeetingEndTimeLength() + 1 +
        c.getRoomNumberLength();
    for (int row = 0; row < maxMeetings; row++) {
        temp[row] = new StringBuffer(lengthNeeded);
        for (int j = 0; j < lengthNeeded; j++) {
            temp[row].insert(j, " ");
        }
    }

The array is created properly, for now and for the future, for the current college standards and for future standards.

Another for loop

What’s the for loop we have been using? It looks something like the for-each loops we’ve seen when processing collections but it’s different.
Rather than looping over all the elements in a collection, the for loop is controlled by a counter.

The logic here is:
- Initialize the counter (row in the first loop, j in the second) to the value specified after the first equals sign, usually zero.
- Check that the continuation condition is met (row < maxMeetings in the first loop, j < lengthNeeded in the second).
- Whenever the continuation condition is met, perform the statements in the body of the loop. Then increase the counter as shown (increment by one), and loop back to the check.
- When the continuation condition is not met, the loop is finished.

Thus row takes on the values of 0, 1, 2, 3, 4 and each corresponding element in the array is set to, in our case, 22 spaces.

**Processing all the meetings**

Now we need a way to visit all the Meeting objects and place them into the appropriate StringBuffer. A for-each loop is good here.

```java
for (Meeting mt: theMeetings) {
    // do something
}
```

**Manipulating StringBuffers**

Now let's explore how to transfer the data in a Meeting object into a line on the schedule.

**Placing data in StringBuffers**

The last snippet of code said we need to “do something” as we process each meeting. What is the something we need to do?

For the first Meeting object, we need to place its details into the first StringBuffer; we need to store the day of the week, the start and end times, and the room number.

For each subsequent Meeting object, we need to compare its details to the details already in the array of StringBuffers; when there is a match of room and times, we need to modify the matching StringBuffer to indicate a meeting on a different day. When there is no match to any of the existing StringBuffers, we need to begin building a new StringBuffer in the appropriate format.

How do we know we are dealing with the first Meeting object? Alternatively, how do we know we are not dealing with the first Meeting object?

In answering those two questions, you begin to review the datatypes you know to find one that takes on only two values. Right, boolean variables!

```java
boolean first = true;
```
for (Meeting mt: theMeetings) {
    if (first) {
        // do something
        first = false;
    } else {
        // do something different
    }
}

From the description of the output we’d like to create (seven characters for the days of the week, a space, the four-digit start time, a space, the four-digit end time, a space, and the room number for the meeting), you can replace the comment // do something with the following.

temp[0].replace(8, 11, mt.getStartTime());
temp[0].replace(13, 16, mt.getEndTime());
temp[0].replace(18, 21, mt.getRoomNumber());

StringBuffers are objects, so the syntax we need is the name of the object, followed by a period and the name of the method we want to invoke, followed by the parameters of that method.

In these cases, we are replacing four blanks in the StringBuffer with the values returned by the Meeting getters. We indicate which characters to replace by giving the index of the first one to be replaced, and the index of the last one to be replaced.

But the numbers 11, 13, 16, 18, and 21 are calculated on the basis of the lengths of the start and end times of a meeting, and the length of the room number. The number eight is calculated as one more than the number of days in the week.

Remember that the first position in the StringBuffer is position zero. Thus the abbreviations for the days of the week will be in positions zero through six. Position seven will be occupied by a space. Thus the first digit of the start time will be position eight.

We should calculate those numbers so they reflect differences between colleges.

// p1a is the location of the first digit of start time
// p1b is the location of the last digit of start time
// p2a ditto for end time
// p2b ditto for end time
// p3a ditto for room number
// p3b ditto for room number
int p1a = 8;
int p1b = p1a + c.getMeetingStartTimeLength() - 1;
int p2a = p1b + 2;
int p2b = p2a + c.getMeetingEndTimeLength() - 1;
int p3a = p2b + 2;
int p3b = p3a + c.getRoomNumberLength() - 1;
Thus when we place the times and room number into the array element, we should use these statements.

temp[0].replace(p1a, p1b, mt.getStartTime());
temp[0].replace(p2a, p2b, mt.getEndTime());
temp[0].replace(p3a, p3b, mt.getRoomNumber());

Recall that getStartTime and getEndTime both return ints. The lines above will work, but they don’t give the correct result. We are expecting the times to be four digits, but early morning times will be only three. (If we had stored the times as Strings we would not have this problem but would probably have others. In particular, we would need to ensure that all times contained the same number of characters.) We will use DecimalFormat (from the java.text.DecimalFormat library) to ensure the times are all four digits in length.

DecimalFormat myFormatter = new DecimalFormat("0000");

creates the DecimalFormat object and

temp[0].replace(p1a, p1b, myFormatter.format(mt.getStartTime()));
temp[0].replace(p2a, p2b, myFormatter.format(mt.getEndTime()));

uses it.

roomNumber is already a four-character string.

**Converting numbers to day of the week**

What about the day of the week? mt.getDayOfTheWeek returns an integer, so we need a way of converting and integer to the appropriate character representation of the day of the week.

We have seen that the String class provides a method called charAt. You provide the position within a String and charAt provides the character at that position.

Suppose we have

String dayAbbreviations = "SMTWRFA";

We have used R to represent Thursday (eliminating confusion with the T representing Tuesday) and A to represent Saturday (eliminating confusion with the S representing Sunday).

Then

dayAbbreviations.charAt(mt.getDayOfWeek()));

will extract the appropriate day letter, and

temp[0].setCharAt(mt.getDayOfWeek(), dayAbbreviations.charAt(mt.getDayOfWeek()));
will place it in the appropriate element of the first row in the array.

setCharAt works with individual chars; replace works with Strings.

Note that the method relies very heavily on the idea that the Calendar class uses the numbers zero through six to represent the days of the week. Should that implementation change, our program will cease working. This is an example of coupling, where one piece of code is very closely involved with another piece of code, in this case by knowing and using the internal details of that other piece of code. This is a situation we would normally avoid.

Since we have mentioned resource bundles and customization, we may wish to consider what would happen if the college we are modelling were in an area where English is not the spoken language. What would be the abbreviations for the days, and how would they be provided?

For example, the website http://italian.about.com/library/fare/blfare109a.htm says “The days of the week (i giorni della settimana) are not capitalized in Italian. The week begins with Monday.
lunedì—Monday
martedì—Tuesday
mercoledì—Wednesday
giovedì—Thursday
venerdì—Friday
sabato—Saturday
domenica—Sunday”

What are appropriate abbreviations for Tuesday and Wednesday?

**Processing second and subsequent meetings**

What happens as we process subsequent meetings?

Rather than developing a mammoth method, we are developing code snippets that we will combine to create a method which will scan through all generated rows, seeking ones (with the same start and end times, and room) to update. Should it not find one, then it needs to create another row.

How many rows have we generated? Well, we have one so far. We should remember that. Declare a variable before the loop begins

```java
int rowsGenerated = 0;
```

and give it a value after the first row has been generated.

```java
rowsGenerated = 1;
```
A lot of code

Now we have an idea of what we need to do to deal with all the Meeting objects. Take a deep breath as there is a lot of code in the following method, but you have seen it before, just in snippets.

```java
/**
 * Produce a list of meeting times.
 */
public String meetingsList() {
    College c = College.getInstance();
    int maxMeetings = c.getMaxMeetingsPerWeek();
    // assumption - no more than maxMeetings meetings per week
    StringBuffer temp[] = new StringBuffer[maxMeetings];
    // create an array of Strings of the correct length,
    // containing blanks.
    // some of the blanks will be replaced later
    int lengthNeeded = 7 + 1 +
        c.getMeetingStartTimeLength() +
        1 + c.getMeetingEndTimeLength() + 1 +
        c.getRoomNumberLength();
    for (int row = 0; row < maxMeetings; row++){
        temp[row] = new StringBuffer(lengthNeeded);
        for (int j = 0; j < lengthNeeded; j++) {
            temp[row].insert(j, " ");
        }
    }
    // remember if we are processing the first Meeting
    boolean first = true;
    // ensure times are four-digits
    DecimalFormat myFormatter = new DecimalFormat("0000");
    // abbreviations for days of the week
    String dayAbbreviations = "SMTWRFA";
    // number of rows of output we generate
    int rowsGenerated = 0;
    // p1a is the location of the first digit of start time
    // p1b is the location of the last digit of start time
    // p2a ditto for end time
    // p2b ditto for end time
    // p3a ditto for room number
    // p3b ditto for room number
    int p1a = 8;  // days in the week plus 1
    int p1b = p1a + c.getMeetingStartTimeLength() - 1;
    int p2a = p1b + 2;
```
int p2b = p2a + c.getMeetingEndTimeLength() - 1;
int p3a = p2b + 2;
int p3b = p3a + c.getRoomNumberLength() - 1;

// examine all Meeting objects
for (Meeting mt: theMeetings) {
    // extract fields necessary for comparison
    // done mainly for efficiency, so we don't need to extract the same field several times
    String st = myFormatter.format(mt.getStartTime());
    String et = myFormatter.format(mt.getEndTime());
    String rt = new String(mt.getRoomNumber());
    int dt1 = mt.getDayOfWeek();
    char dt = dayAbbreviations.charAt(dt1);
    // the first Meeting object is a special case
    if (first) {
        // simply place the Meeting details in the appropriate positions of the first element
        // of temp. Note that we use the replace method for Strings, and setCharAt for
        // a single char
        temp[0].replace(p1a, p1b, st);
        temp[0].replace(p2a, p2b, et);
        temp[0].replace(p3a, p3b, rt);
        temp[0].setCharAt(dt1, dt);
        // remember that we have generated a row
        rowsGenerated = 1;
        // and remember that we are no longer processing the first Meeting object
        first = false;
    } // end first
    else {
        // process all Meeting objects except the first
        // we'll be searching through the rows we have generated and need to know if we
        // find a match
        boolean found = false;
        // scan all existing generated rows
        for (int i = 0; i < rowsGenerated; i++) {
            // check things that need to match
            // IMPORTANT NOTE:
            // Substring has two parameters. The first is the start of the substring
            // you are looking for. The second is ONE MORE than the end position you wish
            // to consider.
            boolean matchStart = temp[i].substring(p1a, p1b + 1).equals(st);
            boolean matchEnd = temp[i].substring(p2a, p2b + 1).equals(et);
            boolean matchRoom = temp[i].substring(p3a, p3b + 1).equals(rt);
            // if everything matches, update the matching room to show another day of
            // the week
            if (matchStart && matchEnd && matchRoom) {
                // update the matching row
                temp[i].setCharAt(dt1, dt);
                // remember we had a match
                found = true;
                // and exit the for loop early
                break;
            } // end match found
        } // end for
    } // end else
} // end for
// if we didn't find a match, we need to create a new row of the output.
// found will still be false if we didn't find a match
if (!found) {
    temp[rowsGenerated].replace(p1a, p1b, st);
    temp[rowsGenerated].replace(p2a, p2b, et);
    temp[rowsGenerated].replace(p3a, p3b, rt);
    temp[rowsGenerated].setCharAt(dt1, dt);
    rowsGenerated++;
} // added new row
} // end except the first
} // end for all Meetings

// and finally combine all the generated rows into one large string
String result = "";
boolean needsReturn = false;
for (int row = 0; row < rowsGenerated; row++) {
    if (temp[row].length() > 0) {
        if (needsReturn)
            result = result + '\n';
        result = result + temp[row];
        needsReturn = true;
    }
} // end for all Meetings
return result;

This is the longest method we have examined so far; some would say it is too long. The Checkstyle extension says it is too long. We'll address that concern in a moment.

It uses a new method, the substring method for StringBuffers. This method allows us to access a portion of a StringBuffer. Note the comment in the code about substring. It’s a good idea to leave a note behind explaining subtleties in your code for those who follow afterwards.

We have also used the break statement as a tool to force and exit from a loop before it has completed its normal processing. This is often done while searching for something. As in life, there is no point in continuing searching once you have found what you sought.

**Programming Style**

Let’s step back a minute and look at the style in which the method has been written.

Does it contain too many statements? Some people will suggest that a method should fit on a page, or on a screen; here we take several printed pages.

Others suggest that a method should be *cohesive*. A method is cohesive when it carries out one task. Thus cohesion is at least partly in the eye of the beholder.
I would argue that the method is cohesive, since it summarizes a collection of Meeting objects into a String. Yes, there are several subtasks involved, but they are all essential to the main task; the subtasks probably would not exist independently of the main task. (See the discussion on composition and aggregation later in this chapter.)

Since the method is cohesive, I feel there is no need to decompose it into shorter methods. To check that a method is cohesive give it a name that describes what the method does. If the name requires the word “and” or the word “or”, then the method is probably not cohesive.

Are there too many comments? Some would argue yes, but I feel it is generally impossible to have too many comments, providing they are meaningful. An example of a useless comment is

```
i = 3; // assign the value of 3 to i
```

When you are using i to represent the month of March, then the comment should say that. Other simple statements, for example found = true; should be commented as it is perhaps unclear what the effect of the statement is.

Notice that declaration statements are scattered throughout the method. Some would argue that all declarations should be at the beginning of the method. Sometimes I agree, when it is a short method, but for longer methods, I prefer a just-in-time style where I declare variables when I need them.

Are the variable names meaningful? Guilty!

Some of the names are very terse (et, st, rt, dt1, dt) and could be better. However, my justification is that the names are used only within a small portion of the method (their scope is very limited) and they actually make sense; the two-letters names are temporary (hence the “t”) values for end, start, room, and day of the week. dt1 does not quite follow that pattern, but it follows the pattern we have established in unit tests. When tests are testing different aspects of the same thing, we use the same name but follow it with a digit. That’s what we’ve done here; dt and dt1 are both related to dates.

**Course contains Section**

Now we almost have a fully-functioning Section class. We can go back and resume adding Section objects to a Course object. And we can create a unit test to add some Sections.

How do you know that addSection works? Create a method, listSections, which will return a String showing all the sections (but not the students in the section) in a course.

Test it and examine the value it returns.

Did listSections list the sections in the expected order? What is the “expected order”?
At my college, lecture sections are numbered 001, 002, and 003, and should come first. Then lab sections are numbered L01, L02, followed by seminar sections which are numbered S01, S02. Fortunately sectionNumber is a String and Strings will sort into that order naturally, as we saw in the discussion on uppercase and lowercase letters.

However, we need a Comparator<Section> to do the work for us.

Make it so.

```java
/**
 * section number comparison.
 * @param s1  The first section
 * @param s2  The second section
 * The compare method returns negative, 0, positive
 * if Section s1 is <, =, or > Section s2
 * based on the sectionNumber
 */
public static final Comparator<Section>
SECTION_NUMBER_ORDER = new Comparator<Section>() {
    public int compare(Section s1, Section s2) {
        return s1.getSectionNumber().compareTo(s2.getSectionNumber());
    }
};
```

You would certainly need a different comparator if your college uses a different notation for the sections of a course.

As an aside, note that it doesn’t make sense to have a Section object without having an associated Course object. If we try to create such a situation, your program should throw an exception.

Thus the association between Course and Section is an example of composition. This is a type of association between a whole (the Course) and its parts (the Section) where the part cannot exist independently of the whole.

On the other hand, the association between Student and Section describing the sections in which a student is enrolled is an aggregation. This is a type of association between a whole (the Student) and a part (the Section) where the part can exist independently of the whole; the student exists even though he or she may not be registered in a section.

The reverse association between Section (the whole) and Student (the part) describing the students enrolled in a section is also an aggregation; the section exists whether or not students are registered in it.

**Mark, and Student contains Mark**

A student takes many courses at one time. One possible (poor) way to show this is to place a collection of Course objects in the Student class. Why is this solution poor?
How do we represent the courses a student took in the past? This presents an interesting problem.

What happens when the name of a course changes after a student completes it? The old name should appear on the student’s transcript. We also need to remember the mark the student earned in the course. Thus, it appears a good solution is to create a Mark object, which contains the information about the course as it was taken, plus the mark earned. We will explore the Mark class a little later.

A collection of courses represents the courses being taken in the current semester. A collection of marks represents the courses taken in the past, and the student’s performance in them.

Consider the transcript that will be produced for a student, at the end of a semester, at the end of the student’s studies at the college, or at some other time. That transcript will list all the courses a student has taken, some once, some more than once, organized by semesters.

Thus instances of the Mark class must contain the semester (perhaps the year and month, both integers, in which it started), the course taken (subject abbreviation, course name, course number, and credits. Credits will appear on the transcript and will be necessary to calculate the overall grade point average.), and the mark earned.

Then a Student object can contain a collection of Mark objects.

Follow our usual practice when creating a class.
- Use BlueJ to create the skeleton code for a new class, called Mark.
- Implement a constructor and a toString method.
- Implement unit tests for the getters and setters.
- Implement the getters and setters

Since we need a collection of Mark objects, we expect to need a Comparator (don’t forget the import statement!) and an equals method. The Comparator is based on the course and semester in which the mark was earned; the transcript shows semesters chronologically (usually oldest to most recent. The January semester comes before September, regardless of when the student started his/her studies.). The courses in a semester are listed alphabetically by subjectAbbreviation and courseNumber.

What does it mean to say that two Mark objects are equal? We can define equality to mean the subjectAbbreviation, courseNumber, semesterYear, and semesterMonth all match.

Implementing that equals method is left as an exercise for the reader.

**Professor contains Section**

As a Section object contains a collection of student identifiers, so a Professor object contains a collection of section identifiers. This represents the professor’s teaching assignment during the current academic year, and may include courses in two or three semesters. Since there is an ordering by date to these sections, we will be able to use TreeSets again.
A Professor object could also contain a second collection of section identifiers, representing the Sections the professor has taught in the past.

This second collection could be summarized to represent the courses the professor has taught at any time in the past.

Implementing the collections is left as an exercise for the reader.

**Summary**

That was quite a chapter. The coding was long and complicated, but it does show what you can do with a little time and care.

The next chapter continues processing collections.

Once you complete that chapter, you will have a solid understanding of how to represent collections of data.

Since the student has data which must persist over time (the marks he/she has earned in the past), and the professor has data which must persist over time (the sections and courses taught in the past), perhaps we should consider how to implement persistence. That is, how do we save data so that it will be there when we run our program again?

That will occupy a few chapters, starting in chapter 13, since there are many forms of persistence.
Exercises

1. Implement `alphaClassList` using an iterator, instead of a `for-each` statement.

2. Explore the Java documentation to determine the differences between `StringBuffer` and `StringBuilder`. How would these differences affect the code shown in the chapter?

3. In a previous chapter, we discussed modelling a bank account. Create a method which will allow you to display all the transactions for an account which occur between two dates which you specify.

   Use that method to create another method which will display the transactions within the last n days, where you specify the value for n.

4. In a previous chapter, we discussed modelling the sightings a birder makes. Create a method which will allow you to display all the sightings a birder has made between two dates which you specify.

   Use that method to create another method which will allow you to list the sightings in the current year.

5. The Java documentation tells us that “Resource bundles contain locale-specific objects. When your program needs a locale-specific resource, a `String` for example, your program can load it from the resource bundle that is appropriate for the current user's locale. Thus, you can write program code that is largely independent of the user's locale isolating most, if not all, of the locale-specific information in resource bundles.”

   While used for internationalization, resource bundles can also be used for other customizations. This chapter referred to different lengths for room numbers and course abbreviations. Explore the `ResourceBundle` class and see how you could use a resource bundle to allow for these two customizations.

6. Implement the `Mark` class.

   Implement the Student contains `Mark` association.

7. Implement the `Professor` contains `Section` association.

8. Explore the use of the modulus operator when either one or both of its arguments are negative.


10. Explore alternative ways of changing the time of a meeting.
Chapter 11 – Collections, part 4

Learning objectives

By the end of the chapter you will be able to

- Use many of the String methods
- Summarize collections using a variety of techniques

Introduction

We have seen many collections so far. Usually, we have added elements to them, and then converted them to a String for display.

In one case we have done more. We processed all the meeting objects associated with a section and merged them into something suitable for a timetable entry.

Now, I’d like to return to another collection, involving Mark objects.

Grade Average

Recall that each course has a credit value associated with it. We can calculate how many credits a student has earned in total by summing data in all Mark objects (eliminating duplicates) but we don’t yet know the total by semester. Let’s see how to calculate those numbers.

When do we want to know the number of credits completed in a semester? The obvious time is when we are producing a transcript, a record of a student’s progress in a semester. At the same time, we will be calculating an average mark. How do you calculate the average mark?

When to calculate an average

We have two types of average.

The first is the semester average, the average of all the courses taken in a semester, regardless of the marks earned. The second is the graduating average. The graduating average involves all courses taken towards graduation, taking the higher mark in case a student repeated a course, and eliminating failures.

At Okanagan College, the graduating average actually involves only the last 60 credits completed in a program. This has interesting issues in a program which consists of more than 60
credits and creates interesting programming problems as well. You’ll be glad to know that we
will not be pursuing that challenge, except in the exercises at the end of the chapter.

**Weighted average**

In either case, we calculate a weighted average. Create a weighted sum of the percentages (the
percentage earned in a course is multiplied by the number of credits the course is assigned) and
divide the weighted sum by the sum of the credits. The result of the division is the grade average.

The priority of operations in a programming language is the same as the priorities in
mathematics.

For example, assume that a student takes three courses at one time. In the first course, worth two
credits, the student earns a mark of 65%. In the second course, worth three credits, the student
earns a mark of 74%. In the third course, worth one credit, the student earns a mark of 81%.

At Okanagan College, the marks we submit are integers, as are the credits. But the division
below is not done as an integer division.

The weighted average is calculated as

\[
\frac{2 \times 65 + 3 \times 74 + 1 \times 81}{2 + 3 + 1}
\]

To evaluate that expression, you must first evaluate the numerator, then the denominator, and
then divide the numerator by the denominator.

For the numerator, calculate \(2 \times 65\) and remember the result (130). Calculate \(3 \times 74\) and
remember the result (222). Calculate \(1 \times 81\) and remember the result (81). Add together 130 and
222 and remember the result (352). Add 352 and 81 and remember the result (433).

For the denominator, add two and three and remember the result (five). Add five and one and
remember the result (six).

For the average, divide the numerator (433) by the denominator (six) and get a result of 72.17.
This is not an integer division, so the result has decimal places.

**The transcript method – original version**

Recall that we have a very simple transcript method in the `Student` class. You were asked to
complete it on your own. Let’s modify it to serve our needs. This is what I have.

```java
/**
 * create a transcript.
 * @return String representation of Marks
 */
public String transcript() {
```

242
Knowing when a semester changes

Recall that the collection of Mark objects is stored in a TreeSet, and the objects within that collection are stored in order by semester. Within a semester, the objects are stored by subject abbreviation and course number.

Whenever we produce a transcript, we will calculate an average of all the courses taken (which may be a graduating average) and we will calculate the semester average for each semester.

We need to be able to tell whenever a semester has changed. When a semester changes, we can calculate the semester average and reset various semester totals back to zero.

To determine when we have changed semesters, we simply compare the value of the year in the current Mark object to the value of the year in the previous Mark object. Should they be different, we have a new semester. If the years are the same, we compare the (starting) month of the current Mark object to the (starting) month of the previous Mark object. Should they be different, we have a new semester.

But what of the first Mark object? There is no previous Mark object. Hence, we assume the previous year is 0 (or some other impossible value) and the previous month is also 0 (or some other impossible value.)

Transcript – a skeleton for changes

Thus, we have the first modifications to the transcript method.

```java
public String transcript() {
    String result = "";
    int previousYear = 0;
    int previousMonth = "";
    int currentYear;
    int currentMonth;
    for (Mark m: theMarks) {
        currentYear = m.getSemesterYear();
        currentMonth = m.getSemesterMonth();
        if ((currentYear != previousYear) ||
            ((currentYear == previousYear) &&
             (currentMonth != previousMonth))) {
            // semester has changed
            // calculate averages and reset totals
```

```java
```
// display current Mark
result = result + m.toString() + '\n';

// calculate semester totals

return result;

} // display current Mark

Note the use of || to mean “or”. “When the year has changed or the semester has changed” becomes “If the years are not the same or the years are the same but the months are not the same” which becomes

if ((currentYear != previousYear) ||
    ((currentYear == previousYear) &&
    (currentMonth != previousMonth)))

Since the years and months are primitive datatypes, we use == and !.=.

**When the semester changes**

What calculations are necessary when the semester changes?

The grade average for the semester needs to be calculated and displayed, along with the number of credits earned for the semester. The semester totals need to be added to the student totals, and then the semester totals are reset to zero. We need to remember the year and month of the previous semester.

These English statements translate into the following Java statements.

// calculate grade average
semesterAverage = (double) semesterNumerator / semesterDenominator;

// display semester results
result += "credits earned = " + semesterCredits + " average = " + semesterAverage + '\n';

// add semester totals to overall totals
totalNumerator += semesterNumerator;
totalDenominator += semesterDenominator;
totalCredits += semesterCredits;

// reset semester totals
semesterNumerator = 0;
semesterDenominator = 0;
semesterCredits = 0;

// remember the previous semester
previousYear = currentYear;
previousMonth = currentMonth;

**Integer division**

With the exception of the semesterAverage, all these variables are ints. Recall our discussion on splitting an integer into pieces using `/` and `%.

When you divide any type of integer variable by another integer variable, your result is an integer. $4739/60$ is $78$, rather than the $78.98333$ my calculator shows.

When calculating an average mark, most students like to see at least one or two decimal places. To ensure that an integer (int) divided by an integer gives a result with decimal places, we cast (or convert) one of the integers to a double datatype and then do the division. When you have an expression involving a variety of datatypes, the Java compiler then takes over and converts them all to the “highest” datatype. Here, it converts the other integer (int) to a double and divides a double by a double, giving a double. You see this in the first statement.

```
semesterAverage = (double) semesterNumerator / semesterDenominator;
```

When you test the method, you will find that unfortunately the semester and overall averages appear with a varying number of decimal places. A DecimalFormat object using the pattern `##0.00` will resolve that. Read the pattern as “Display the digit in the hundreds column when it is not zero, otherwise leave a blank. Display the digit in the tens column when it is not zero or when it is zero and the digit in the hundreds column was printed, otherwise leave blank. Always display the digit in the units column. Always display the decimal point and two digits after it, rounding as appropriate.”

As I write these words (Spring 2006), the track and field world is discussing how a time of 9.766 seconds was mistakenly rounded to 9.76 instead of 9.77.

The rule usually used is “When the first digit to be dropped is more than five, drop it and increase the last kept digit by one. When the first digit to be dropped is less than five, drop it and leave the last kept digit unchanged. When the first digit to be dropped is exactly five, change the last kept digit to be even.”

Thus 9.766 rounds to 9.77 (first part of the rule), 9.773 rounds to 9.77 (second part of the rule), and 9.775 rounds to 9.78 (third part of the rule). In many cases, the third part of the rule is ignored or forgotten.

When the semester remains the same

We need to calculate and accumulate the totals for the current semester.

```
int percent = m.getMarkEarned();
int credit = m.getCourseCredits();
semesterNumerator += percent * credit;
semesterDenominator += credit;
semesterCredits += credit;
```

To count towards the graduating average, we need to omit courses which the student did not pass. Usually this means 50% is the minimum passing mark. When a student earns a mark of less than 50, the course is counted towards the semester average, but it does not count towards the credits earned.

How would we modify our model if different courses had different passing marks?

Then we need to display the information about the current Mark, as we did before.

```
result = result + m.toString() + '\n';
```

**The declarations**

Of course, we require the declarations of the variables.

```
String result = "";
int previousYear = 0;
String previousMonth = "";
int currentYear = 0;
String currentMonth = "";

int totalNumerator = 0;
int totalDenominator = 0;
double totalAverage;
int totalCredits = 0;

int semesterNumerator = 0;
int semesterDenominator = 0;
double semesterAverage;
int semesterCredits = 0;
```

**The overall average**

Once we have processed all the Mark objects associated with the student, we can display the final average.

```
totalAverage = (double) totalNumerator / totalDenominator;
result += "total credits earned = " + totalCredits + " average = " + totalAverage;
```
The transcript String – problems and deficiencies

Like much code which people write, the transcript method we have produced is partly correct, but not wholly correct.

First of all, the method attempts to display a semester average when it reads the first mark. That makes no sense.

We have already seen how to use a boolean variable to tell us when we are processing the first element of a collection. We can use the same technique to avoid printing an unnecessary subtotal.

Second, the method does not display the semester average of the final semester. Nor does it add the marks and credits for the final semester into the total. That’s an error; fix it.

Third, the vertical spacing is not very good, resulting in a transcript that is hard to read. Adding a few ‘\n’ characters to the output will solve that problem. Make it so.

Fourth, the summary for the semester follows the details for the semester. This may be acceptable for a printed transcript but sometimes (in the online transcript my college provides, for example), the summary comes first.

How do we tell the transcript method that the summary comes first sometimes and last other times? One way is to pass it a boolean variable which is, for example, true when the resulting String should have the semester summary appearing last, and false when the resulting String should have the semester summary appearing first.

A second way is to have two different methods, one for summary-first and one for summary-last.

You would be right in thinking these methods would repeat much of the code in the transcript method, but in some other order. That would not be sensible, since any errors in one method could be repeated in the other.

Instead, we will write a method which takes the String which the transcript method calculates and manipulate it so the details about the averages appear in the correct places.

The String class and its methods

I find it is a great deal of fun to manipulate a String. There are so many methods available for manipulating and rearranging Strings that you can do almost anything you want to Strings. Let's look at some of these methods before we continue generating our transcript.

What if we want to disassemble a string?

Suppose we have a string

String name = "Marmaduke Archibald Cholmondley";
If we wish to retrieve single characters, we use the `charAt` method.

```java
char firstInitial = name.charAt(0);
```

Since the characters which make up a string are numbered starting at zero, `firstInitial` now contains the character ‘M’. `charAt` does not remove characters from the original string.

If we wish to retrieve substrings, collections of consecutive characters, we use one of the two forms of the `substring` method.

```java
String firstWord = name.substring(0, 9);
String lastWord = name.substring(20);
```

The first retrieves the characters beginning at position zero, **up to but not including** the character in position nine. That is, `firstWord` will be the string “Marmaduke”. `substring` does not remove characters from the original string.

The second retrieves the characters beginning at position 20 and extending to the end of the string. That is, `lastWord` will be the string “Cholmondley”.

Of course, you can use a similar statement to determine the middle word.

```java
String middleWord = name.substring(10, 19);
```

How did we know which positions to use in the above statements? We counted characters. But there are alternative.

We could use the `split` method, which returns an array of `String`. Or we could use the `indexOf` method to help us find the blanks.

**Details of split**

The `split` method takes a string and returns an array of strings, breaking the original string at places which match the value of a “regular expression” which you specify. A regular expression is a pattern, possibly including wildcards, of letters, numbers, and/or punctuation.
The following statements will break the variable name into its pieces and display them.

Pattern p = Pattern.compile("\s");
String[] names = p.split(name);
System.out.print("broken into strings, " + name + " consists of [");
for (int i = 0; i < names.length; i++)
    System.out.print(" " + names[i]);
System.out.println");

Recall that previously we have used \n and \t to represent a new line and a tab, respectively. The regular expression we want to use consists of two characters, \s, which refers to any whitespace character, a blank, a tab, a newline, etc. What do we use to represent a backslash?
Right, \. To get both the backslash and the s, we use \s.

More details on regular expressions and patterns are in the Java documentation describing the Pattern class.

**Details of indexOf and lastIndexOf**

The indexOf method comes in several forms, as did substring. The first starts seeking a specified character, looking from the beginning of the string.

```java
int n = name.indexOf('a');
```

returns the number 1, since the first character ‘a’ in name is at the second position. Remember that we number the positions starting at 0.

The statements

```java
n = n + 1;  // could use n++;
n = name.indexOf('a', n);
```

returns the number 4, the position of the next ‘a’ in the string. In this form of indexOf, the second parameter is the position at which we begin seeking the specified string, the first parameter.

Rather than seeking a character, we can seek a string.

```java
n = name.indexOf("duke");
```

returns the number 5 since the string “duke” begins in the sixth position of the string we are searching. We could also specify the position at which we begin seeking the string.

With any form of the indexOf method, when the string we are seeking is not in the string we are examining, the method returns -1.

**Why would it return -1? What is special about -1?**

If you wish to start at the end of the string and search towards the beginning, the String class supports a number of different lastIndexOf methods.

**Reversing a String**

As an exercise in using various String methods, write a method which will accept a string and then generate the string in reverse. That is, for an input of “here is a string”, the output will be “gnirts a si ereh”.

One possibility is to reverse a string using an iterative algorithm. That is, we could look at each character and process it.
public static String iReverse(final String s) {
    String result = "";
    for (int i = 0; i < s.length(); i++)
        result = s.charAt(i) + result;
    return result;
}

The method is declared static so that we can use it with any string.

A second possibility is to reverse the string using a recursive algorithm.

When we use a recursive algorithm, we formulate a problem in terms of one or more smaller versions of the same problem plus one or more smallest versions of the problem. Here, there are two smallest versions, a string which contains a single character and a string which contains no characters.

Both can be detected by looking at the length of the string. In the first case, it is one; in the second case it is zero. In both these smallest cases, the reverse of the string is the string itself.

public static String rReverse(final String s) {
    if (s.length() <= 1)
        return s;
    else
        return rReverse(s.substring(1)) + s.charAt(0);
}

The logic in the method is as follows:
- If there is zero or one character in the string, the string is its own reverse.
- If there are several characters in the string, then the reverse of the string is the reverse of all of it except for the first character, followed by the first character.

You may need to ponder the method for a while, since it is the first time we have used recursion. That is, it is the first time that we have seen a method call itself. We have seen many examples where one method calls another. Why should a method not be able to call itself?

Remember that recursion works because you are using a smaller version of the same problem, and there is at least one smallest version, for which we know the answer. You will see recursion again when we look at some mathematical methods in a later chapter.

**Palindromes**

While we are playing with strings, how about exploring palindromes?

The simplest definition of a palindrome is a sequence of characters that (not counting spaces, punctuation, and case) reads the same forwards and backwards. Some English words (civic, tot, level, a) are palindromes. Wikipedia has a discussion of palindromes in other languages and music, as well. [http://en.wikipedia.org/wiki/Palindrome](http://en.wikipedia.org/wiki/Palindrome)
Can we write a method that tells us whether or not a string is a palindrome?

Certainly. It uses one of the reverse methods we just created.

```java
public static boolean isPalindrome(final String s) {
    String target = "";
    // keep only those characters which are letters
    for (int i = 0; i < s.length(); i++) {
        char ch = s.charAt(i);
        if (Character.isLetter(ch))
            target = target + ch;
    }
    return target.equalsIgnoreCase(rReverse(target));
}
```

As you can see, most of the method involves taking out the characters that are not letters. We use a method from the Character class, isLetter, to identify the characters which are letters.

Determining that the string is a palindrome takes place in the return statement.

Try the method with some of the well-known palindromes.

- “Madam, I’m Adam.”
- “A man, a plan, a canal: Panama!”

Other palindromes are given in the Wikipedia article cited earlier.

Now that we have seen some of the String methods, we can go back to our original problem of producing a transcript.

**The transcript String – reordering**

But wait a minute! We’ve said on many occasions that Strings are immutable. That is, they cannot be changed. We have used statements like

```java
result += "total credits earned = " + totalCredits + " average = " + totalAverage;
```

which changes a String. And now we’ve seen how to disassemble strings. What’s going on here?

We explained this situation earlier, in less detail.

Yes, a String is immutable. However, a String is a reference to an area of memory which contains the contents of the String. Those contents may not be changed. But the reference itself may change, thus pointing to a different area of memory.

The way the assignment statement above could be implemented might be as follows.

- Copy the contents of the memory to which result refers to another area of memory (call it area 1) and append “total credits earned=” to the contents of area 1. Change result so its reference points to area 1.
• Copy the contents of area 1 to another area of memory (call it area 2). Convert totalCredits to a String and append that to the contents of area 2. Change result so its reference points to area 2.
• Copy the contents of result to another area of memory (call it area 3) and append “average =”. Change result so its reference points to area 3.
• Copy the contents of result to another area of memory (call it area 4). Convert totalAverage to a String and append that to the contents of area 4. Change result so its reference points to area 4.

But what about the areas of memory I have called area 1, area 2, and area 3? They now have no references referring to them, so they are eligible for garbage collection, a process by which unused (that is, unreferenced) areas of memory are made available for reuse.

So now you see how we have been able to construct Strings from other Strings, even though Strings are immutable.

How do we use this to manipulate the String that transcript has produced?

First we note that there are some special substrings we need to identify and move. These begin with “credits earned” and end with ‘\n’. (Be careful that you do not find the substring beginning “total credits earned”). Only these substrings need to be moved. All others remain in the same order.

Try to write the method transcriptSummaryFirst without reading on.

Here is my transcriptSummaryFirst method.

```java
/**
 * create a transcript String in which the summary comes before the details of the semester.
 * @return transcript with summary first
 */
public String transcriptSummaryFirst() {
    String result = "";
    String temp = "";
    int startPhrase; // where the phrase starts
    int endPhrase;   // where the phrase ends
    String semesterDetails = "";
    String summaryLine = "";

    // get transcript with summary last
    temp = transcript();

    // repeatedly search for the phrase "credits earned"
    startPhrase = temp.indexOf("credits earned");
    // at the end of the search, you find "credits earned" followed by
    // "total credits earned". startPhrase is 6 when this happens
    while (startPhrase != 6) {
        // extract everything up to but not including the phrase "credits earned"
        semesterDetails = temp.substring(0, startPhrase);
```


To understand how the method works, it may help to sit down in a quiet place and draw a series of diagrams to show the steps in the process.

**The transcript String – problems and deficiencies, continued**

Problem - The overall average is incorrect for a student who has repeated courses.

At my college, when a student takes a course twice (or more), its mark counts towards the semester average for each semester in which it is taken. But only the highest mark should be included in the overall graduating average. This will involve rethinking how the transcript method calculates the overall average.

Actually, the problem is even more complicated. We have “special topics” courses which may be taken more than once for credit, provided the topics are different. Accommodating these courses would require us to rethink our model quite seriously, so we will ignore this problem.

This emphasizes the importance of getting your model correct before you start implementing it.

Rather than saving the credits and percentages we need to calculate the overall average, we’ll create a collection that contains only the marks we’ll need. Then, once we have examined all the marks, we’ll calculate the average.

We need a collection in which we can store objects containing a subject abbreviation, course number, course credits, and mark earned. But we already have a class like that, Mark, except that Mark contains the year and month of the semester in which the mark was earned. Since the year and month don’t matter in calculating the grade average, can we use the Mark class?
Yes, we can. We can extend it to create a new class, `HighestMark`, that does not concern itself with the year and semester.

```java
class HighestMark extends Mark {
    public HighestMark(final String subjectAbbreviation, 
                        final int courseNumber, final int credits, final int markEarned) {
        super(subjectAbbreviation, courseNumber, 
               credits, 1999, "SOMETIME", markEarned);
    }
    public HighestMark(final Mark m) {
        super(m.getSubjectAbbreviation(), m.getCourseNumber(), m.getCourseCredits(), 
              1999, "SOMETIME", m.getMarkEarned());
    }
}
```

Since we don’t care about the year and semester, we just provide some values; the exact values we provide don’t matter!

That is all there is to the class. Well, it could be simpler, if we had just one constructor.

All the methods which we created for `Mark` will be available to `HighestMark` objects as well. That is, the child class `HighestMark` is derived from its parent class `Mark`, so its methods are inherited from its parent.

So what type of collection should we use as the basis of the collection of `HighestMark` objects? The order in which they are stored does not matter, but we would like to be able to check whether there already is a mark stored for the course. Many datatypes allow this, but we haven’t used an `ArrayList` for a while, so let’s use one now.

Recall that I have mentioned that at Okanagan College the graduating average is calculated from the last 60 credits completed. We are ignoring that wrinkle.

How would `HighestMark` be altered if we did want to use only the last 60 credits calculated?

We modify our transcript method by adding the following three statements.

```java
import java.util.ArrayList;

ArrayList< HighestMark > best = new ArrayList< HighestMark >();
HighestMark h;

Each time we process a Mark object, we convert it to a HighestMark object and compare it to other HighestMark objects.

h = new HighestMark(m);
// is there already mark for this course?
```
int i = best.indexOf(h);
// if so, i is zero or more. If not, i is -1
if (i == -1)
  // a mark for the course does not exist
  best.add(h);
else
  // a mark for the course exists
  if (best.get(i).getMarkEarned() < m.getMarkEarned())
    best.get(i).setMarkEarned(m.getMarkEarned());

This works because the indexOf method returns the position of the mark, if it exists, or -1, an invalid position, if it does not. We have seen the use of special values for invalid data when using Strings. Some String methods also return -1 to indicate a lack of success.

When we have processed all the Mark objects, we compute the numbers we need for the overall average.

    totalNumerator = 0;
    totalDenominator = 0;
    for (HighestMark b:best) {
      totalNumerator += b.getMarkEarned() * b.getCourseCredits();
      totalDenominator += b.getCourseCredits();
    }
    totalCredits = totalDenominator;

Now we have a transcript method that calculates the overall average correctly.

And what if we wish the summary lines of the transcript printed before the details? Nothing we have done has changed that method. But to be safe, run a unit test on it again. This is a practice called regression testing, making sure that a change you made has not broken anything else.

**Summary**

This chapter has focussed on manipulating strings. This is an important skill, as much processing involves string manipulation.

Now that we have experienced a serious amount of sophisticated Java code, it is time to go back and cover some of the things that have been glossed-over. In particular, we need to review exceptions.

Then we will look at persistence and then developing a graphical front-end on our program, so that we are not so dependent on BlueJ.
Exercises

1. How would you change the model to handle special topics courses?

   A student may take these courses more than once. In fact, the student may take the same special topics course more than once in a semester, as long as she/he is registered in sections which have different topics.. There may be more than one section sharing a common topic.

2. You can use the ideas from creating a transcript to create a class list. Of course class lists may be in order by the name of the student or by student number.

   Make it so.

3. How would you modify the calculations of the graduating average if it was based on only the final 60 credits completed in a program?

   Many programs have only three-credit courses, so “the last 60 credits” refers to 20 courses.

   What would happen if a program had a mix of one-, two-, and three-credit courses?

4. How would you modify the grade averaging process if your college assigned letter grades?

5. How would you modify the grade averaging process if different courses at your college had different passing marks?

   This may entail rethinking more than a small part of the model.

6. In previous chapters we have looked at the data a birder collects. Create a method to list all the birds seen.

   Make sure that the list tells how many birds have been seen each month, and how many each year.

   In addition, provide the number of species seen each month, and each year.

7. In a previous chapter we examined bank accounts and the collection of transactions they contain.

   Create a method to list all the transactions for a particular account, showing the total number of transactions in each month. Recently my bank has begun to show the account balance at the end of each day, as well.

   Make it so.
Chapter 12 – Exceptions revisited

Learning objectives

By the end of the chapter you will be able to:

- Define exception
- Define catch and try
- Identify when exceptions are thrown and describe how to catch them
- Create, throw, and catch your own exceptions
- Use exceptions to create programs which gracefully handle erroneous input and mistakes in computation

Definition

We have used and examined exceptions many times in the preceding chapters. There many other exceptions and we shall see some of them in the following chapters. These include ClassNotFoundException, IOException, and FileNotFoundException.

Before we see these exceptions, we should review the answer to the question “What is an exception?”

One source of the answer is Oracle’s Java tutorial. One of the trails through the tutorial takes you through essential classes and features. Exceptions are certainly essential, and are described at http://download.oracle.com/javase/tutorial/essential/exceptions/definition.html.

This section of the tutorial contains a number of quotations from the tutorial, as viewed on 2006-05-19. For example, we have the introduction:

“What Is an Exception?

“The term exception is shorthand for the phrase "exceptional event."

“Definition: An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

“When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when
the error occurred. Creating an exception object and handing it to the runtime system is called **throwing an exception**.

“After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible “somethings” to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the **call stack**.

“The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an **exception handler**. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

“The exception handler chosen is said to **catch the exception**. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, the runtime system (and, consequently, the program) terminates.”

**Examples**

When we attempt to clone an object, a `CloneNotSupportedException` may occur if cloning is not allowed. Normally, you would want to be able to clone any object. However, there are some cases where you want to ensure there is one and only one copy of an object.

An example would be in the College system we have been modelling, where we wish a college-wide policy to describe how percentages translate to letter grades. Everybody follows the same policy, so there should be only one copy. We accomplish the same result by using the Singleton pattern, which we will have already seen.

If you want to prevent cloning an object, you could use the following clone method.

```java
public SomeDataType clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException("Cloning is not supported by this object");
}
```

In the method we have created a `CloneNotSupportedException` with our own message attached to it.

**IOException**

Division by zero is another place that you may want to throw an exception. Fortunately, we have not seen one of those yet. We might have seen one if we tried to calculate a student’s grade
average for a semester in which the student was not enrolled in any courses or did not complete any courses.

When we developed the Meeting class, we wanted a way to identify incorrect or inappropriate times for the meeting. 2507 is an incorrect time. 0400 is a correct time, but it may be inappropriate. Perhaps the class should throw a DataFormatException. This exception, coming from a library used in compressing files, is one I find very useful when I don’t want to create my own exception. For incorrect times, we might need IncorrectTimeException and UnlikelyTimeException but I would probably just use DataFormatException.

When we were developing the Professor class, we wanted a way to ensure that a hiring date was specified. Perhaps we need to create our own type of exception, perhaps naming it MissingHiringDateException.

**Runtime and nonruntime exceptions**

As exceptions in Java are very important, there is a class named Exception and a number of subclasses derived from it. To see a (lengthy) list of the possible exceptions available in Java, check the documentation on Exception and its subclasses. That documentation contains the sentence “The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch.” Reasonable, indeed.

By the way, you may wish to look at the class Throwable. “The Throwable class is the superclass (or parent class) of all errors and exceptions in the Java language.” Many of the Exception methods we use are actually derived from Throwable.

Exceptions come in two flavours: runtime and nonruntime. The Java tutorial states

“Runtime exceptions occur within the Java runtime system: arithmetic exceptions, such as dividing by zero; pointer exceptions, such as trying to access an object’s members through a null reference; and indexing exceptions, such as trying to access an array element with an index that is too large or too small. A method does not have to catch or specify runtime exceptions, although it may.

“Nonruntime exceptions are exceptions that occur in code outside of the Java runtime system. For example, exceptions that occur during I/O are nonruntime exceptions. The compiler ensures that nonruntime exceptions are caught or specified; thus, they are also called checked exceptions.”

The CloneNotSupportedException is a runtime exception. IOException is a nonruntime exception.

The exceptions reporting incorrect and inappropriate time, or a missing hiring date, will be runtime exceptions, ones we create them ourselves.
Creating and throwing exceptions

Consider the constructor for the Meeting class, which we saw earlier.

```java
/**
 * Constructor for objects of class Meeting.
 * @param dayOfWeek the day of the week on which the meeting occurs
 * @param roomNumber the location of the meeting
 * @param startTime the time the meeting starts
 * @param endTime the time the meeting ends
 */
public Meeting(final int dayOfWeek, final String roomNumber, final int startTime, final int endTime) {
    this.dayOfWeek = dayOfWeek;
    this.roomNumber = roomNumber;
    this.startTime = startTime;
    this.endTime = endTime;
}
```

This constructor assumes it has been given clean input. But what could go wrong?

dayOfWeek could be an unreasonable value. The intent is that it is between zero (Sunday) and six (Saturday) inclusive but there is nothing to force those conditions. Thus, due to some programming error or malicious input, it may be negative or it may be seven or more. We can detect that, and throw an exception.

roomNumber is a String. It could be a String which does not represent an existing room. To detect that, we would need a database of rooms, which we do not have. But if we did, when the String provided is not in that database, we have a problem and should signal it by throwing an exception. But we will have to leave that exception for another time since we don’t have time in this textbook to see how to have a Java program talk to a database. We could, though, have the College object contain a collection of available rooms and ask the College singleton if it knows of the room.

startTime and endTime are both ints meant to represent time on a military clock, from 0000 to 0059, 0100 to 0159, ..., 2300 to 2359. Nothing prevents an errant program or a malicious user from specifying a value like 0875 or 4500. We can detect that, and throw an exception.

To create an exception, we have two choices.

- When there is an appropriate type of exception already available, we create an object of that type.
- When there is not an appropriate type of exception, we create our own exception.

When a bad value is provided to dayOfWeek it makes sense to use a DataFormatException, an existing type of exception. Thus we can use the following statements in the constructor, before we copy the values in the parameters to the instance variables.

```java
if ((dayOfWeek < 0) || (dayOfWeek > 6))
```
throw new DataFormatException("dayOfWeek may only be 0..6, inclusive.");

along with the appropriate import statement.

import java.util.zip.DataFormatException;

Normally you don’t create an exception and do nothing with it; you throw it to some other section of your program which acknowledges the problem and fixes it. Here, the Meeting object we desired has not been created and the portion of the program which asked to create it will need to determine the correct value for dayOfWeek before the Meeting can be created.

Thus, we need one more modification to the constructor, indicating that it throws an exception which must be handled in some other method.

public Meeting(int dayOfWeek, String roomNumber, int startTime, int endTime)
    throws DataFormatException

Now the constructor looks like this.

/**
 * Constructor for objects of class Meeting.
 * @param dayOfWeek the day of the week on which the meeting occurs
 * @param roomNumber the location of the meeting
 * @param startTime the time the meeting starts
 * @param endTime the time the meeting ends
 * @throws DataFormatException
 */

public Meeting(int dayOfWeek, String roomNumber, int startTime, int endTime)
    throws DataFormatException
{
    if ((dayOfWeek < 0) || (dayOfWeek > 6))
        throw new DataFormatException("dayOfWeek may only be 0..6, inclusive.");
    this.dayOfWeek = dayOfWeek;
    this.roomNumber = roomNumber;
    this.startTime = startTime;
    this.endTime = endTime;
}

You already have a unit test for an appropriate value of dayOfWeek. Now write two unit tests (one for dayOfWeek too small, and one for dayOfWeek too large) to check that the exception is properly thrown. The tests should be something like the following.

import java.util.zip.DataFormatException;

import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

/**
* The test class MeetingTest.

/**
 * The test class MeetingTest.
 *
 * @author rick
 * @version may
 */

*/
public class MeetingTest
{
    /**
     * Default constructor for test class MeetingTest
     */
    public MeetingTest()
    {
    }

    /**
     * Sets up the test fixture.
     * Called before every test case method.
     */
    @Before
    public void setUp()
    {
    }

    /**
     * Tears down the test fixture.
     * Called after every test case method.
     */
    @After
    public void tearDown()
    {
    }

    @Test
    public void testMeetingSmallDay() throws DataFormatException {
        Meeting m = new Meeting(-1, "L322", 1000, 1030);
        assertNotNull("was created", m);
    }

    @Test
    public void testMeetingLargeDay() throws DataFormatException {
        Meeting m = new Meeting(7, "L322", 1000, 1030);
        assertNotNull("was created", m);
    }
}
What happens when you execute these two tests?

We can also use a `DataFormatException` to indicate that the start or end times are incorrect or inappropriate, but it may be better to create our own, more meaningfully-named exceptions. Note that the `DataFormatException` we have created can tell us about two problems – that the input is too small, or that it is too large. But the same exception in thrown is both cases and, perhaps worse, we used the same message for both.

By providing our own messages (the parameter passed to the constructor), we can include the offending data in the message should we wish (and we should wish, since it is a good idea to provide as much information as possible about errors which occur.)

But it may be better, since our programs will be simpler, to create our own exceptions. Create a class named `BadHourException`. Its contents are

```java
/**
 * Exception for bad hours.
 */

@Author Rick
@version December 2010

public class BadHourException extends Exception {

/**
 * Constructor for objects of class BadHourException
 * @param msg The message associated with the exception
 */

public BadHourException(final String msg)
{
    super(msg);
}
}
```

Where do we import `Exception`?

We don't since it is part of the `java.lang` library and that library is imported for us, automatically.

Create a similar class for `BadMinuteException`.

We identify the need to throw these exceptions by modifying the `Meeting` constructor a little more. Since we need the same sort of processing for both the `startTime` and the `endTime`, we create a small helper method to do the processing. This helper method is a private method, callable only within the `Meeting` class. (When you use `javadoc` to generate the documentation for a class, private methods are not exposed.) It is static since there is only one set of rules for determining correct times.

```java
/**
 * check if a time value is acceptable
 * @param msg The message associated with the exception
 */
```
private static boolean isValidTime(int t) throws BadHourException, BadMinuteException {
    int hours = t / 100;
    int minutes = t % 100;
    if ((hours < 0) || (hours > 23))
        throw new BadHourException("For time " + t +
            " hours should be between 00 and 23 inclusive");
    if ((minutes < 0) || (minutes > 59))
        throw new BadMinuteException("For time " + t +
            " minutes should be between 00 and 59, inclusive.");
    return true;
}

When an exception is thrown, the processing in the current block of code ceases (unless there is a finally block, described below) and the method, its calling method, or a method before it in the calling stack, is expected to handle the exception.

We invoke the isValidTime method as follows.

if (isValidTime(startTime))
    this.startTime = startTime;
if (isValidTime(endTime))
    this.endTime = endTime;

Since isValidTime may throw exceptions and the constructor doesn’t handle them (and, in fact, the constructor expects the code that invoked the constructor to handle them), the constructor must indicate that the exceptions may be thrown.

public Meeting(int dayOfWeek, String roomNumber,
    int startTime, int endTime)
    throws DataFormatException, BadHourException, BadMinuteException

Write unit tests to attempt to create Meeting objects with bad times. You’ll need three tests; hours too small, hours too large, and minutes too large. Why don’t you need a test for minutes too small?

Where do you place the unit tests? Do they go in MeetingTest or in BadHourExceptionTest and BadMinuteExceptionTest. In the past, I would have answered MeetingTest. But now, with more experience, I would answer BadHourExceptionTest or BadMinuteExceptionTest, as appropriate.

Why have I changed my answer? It is possible to create so many tests that BlueJ will not display all of them when you right-click the test class. To eliminate the problem, I'm reducing the number of tests in a test class. One way to do that is to move unit tests to another appropriate class. A second way is to create multiple unit test classes for one class. To do that, right-click the class and create one unit test class. Then right-click the background of the class diagram and create a second unit test class.
Wherever the tests may be, perhaps the unit tests look like this.

```java
@Test
public void testMeetingLargeHour() throws BadHourException, BadMinuteException, DataFormatException {
    Meeting m = new Meeting(3, "L322", 2500, 1030);
    assertNull("was created", m);
}
```

Now we can create and throw exceptions. But how do we handle the exception when it occurs. That is what catch clauses do.

### Catching exceptions

In baseball, basketball, Canadian or American football, rugby, water polo, and lacrosse, when you throw the ball, someone else catches it (unless you have scored a goal). That’s the theory behind throwing and catching exceptions.

When the problem is something we can handle ourselves immediately, we place the code which could cause an exception inside a try block. In a general way, we show the structure as

```java
try {
    Some statements which can throw exceptions
}
catch (one type of exception) {
}
catch (another type of exception) {
}
```

Thus, we could modify the last test to look like the following.

```java
@Test
public void testMeetingLargeHour() {
    try {
        Meeting m = new Meeting(3, "L322", 2500, 1030);
        fail("meeting should not have been created");
    } catch (BadHourException bhe) {
        // this is the exception which should occur
        assertTrue(true);
    } catch (BadMinuteException bme) {
        fail("should not occur - " + bme.getMessage());
    } catch(DataFormatException dfe) {
        fail("should not occur - " + dfe.getMessage());
    }
}
```

Note the use of the fail method, from the assertion library, to indicate that the test has failed.
How would you handle a time like 10:30 p.m.? That is, what happens if someone wanted to be able to provide military time as well as civilian time? Can you create a method to handle such times?

**CloneNotSupportedException**

You have already seen a catch clause in the clone method. It is repeated below.

```java
/**
 * clone.
 * @return a copy of the Address object
 */
public final Object clone() {
    try {
        return super.clone();
    } catch (CloneNotSupportedException e) {
        return null; // can not happen
    }
}
```

Here, the CloneNotSupportedException may be thrown by super.clone(). Thus some piece of code must handle it. In this case, we handle it ourselves. Since we know it cannot happen, the processing we do is minimal.

To repeat, here the code in the catch block will never be executed. That is not the case for the majority of exceptions.

Remember that every exception which is thrown must be caught somewhere.

**Bad input data**

But suppose you have gathered the information to create a Course object, but there is some information which is missing or otherwise “bad.” Your program will look somewhat like the following.

```java
// gather information
Course c;
try {
    c = new Course(information);
    // do something with the course object
} catch (anException) { // what can you do?
    // You need to have the person/method which created the information correct it.
    // That can not be done here.
}
Rather than handling the exception yourself here, you need to pass it back to the calling method.

**Finally**

Recall that earlier in the chapter, we referred to a `finally` block. As described in the Java tutorial,

> “The finally block *always* executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break. Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.”

So far, we have seen no cases where there is “cleanup code” necessary, but we will see some soon.

Note that a finally block will be executed unless the try block contains `System.exit` or the Java virtual machine crashes.

Consider [http://download.oracle.com/javase/tutorial/java/nutsandbolts/branch.html](http://download.oracle.com/javase/tutorial/java/nutsandbolts/branch.html) for a broader discussion of the `continue`, `return`, and `break` statements.

**Summary**

Exceptions are the powerful technique Java uses to handle errors. We have used them many times and we will see many more exceptions in the chapters that follow.

The following two chapters involve writing data to external devices. This process uses exceptions extensively.
Exercises

1. Modify the constructor for Person so that it determines that both a first and a last name are provided and throws exceptions if either name is missing. In Indonesia, many people go by just one name. How could you determine that you are in a country where people use only one name?

There are many stories about programs that have difficulty with names, perhaps apocryphal. One story is about a payroll system in which last names with only one character were used for testing purposes. That worked well until a person with the last name A joined the organization.

A second story was about the person who didn’t have a long first and middle name. His name was something like R B Jones. The payroll system threw exceptions for one-character names. So the data entry people entered the name as R(only) B(only) Jones. The people who wrote the system stripped out unusual characters from the input. Thus R B Jones became Ronly Bonly Jones.

2. Modify the constructor for Person so that it throws one or more exceptions if there is a birth date provided but it is an inappropriate date.

3. Explore the Exception class to see what other types of exceptions are available to you.
Chapter 13 – Persistence, part 1

Learning objectives

By the end of the chapter you will be able to:

- Define persistence
- Implement persistence through object serialization

Warning to the reader

This chapter and the next explore some advanced concepts which many people would prefer to omit from an introductory programming course. However, given the model we are building, I feel it makes perfect sense to include the subject of persistence.

Of the two chapters, this one describes an older technique which is becoming less common as time goes by. Chapter 14 describes a technique which is being used more and more. If you need to skip a chapter, make it this one.

If you prefer to skip this chapter, and the next, you should be aware there is a small section in the mathematical chapters which you will need to omit should you omit the subject of persistence now.

Why persistence?

Unit testing provides a way to create and destroy objects, but only for the duration of the test. Yes, BlueJ allows us to place the objects it creates on the Object Bench, but they exist only until you recompile the object or exit BlueJ.

Suppose we wish to create several objects and have them available at a later time, perhaps the next day, after we have ended BlueJ, turned the computer off, turned the computer back on, and then restarted BlueJ. We need a way to create objects and then save and restore them so we can use them again. In short, we need to investigate ways to ensure the persistence of objects.

Persistence has many meanings, ranging through “doggedness” and “a tendency to continue.” In computing, persistence is closer to the second definition; it is the ability to save an object and retrieve it at a later time.
The random access memory (RAM) in a computer is volatile; when it loses power, it loses its contents. Thus RAM is not suitable for persisting data; you would need to ensure that the supply of electricity is uninterrupted. The ability to save data has been required since the earliest days of computing. Thus we have seen the use of non-volatile memory (punched cards, magnetic tapes, floppy disks, hard drives, CDs, DVDs, and memory sticks). There are programming languages (COBOL and RPG are examples) designed to simplify the processing of such data.

Java is a more generalized language, supporting many types of processing of data. It too requires the processing of data saved by other programs.

**What does “saving an object” mean?**

Since an object has identity, state, and behaviour, what does “saving an object” mean?

Saving an object involves converting the object’s state (the values of its instance variables) to a stream (a continuous sequence) of bytes and writing those bytes onto a medium which will not lose its contents when the power is removed.

When an object is restored, it will be given a name and will have behaviour (as specified in the class definition), and its state will be reconstituted from what was previously saved.

The terms used in Java are serialization and deserialization. Java allows us to serialize an object, thus granting it persistence. Serializing an object involves writing it to an external storage device. Deserializing an object involves reading it back from that external storage device.

Notice that there are other ways of saving data. The use of databases is possible, but that is much more complicated and we will not discuss it here.

**External devices**

Before we can look at serialization, we must consider the external devices we will use and their characteristics.

The external devices are mostly disk drives and the files they contain. These files must be opened, written to (or read from), and then closed. Files have names.

The external devices could also be other computers, local or somewhere else on the Internet. You can use serialization to send objects from one computer to another, in real time. We will not explore that option here; it is a topic for more-advanced study, but is the origin of the type of serialization we are considering in this chapter.

**Streams**

Writing to and reading from external devices is done through streams. You can think of streams as sequences of characters, perhaps including special characters.
So we need streams to which we can write and from which we can read and we need a way to associate a stream with an external device.

Which streams will we need for object serialization?

This technique requires writing objects to streams. Looking from the operating system level first, we find that FileOutputStream is a class in the java.io package which allows us to connect a stream to an actual file. As the documentation says, “A file output stream is an output stream for writing data to a File…”

Associated with the FileOutputStream will be an ObjectOutputStream, another class available in the java.io package. As the documentation says, “An ObjectOutputStream writes primitive data types and graphs of Java objects to an OutputStream.” Think of a graph as the instructions for recreating the structure of the object.

Methods in ObjectOutputStream allow us to translate the objects into a stream, and the FileOutputStream connects that stream to a file on the underlying operating system.

Thus, writing an object to an external device requires several steps.

Similarly, we can use FileInputStream and ObjectInputStream to retrieve and reconstruct the objects.

Now we are ready to see how to do object serialization.

**Persistence via object serialization**

Note that we can only serialize (save) and deserialize (restore or rebuild) objects all of whose components are serializable. Thus, before we can serialize a Person object, for example, we need to ensure that all its instance variables are serializable. A Person object contains

- Several Strings which are all serializable by design. If you want to see if a class is serializable, look into the documentation and see if it implements the Serializable interface. If so, it is serializable. If not, it isn’t.
- An Address object which is not yet serializable.
- A MyDate object which is not yet serializable.

Let’s begin by considering the Address class and making it serializable.
Object serialization – Address

For an object to be serializable its class must implement the Serializable interface as noted above and all its instance variables must be serializable. All the instance variables of Address objects are Strings.

Examine the Java documentation to confirm that String does implement the Serializable interface.

While you are looking in the Java documentation, look at the Serializable interface itself for its details.

Serializable is what is called a marker interface; it says something about the class which implements it, but it doesn’t bring along methods and instance variables. In some cases (the instance variables are single-valued, not collections, and are all serializable), implementing the Serializable interface is all you need to do to make a class serializable. Address is such a class.

To make Address serializable, simply add a statement at the beginning of the class.

```java
import java.io.Serializable;
```

and modify the class heading to indicate the class implements the Serializable interface.

```java
public class Address implements Cloneable, Serializable
```

That’s it! Due to the simple nature of the instance variables of Address, you need do nothing else to make an Address object serializable.

You’re ready to create a unit test to confirm that an address is serializable.

Begin by creating two Address objects within the Address unit test, in the setUp method.

```java
Address a;
Address a1;
```

```java
a = new Address("1234", "X", "Fifth", "Avenue", "SE", "Calgary", "AB", "Canada", "T0L0M0");
a1 = new Address("1000", "", "KLO", "RD", "", "Kelowna", "BC", "CA", "V1Y4X8");
```

Here is the unit test method to check that we can serialize these two Address objects. It will attempt to write out the two objects and then check that two objects were really written. The deserialization test will check that the correct data was written.

```java
@Test
public void testSerialization(){
    int i = 0;
    try {
        FileOutputStream out = new FileOutputStream("address.dat");
        ObjectOutputStream oos = new ObjectOutputStream(out);
```

// write one object and remember that we wrote one
oos.writeObject(a);
i++;

// write a second object and remember that we wrote two
oos.writeObject(a1);
i++;

// output may be held in memory until a buffer is full. We have no more output so ensure
// the buffer is written to a disk somewhere
oos.flush();
// we don’t need the stream any more
oos.close();

} catch (FileNotFoundException e) {
   System.err.println("problem with address.dat " + e.getMessage());
} catch(IOException e) {
   System.err.println("problem with ObjectOutput " + e.getMessage());
} // check that we were able to write two objects
assertEquals(i, 2);

Note that in order to use serialization we must also use exceptions, which we saw in earlier chapters.

For the unit test to compile, we need to include the statements

import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.IOException;

But it is easier, when you need to import many classes from one package, to just use

import java.io.*;

The asterisk is a wildcard. A wildcard means “any value that matches the pattern.” In this case, java.io.* means “every class in the java.io package, but not any packages within java.io.”

Checkstyle will complain if you use a wildcard to import classes.

The unit test uses the writeObject method to actually write an Address object. The writeObject method is a method in the ObjectOutputStream class.

Since an Address is made of serializable objects and Address implements Serializable, an Address knows how to serialize itself.
Here’s a suggestion. Before you run any methods which use exceptions, make sure that you open the BlueJ Terminal window, so that the messages displayed in the catch clauses have a place to display, should there be an error of some kind. Select View, Show Terminal, or press \(\text{Ctrl}+\text{T}\).

Here is a unit test to check that we can deserialize Address objects.

```java
@Test
ggpublic void testDeserialization() {
    try {
        FileInputStream in = new FileInputStream("address.dat");
        // we use the same file name as in writeObject
        ObjectInputStream ois = new ObjectInputStream(in);
        a1 = (Address)ois.readObject();
        a = (Address)ois.readObject();
        ois.close();
    }
    catch (FileNotFoundException e) {
        System.err.println("Can not find address.dat "+ e.getMessage());
    }
    catch (IOException e) {
        System.err.println("file problem with ObjectInput "+ e.getMessage());
    }
    catch (ClassNotFoundException e) {
        // will not happen but must be caught
    }
    assertEquals(a1.getCity(), "Kelowna");
    assertEquals(a.getCity(), "Calgary");
}
```

The `readObject` method is a method in `ObjectInputStream`. It reads an object from the stream (throwing an exception to indicate an error when the stream cannot be opened), and we must then indicate the type of object, by casting to an Address.

Note that any method which casts an Object to some other datatype must handle a possible `ClassNotFoundException`. Usually the exception will not happen, but the casting process requires us to handle that possibility.

Both of these tests succeed. But note there is something unusual going on here.

We have talked about saving and then restoring an object. If you look closely at the code, you’ll see that we serialized the two Address objects in one order (a and then a1), but restored them in the reverse order (a1 and then a.) We can do this because both are Address objects. If we were to serialize a collection, then we would have many objects in the stream; when we recreated the collection, their order would be restored.

Note a second unusual feature in the method. We have used two assertions, rather than the one we have used until now. This is acceptable; the unit test will fail if any of the assertions fail. This
use of two or more assertions in one test is useful when the processing to set the initial conditions for the test are complicated and/or long.

But serialization is not designed to work quite like this, with one method serializing and another immediately deserializing. The deserialization takes place at a later time, after all the existing objects become null, or have ceased to exist.

**Object serialization – MyDate**

Recall that we began by discussing how to serialize a Person object. A Person object contains Strings (all serializable, by design), an Address (now serializable), and a MyDate object (not yet serializable, but we know how to do that.)

Make the MyDate class serializable. Since a MyDate object only contains single-valued instance variables, all this involves is asking the class to implement the Serializable interface.

Create the unit tests to show the serialization and deserialization of MyDate objects works.

**Object serialization – Meeting and Mark**

You may have implemented these two classes. If so, they are like Address and MyDate, as they contain only single-valued instance variables with no collections. It is very easy to make them serializable.

Make it so.

Note that it is not necessary to make HighestMark serializable, since objects of that class are never saved. They are used only while creating transcripts.

**Object serialization – Person, Student, Professor**

Assuming the code you created to serialize MyDate (and perhaps Mark) works, all the instance variables of a Person are now serializable.

But that fact is not particularly useful to us right now. The Person class is abstract; we cannot create instances of a Person object. Person is the parent class of Student and Professor and we can create instances of Student and Professor. Are they serializable? No.

This is because they each contain collections. We will explore how to deal with the collections in a few moments.

**Object serialization – some subtleties**

The readObject and writeObject methods we have used so far work well for simple situations. In particular, they work when there are no static or transient variables (a fact not already mentioned), and when there are no collections (a fact already mentioned).
static instance variables persist from one method call to another.

static class variables are accessible to all objects of the class. We have seen some static constants, in the College class.

As another example, consider the interest rate paid on all bank accounts. That will be implemented as a static class variable, since the rate changes from time to time but is common for all bank accounts.

But writeObject does not serialize static instance variables and thus readObject cannot deserialize them. When we have static variables, we will need to create our own methods to read and write objects.

transient is a reserved word, used to identify variables whose state should not be saved, usually because they can be calculated (like iterators or retirement dates), or because they require special handling (like encryption). We will need to deal with such variables on our own.

We can serialize static variables ourselves, using the appropriate write methods of the ObjectOutputStream class and the read methods of the ObjectInputStream class.

We can encrypt variables using the java.security package, a topic which we will not discuss further here.

We can handle collections on our own, too. Like encryption, to use that technique we will need to write some extra methods. In particular, we will need to provide replacements for writeObject and readObject.

**Object serialization – creating writeObject and readObject methods**

Some classes contain collections. The standard readObject and writeObject methods won’t work with those classes so we must override those methods with our own.

How do we create our own readObject and writeObject methods?

The signatures of these methods are specified in the ObjectOutputStream documentation as follows.

“Classes that require special handling during the serialization and deserialization process must implement special methods with these exact signatures:

```java
private void readObject(java.io.ObjectInputStream stream)
    throws IOException, ClassNotFoundException;
private void writeObject(java.io.ObjectOutputStream stream)
    throws IOException
```

Assuming we have the appropriate import statements, we can just use `ObjectInputStream` instead of `java.io.ObjectInputStream` and `ObjectOutputStream` instead of `java.io.ObjectOutputStream`, so the skeleton for the `writeObject` method we need to create is as follows.

```java
private void writeObject(ObjectOutputStream oos) throws IOException {
    // write instance variables
}
```

The comment indicates where we write the instance variables. My style is to write out the single-valued instance variables, the ones which are not collections, first. Then I write the collections, first writing the number of elements in the collection and then the elements of that collection, should there be any.

The only place that we should have a collection of `Student` objects is the `College` class. If we are serializing that class, we could use the following statement.

```java
oos.writeObject(theStudents.size());
for (Student s:theStudents) {
    oos.writeObject(s);
}
```

where `oos` has been opened earlier. This assumes `Student` objects are serializable; they will be soon.

The `Course` class contains a collection of student identifiers, corresponding to the students in the class. If we are serializing that class, we could use the following statement.

```java
oos.writeObject(theStudents.size());
for (String s:theStudents) {
    oos.writeObject(s);
}
```

where `oos` has been opened earlier. This assumes student identifiers are serializable; they are since they are `Strings`.

Similarly, we create our own `readObject` method. This is a generic method and needs to be modified for each class in which we use it.

```java
private void readObject(ObjectInputStream ois) throws IOException {
    try {
        // read instance variables here
    }
    catch (ClassNotFoundException e) {
        System.err.println("Unable to deserialize " + e.getMessage());
    }
```
To use the method, in the `College` class, we could use, for example,

```java
Set<Student> theStudents = new HashSet<Student>();

int n = (Integer) oos.readObject();
for (int i = 0; i < n; i++) {
    theStudents.add((Student) oos.readObject());
}
```

Any special handling required by transient or encrypted variables will need to be included in the method as well.

As we have seen in the examples above, both the `readObject` and the `writeObject` methods will be called by other methods, which look after opening and closing the streams, as well as extracting the objects and saving them with unique identities, perhaps as members of a collection.

Let’s see how this applies to the classes we have developed, in particular, to the ones which contain collections.

**Object serialization – Section and Student**

Due to the similarities between these two classes we’ll discuss them together.

Section is a class which contains two collections. One (theStudents) is a collection of student identifiers (serializable. Why?); the other (theMeetings) is a collection of `Meeting` objects (which we just made serializable).

Student is also a class which contains two collections. One (theSections) is a collection of section identifiers (serializable. Why?); the other (theMarks) is a collection of `Mark` objects (which we just made serializable). Student is derived from `Person`, which is serializable.

For both of these classes, we use the same logic, illustrated below for the `Section` class.

Following the idea of creating the unit test first, we have the following method in `SectionTest`.

```java
@Test
public void testObjectSerialization() {
    int i = 0;
    try {
        FileOutputStream out = new FileOutputStream("sections.obj");
        ObjectOutputStream oos = new ObjectOutputStream(out);
        System.out.println("s2 = " + s2.toString());
        oos.writeObject(s2);
        i++;
        System.out.println("s1 = " + s1.toString());
        oos.writeObject(s1);
        i++;
    }
    finally {
        System.out.println("i = ");
        System.out.println(i);
    }
}
```
Now we can create the writeObject method itself, in the Section class. Much of it is taken up with processing the collections which are part of the object.

```java
private void writeObject(ObjectOutputStream oos) throws IOException {
    // write the single-valued instance variables
    oos.writeObject(identifier);
    oos.writeObject(subjectAbbreviation);
    oos.writeObject(courseNumber);
    oos.writeObject(sectionNumber);
    oos.writeObject(startYear);
    oos.writeObject(startMonth);
    oos.writeObject(endYear);
    oos.writeObject(endMonth);

    // write the collections
    oos.writeObject(getStudentCount());
    if (getStudentCount() > 0)
        for (String s : theStudents)
            oos.writeObject(s);

    oos.writeObject(howManyMeetings());
    if (howManyMeetings() > 0)
        for (Meeting m : theMeetings)
            oos.writeObject(m);
}
```

Note that we have two writeObject methods. One appears in testObjectSerialization as `oos.writeObject(s2);`

The other appears as a private method in the Section class, as

```java
private void writeObject(ObjectOutputStream oos) throws IOException
```

The first is automatically translated into a call to the second. Run the unit test and confirm that you can serialize a Section.

Now you can serialize a section, but how do you deserialize it?
Object deserialization – Section and Student

First we create the unit test for the readObject method.

```java
@Test
public void testObjectDeserialization() {
    try {
        FileInputStream in = new FileInputStream("sections.obj");
        ObjectInputStream ois = new ObjectInputStream(in);
        s1 = (Section)ois.readObject();
        s2 = (Section)ois.readObject();
        ois.close();
    } catch (FileNotFoundException e) {
        System.err.println("Can not find sections.obj " + e.getMessage());
    } catch (IOException e) {
        System.err.println("file problem with ObjectInput " + e.getMessage());
    } catch (ClassNotFoundException e) {
        // will not happen but must be caught
    }
    assertEquals(s2.getSubjectAbbreviation(), "COSC");
    assertEquals(s1.getSubjectAbbreviation(), "PHYS");
}
```

Finally, we create the readObject method itself. Much of it is again taken up with processing the collections which are part of the object.

```java
private void readObject(ObjectInputStream ios)
    throws IOException, ClassNotFoundException {
    // single-valued instance variables
    this.identifier = (Integer)ios.readObject();
    this.subjectAbbreviation = (String)ios.readObject();
    this.courseNumber = (Integer) ios.readObject();
    this.sectionNumber = (String) ios.readObject();
    this.startYear = (Integer) ios.readObject();
    this.startMonth = (Integer) ios.readObject();
    this.endYear = (Integer) ios.readObject();
    this.endMonth = (Integer) ios.readObject();
    // collection instance variables
    this.students = new HashSet<Student>();
    int n = (Integer) ios.readObject();
    for (int i = 0; i < n; i++)
    {
        Object o = ios.readObject();
        theStudents.add((String) o);
    }
}
```
this.meetings = new TreeSet<Meeting>(Meeting.TIME_ORDER);
    n = (Integer) ios.readObject();
    for (int i = 0; i < n; i++)
    {
        Object o = ios.readObject();
        theMeetings.add((Meeting) o);
    }

Note that we again have two readObject methods. The one in the testObjectDeserialization method appears in s1 = (Section)ois.readObject();
The one in the Section class is readObject(ObjectInputStream ios)
Again, the first is translated into the second.

Note also that we have used statements like this.
	n this.identifier = (Integer) ios.readObject();

This appears a little unusual, since the left side of the assignment statement is a primitive
datatype and the right side is an object. Two new concepts were added to Java beginning with
Java 1.5.0 (also known as Java 5), autoboxing and autounboxing.

In many places we need to convert primitive datatypes like int to their corresponding object, here
Integer. When would we need to do this? Well, collections contain objects, not primitive
datatypes.

We could convert an int variable i to an Integer object iObject with the statement

    Integer iObject = new Integer(i);

and then insert iObject into the collection.

We could extract the int from the object with the statement

    i = iObject.intValue();

Autoboxing carries out the first statement for us and thus makes it easier to convert primitive
datatypes to the corresponding object.

Autounboxing carries out the second statement for us and thus makes it easier to extract
primitive datatypes from objects.
Programming style

Should you have one large data file which contains many objects of different types and you don’t know the order of the types, you may extract objects one at a time from the file and then use instanceof to determine the datatype of the object before you cast it appropriately. Such use gives rise to very complicated programs, and it is not advisable. In fact, many programming standards forbid the use of instanceof. We will follow those standards.

Instead, it is better to have an idea ahead of time of the structure of the file; in particular, it is important to know that the file contains objects of only one class. We will do so.

Object serialization – Section and Student, revisited

We may be serializing a class when its collections are empty. For example, we may be serializing a section before it has its meeting times determined and/or before students have registered. It is mid-May as I review this portion of the chapter; the sections are all in place for the next semester and their meeting times have been set. But the students don’t begin registering until next month.

Similarly, a course may not have any sections, particularly before the scheduling of those sections occurs.

Our way of writing a collection (record the number of items in the collection, and then the items themselves, if any) accommodates this nicely. We simply serialize a zero if the collection is empty.

Object serialization – Professor

Since Professor is derived from Person (which is serializable), and any additional instance variables (birthDate and retirementDate) are serializable (why?), Professor objects are serializable.

We mentioned adding some collections to a Professor object. The collection of current sections taught is a collection of section identifiers, Strings. The collection of courses taught in the past is a collection of Strings. The collections require separate processing, not because they contain Strings, but because they are collections.

Make Professor serializable, using individualized writeObject and readObject methods because of the collections.

Object serialization – College

Now that all the classes we have built are serializable, we can make the College class serializable. It contains various collections so we had to wait until all their elements are serializable, and they finally are.
Make it so.

**Summary**

In the chapter we have seen one way to save the state of our objects. The problem with the particular technique is that it produces files which are only readable by another Java program. In many cases it would be better if the file could be read by a program written in a language other than Java. Why would we want to do so? Perhaps so that we could create a webpage from the data we read. Perhaps so that we could create printed material from the data we read.

We’ll explore a technique which allows this in the next chapter.
Exercises

1. Make the Meeting class serializable.

2. Make the Mark class serializable.

3. Make the MyDate class serializable.

4. If you have implemented a Department class, make it serializable.

5. In a previous chapter we discussed modelling bank accounts. Is the Transaction class serializable? If not, make it so.

   Make the BankAccount class serializable.

6. In a previous chapter we discussed modelling the sightings a birder makes. Make the class Sighting serializable.

7. In a previous chapter we discussed modelling playing cards. Make the AngloAmericanPlayingCard class serializable.

8. Explore the java.security package to see how you can encrypt data when you serialize it.
Chapter 14 – Persistence, part 2

Learning objectives

By the end of the chapter you will be able to:

- Define XML and describe its characteristics
- Implement persistence using XML

Persistence via XML

Wouldn’t it be great to have a way to write a file so that it contained both data and a description of that data?

What if that file could be read by many different programs, written in different languages?

And wouldn’t it be great if a file were also human-readable?

There is a way to do all that, and it involves the use of XML, the Extensible Markup Language.

Wikipedia (http://en.wikipedia.org/wiki/XML) defines XML as follows: “The Extensible Markup Language (XML) is a W3C-recommended general-purpose markup language for creating special-purpose markup languages, capable of describing many different kinds of data. In other words: XML is a way of describing data and an XML file can contain the data too, as in a database.”

We are particularly interested in the “in other words” portion of that definition, “XML is a way of describing data and an XML file can contain the data too”, since that is what we wish to do.

The serialization described in the previous chapter allows us to store the structure of objects and to remember the data they contain. But it is a less-common technique, so you may have skipped that chapter. If so, there is no problem. Here we will see that XML serialization is an alternative way to do this and much more.

Why would we wish to use XML? The files containing XML are plain text files. Thus you can examine them in a text editor. You can write programs to process the files containing XML in languages other than Java. You can display XML files on the web.
A Digression – Scalable Vector Graphics

Scalable Vector Graphics (SVG) files are an interesting application of XML files. These contain the instructions to paint a picture. What sort of picture? It could be as simple as a bar graph, or it could be an animated movie. It could be a logo.

You could create a picture using something as simple as Notepad or as complicated as Adobe Illustrator. There are even pictures you can create using a Java program.

Most modern browsers will display SVG graphics. The advantage of them is that they are very small. For more details on SVG files, search the web for terms like SVG and create.

A Second Digression – PDF

Recently I have been working with dynamic PDF files. You may be used to static PDF files, which cannot change their appearance. Dynamic ones, on the other hand can change their appearance in response to data the user enters.

For example, many of my projects display certain portions of the document only when necessary. If you enter a field which contains too many characters, it asks you for a shorter form.

Underlying these PDF documents is an XML file.

Using an XMLEncoder – no collections

Back to persistence.

To translate an object into the appropriate XML statements (some of which describe the instance variable and others of which describe the actual value), we will use an object of the XMLEncoder class, which will be associated with a file. To reconstitute the objects using the data and instructions in a file we will use an object of the XMLDecoder class.

Using an XMLEncoder is similar to serialization with its writeObject method. An XMLEncoder translates an object into the instructions necessary to restore it and provides the value of the object’s instance variables to be used when restoring it.

Using an XMLDecoder is similar to deserialization with its readObject method. An XMLDecoder processes the instructions the decoder created.

Both XMLEncoder and XMLDecoder are in the java.beans package so we need to import them into any class which will be using XMLEncoder and XMLDecoder.

import java.beans.XMLEncoder;
import java.beans.XMLDecoder;
Note that these classes are designed for use with JavaBeans, classes which have a no-parameter constructor and setters for all instance variables. Our classes are not JavaBeans, but we’ll convert them to JavaBeans in a moment.

Here’s a method within the Address class which translates an Address object into XML. Similar methods work with all of our classes which do not contain collections.

```java
/*
* persist an Address object.
* @param e the XMLEncoder to be used
*/
public void writeObject(XMLEncoder e){
    e.writeObject(this);
} // end writeObject
```

This method is named writeObject and it calls a method also named writeObject. The class Address contains a writeObject method. To carry out its task, it delegates responsibility to the writeObject method from the XMLEncoder class. These methods, while sharing the same name, are different because they are in different classes.

Since the method we created is very short, and uses an XMLEncoder which has miraculously appeared from nowhere, there must be some other code elsewhere. It’s in the unit test.

```java
@Test
public void testXMLSerialization() {
    Thread.currentThread().setContextClassLoader(getClass().getClassLoader());
    try {
        XMLEncoder e = new XMLEncoder(
            new BufferedOutputStream(
                new FileOutputStream("address.xml")));
        // the variable a1 is an Address and has been defined in the setUp method
        a1.writeObject(e);
        e.close();
    } catch (FileNotFoundException fe) {
        fail(fe.getMessage());
    }
}
```

Don’t forget the import statements for BufferedOutputStream, FileOutputStream, and FileNotFoundException.

Note the line that reads

```java
Thread.currentThread().setContextClassLoader(getClass().getClassLoader());
```

Threads are a mechanism by which a program can be doing several things at once, perhaps sharing a processor, or perhaps using separate processors. We will explore threads in a later
chapter. For now, we simply use that statement whenever we wish to use an XMLEncoder or an XMLDecoder.

Without that line, the XMLEncoder is unaware of the class Address. Your method will generate a ClassNotFoundException error. To avoid that error, include the statement in any method which creates either an XMLEncoder or XMLDecoder object. This hint was provided through an answer to a question at http://dev.eclipse.org/mhonarc/lists/gef-dev/msg00469.html.

Once the XMLEncoder is aware of the class it is processing, it can prepare output for all instance variables which have setter methods.

Note that when our objects contain collections, there are no setters for the collections, so the collections are not output. We need to do that ourselves and we will see how in a few moments.

Suppose we create an address object with the statement

```java
Address a1 = new Address("1234", "X", "Fifth", "Avenue", "SE", "Calgary", "AB", "Canada", "T0L0M0");
```

This is not a real address; it’s just values I made up.

But the file created when I pass a1 to the XMLEncoder does not contain any data. The XMLEncoder requires us to have a no-parameter (nullary) constructor. We don’t have a no-parameter constructor yet, so we need to create one.

```java
/**
 * no-parameter constructor for an Address, used with XMLEncoder and XMLDecoder.
 */
public Address() {
}
```

This nullary constructor creates the object, but none of the instance variables have values. It might be better if we at least gave the instance variables the value of blank, or zero, or null, depending on their datatype.

Note that when you are saving an object to an XML file, there MUST be a nullary constructor for the object.

If the class contains any collections (Address does not), your nullary constructor must create them (as empty collections) too, as we will see in a few moments.

Check that you have setters for all your instance variables.

Check also that the name of the setter is the word “set” followed by the name of the instance variable. If you need to rename any of your setters, change (and rerun, just to be sure) the appropriate unit test as well.
Now the file created when we pass a1 to the XMLEncoder is shown below. (Double-click it and the file opens in Internet Explorer.) The heading lines may change as different versions of XML and Java appear. This file was created using an older version of Java but the file structure remains unchanged.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.5.0_01" class="java.beans.XMLDecoder">
<object class="Address">
    <void property="city">
        <string>Calgary</string>
    </void>
    <void property="country">
        <string>Canada</string>
    </void>
    <void property="direction">
        <string>SE</string>
    </void>
    <void property="name">
        <string>Fifth</string>
    </void>
    <void property="number">
        <string>1234</string>
    </void>
    <void property="postalCode">
        <string>T0L0M0</string>
    </void>
    <void property="province">
        <string>AB</string>
    </void>
    <void property="suffix">
        <string>X</string>
    </void>
    <void property="type">
        <string>Avenue</string>
    </void>
</object>
</java>
```

You can see how the file encodes the class name and instance variables. The <object> tag indicates the class of the object. The property tags indicate, in alphabetical order, the instance variables and <string> indicates the datatype of the instance variables. Other datatypes will appear when necessary. Each tag has its corresponding closing tag, beginning with <./.

You can display an XML file in a text editor (like Notepad), in a browser (like Internet Explorer or Mozilla Firefox), or in an XML editor (like XMLSpy) or, as noted earlier, you can write a program in some other language to read and display the contents of the file.
Using an XMLDecoder – no collections

Once we have created a file containing XML statements, we can create a method to retrieve information from it.

Here is a method which retrieves a single Address object from the file address.xml, assuming the nullary constructor and all setters exist, named correctly.

```java
/**
 * restore an Address from XML.
 * @param d the XMLDecoder to use
 */
public void readObject(XMLDecoder d){
    try {
        Address a = (Address) d.readObject();
        // a has been created as a temporary variable.
        // now copy its contents to the variable we really want.
        this.number = new String(a.number);
        this.suffix = new String(a.suffix);
        this.name = new String(a.name);
        this.type = new String(a.type);
        this.direction = new String(a.direction);
        this.city = new String(a.city);
        this.province = new String(a.province);
        this.country = new String(a.country);
        this.postalCode = new String(a.postalCode);
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
    return;
}
```

We can test the decoding with simple unit tests. Create two Address objects (a1 and a2) in the setUp method, containing different values. Then write one out (as we did earlier), and retrieve its value into the other.

```java
@Test
public void testXMLDeserialization() {
    try {
        XMLDecoder d = new XMLDecoder(new BufferedInputStream(new FileInputStream("address.xml")));
        a2.readObject(d);
        d.close();
    } catch (FileNotFoundException fe) {
        System.err.println(fe.getMessage());
    }
}
```
assertEquals("addresses match", a1.toString(), a2.toString());
}

Don't forget the import statements.

Does the method work?

/**
 * Possible alternative to restoring an Address from XML.
 */
public void readObject(XMLDecoder d){
    Thread.currentThread().setContextClassLoader(getClass().getClassLoader());
    try {
        Address a = (Address) d.readObject();
        // this part of the method is different!
        this = a;
    }
    catch (Exception e) {
        System.out.println(e.getMessage());
    } // end catch
    return;
} // end readObject

Why or why not?

Using an XMLEncoder – with a collection

Most of our classes include collections. The collections are instance variables which have neither setters nor getters. Instead, they have methods (which I usually, though not always, name remember) for adding elements to the collection.

This lack of getters and setters causes problems with XML serialization, since the XMLEncoder and XMLDecoder use the setters and getters.

However, Java provides an alternative mechanism we can use here, creating an object of the DefaultPersistenceDelegate class. A DefaultPersistenceDelegate determines the statements the XMLEncoder produces for us.

Unknowingly, we have been using a DefaultPersistenceDelegate, a simple one which assumes the class is a JavaBean, when we saved Address objects using XML. It knew what to do with the nullary constructor, the getters, and the setters.

But we must provide an alternative DefaultPersistenceDelegate whenever we are dealing with a class which contains one or more collections. Let’s see how we do that with the Section class. Remember that the Section class contains two collections, one (theStudents) of student identifiers and the other (theMeetings) of Meeting objects.
First, we must make modifications to the Section class. Create a nullary constructor. This constructor creates the empty collections.

```java
public Section() {
    this.theStudents = new HashSet<Student>();
    this.theMeetings = new TreeSet<Meeting>(Meeting.TIME_ORDER);
}
```

Note that you probably have two places in your code where you create these two collections. Good programming practice suggests the code to create these collections should be in only one place. Thus, you should create a private method and call that method whenever necessary.

```java
private void makeCollections() {
    this.theStudents = new HashSet<Student>();
    this.theMeetings = new TreeSet<Meeting>(Meeting.TIME_ORDER);
}
```

The constructor becomes

```java
public Section() {
    makeCollections();
}
```

You can also use `makeCollections` in your other constructor.

Then we extend the DefaultPersistenceDelegate class within the file containing the Section class. That is, Section.java will now contain two classes.

class SectionPersistenceDelegate extends DefaultPersistenceDelegate{
    // create XML statements to define instance
    // variables and their values.
    // each statement includes the method to be called and its arguments, as an array
    protected void initialize(Class type,
            Object oldInstance,
            Object newInstance,
            Encoder out) {
        // single-valued instance variables
        Section s = (Section) oldInstance;
        out.writeStatement(new Statement(oldInstance, "setIdentifier",
                new Object[]{s.getIdentifier()}));

        out.writeStatement(new Statement(oldInstance, "setDepartmentAbbreviation",
                new Object[]{s.getDepartmentAbbreviation()}));

        out.writeStatement(new Statement(oldInstance, "setCourseNumber",
                new Object[]{s.getCourseNumber()}));
    }
}
out.writeStatement(new Statement(oldInstance, "setSectionNumber", new Object[]{s.getSectionNumber()}));

out.writeStatement(new Statement(oldInstance, "setStartYear", new Object[]{s.getStartYear()}));

out.writeStatement(new Statement(oldInstance, "setStartMonth", new Object[]{s.getStartMonth()}));

out.writeStatement(new Statement(oldInstance, "setEndYear", new Object[]{s.getEndYear()}));

out.writeStatement(new Statement(oldInstance, "setEndMonth", new Object[]{s.getEndMonth()}));

// collections

Iterator<Student> itS = s.studentIterator();
while (itS.hasNext()) {
    out.writeStatement(new Statement(oldInstance, "remember", new Object[]{itS.next()}));
}

Iterator<Meeting> itM = s.meetingIterator();
while (itM.hasNext()) {
    out.writeStatement(new Statement(oldInstance, "remember", new Object[]{itM.next()}));
}

Recall that the theStudents and theMeetings collections both have private visibility. To access their elements, we need two methods in the Section class to provide iterators over the collections, studentIterator() and meetingIterator().

public Iterator<Student> studentIterator() {
    return theStudents.iterator();
}

public Iterator<Meeting> meetingIterator() {
    return theMeetings.iterator();
}
As noted in an earlier chapter, it would be better style if we created iterators allowing us to access individual elements of the collections.

Make it so.

Finally we create a DefaultPersistenceDelegate object and attach it to the XMLEncoder we are using.

```java
/**
 * save a Section as XML.
 * @param e The XMLEncoder which will create the file
 * @return nothing
 */
public void writeObject(XMLEncoder e){
    // need a DefaultPersistenceDelegate because of students and meetings
    e.setPersistenceDelegate(Section.class, new SectionPersistenceDelegate());
    e.writeObject(this);
} // end writeObject
```

That’s it.

Except of course there are some import statements we need.

```java
import java.beans.DefaultPersistenceDelegate;
import java.beans.Statement;
import java.beans.Encoder;
```

Yes, there is a unit test which defines the file to which the XMLEncoder is connected.

```java
@Test
public void testXMLSerialization() {
    Thread.currentThread().setContextClassLoader(getClass().getClassLoader());
    try {
        XMLEncoder e = new XMLEncoder(new BufferedOutputStream(new FileOutputStream("section.xml")));
        // save the object
        s.writeObject(e);
        // clean up
        e.flush();
        e.close();
    } catch (FileNotFoundException fe) {
        fail(e.getMessage());
    }
}
```

Don't forget the import statements in the test class.

Now you are able to write a Section object to an XML file, sometimes called an *archive*. 
You may see messages in the Terminal windows like the following one.

java.lang.Exception: Encoder: discarding statement Section.setIdentifier(Integer);
Continuing ...

You have a problem. The archive has not been created because the method (here setIdentifier with an Integer parameter) you have directed the PersistenceDelegate to use does not exist. Why does it not exist? Did you not create it? Did you misspell the name of the method? Did you indicate the wrong type of parameter?

If you wish to see the XML file produced by the writeObject method, simply navigate to the folder or directory in which it was placed (the default is the one in which your project is located) and use any of the programs noted earlier.

The file we have just created contains many lines, but is not that complicated. Here is sample output, describing a section containing four students, meeting twice a week.

<?xml version="1.0" encoding="UTF-8" ?>
 <java version="1.5.0_01" class="java.beans.XMLDecoder">
   <object class="Section">
     <int>123</int>
   </void>
   <void property="identifier">
     <int>1</int>
   </void>
   <void property="departmentAbbreviation">
     <string>COSC</string>
   </void>
   <void property="courseNumber">
     <int>111</int>
   </void>
   <void property="sectionNumber">
     <string>001</string>
   </void>
   <void property="startYear">
     <int>2005</int>
   </void>
   <void property="startMonth">
     <int>8</int>
   </void>
   <void property="endYear">
     <int>2005</int>
   </void>
   <void property="endMonth">
     <int>11</int>
   </void>
   <void method="addStudent">
     <string>3</string>
   </void>
   <void method="addStudent">
     <string>2</string>
   </void>
</java>
<object class=" Meeting">
  <void method="addMeeting">
    <object class="Meeting">
      <void property="dayOfWeek">
        <int>1</int>
      </void>
      <void>
        <object class="Meeting">
          <void property="endTime">
            <int>1200</int>
          </void>
          <void>
            <object class="Meeting">
              <void property="dayOfWeek">
                <int>3</int>
              </void>
              <void>
                <object class="Meeting">
                  <void property="endTime">
                    <int>1250</int>
                  </void>
                  <void>
                    <object class="Meeting">
                      <void property="roomNumber">
                        <string>L322</string>
                      </void>
                      <void>
                        <object class="Meeting">
                          <void property="startTime">
                            <int>1100</int>
                          </void>
                          <void>
                        </object>
                      </void>
                    </object>
                  </void>
                </object>
              </void>
            </object>
          </void>
        </object>
      </void>
    </object>
  </void>
</object>

You can simplify the display somewhat (when you are using Internet Explorer) by clicking the minus sign beside each line that begins <object class =

This gives the following file.

<?xml version="1.0" encoding="UTF-8" ?>
- <java version="1.5.0_01" class="java.beans.XMLEncoder">
  + <object class="Section">
The plus and minus signs are from the editor. They tell us that the file contains an object of type *Section* and nothing else, unlike when we were using object serialization. To see more details about the instance variables of an object, click the plus sign. To suppress the details, click the minus sign which replaces the plus sign.

**Using an XMLDecoder – with a collection**

The archive contains all the instructions necessary to reconstitute a *Section* object. The following method does the reconstituting.

```java
/**
 * restore a Section from XML.
 * @param d The XMLDecoder being used
 */
public void readObject(XMLDecoder d){

    Section s = (Section) d.readObject();

    this.identifier = s.identifier;
    this.subjectAbbreviation = new String(s.subjectAbbreviation);
    this.courseNumber = s.courseNumber;
    this.sectionNumber = new String(s.sectionNumber);
    this.startYear = s.startYear;
    this.startMonth = s.startMonth;
    this.endYear = s.endYear;
    this.endMonth = s.endMonth;

    Iterator<Student> itS = s.studentIterator();
    while (itS.hasNext()) {
        this.theStudents.add(itS.next().clone());
    }

    Iterator<Meeting> itM = s.meetingIterator();
    while (itM.hasNext()) {
        this.theMeetings.add(itM.next().clone());
    }
}
```

The statement

```java
Section s = (Section) d.readObject();
```

implements the instructions we placed in the archive to create a temporary object of the appropriate type and we copy that temporary object into our object.

While copying elements from the temporary copy into our object, we ensure the object contains its own elements, not just references to the elements of the temporary copy. Thus we need to use new `String` for Strings, and the clone method for the `Meeting` objects. If you have not implemented the clone method for `Meeting`, now would be a good time to do so.
The other instance variables are numbers, primitive datatypes. We simply use an assignment statement for them. That creates a copy.

The previous code shows how to deal with collections which are lists or sets. What about maps, if we had any?

One possible solution is to save each key-value pair, and then reconstruct the map. The Map.Entry interface provides an easy way to retrieve the key-value pairs.

**Programming Style**

I have adopted the practice of having readObject (having an XMLDecoder object as a parameter) and writeObject (having an XLEncoder object as a parameter) methods in all my classes. The corresponding unit tests are called testXMLSerialization and testXMLDeserialization.

Similarly, if I had covered the previous chapter, I would have readObject (having an ObjectInputStream object as a parameter) and writeObject (having an ObjectOutputStream object as a parameter) methods in all my classes. The corresponding unit tests are called testObjectSerialization and testObjectDeserialization.

As these methods have different signatures, Java does not get confused. I find it useful to use similar names for similar methods, my own personal use of overloading.

**Using XML – summary**

This is only a brief introduction to using XML as a tool to provide persistence for objects. There is much more to learn than what is presented here.

Online sources you may wish to explore include the following.
- XML.org, found at [http://www.xml.org/](http://www.xml.org/)
- An O’Reilly (a publisher of well-received computer science books) website on XML, [http://www.xml.com/](http://www.xml.com/)
- A collection of XML resources and tutorials, [http://www.w3schools.com/xml/default](http://www.w3schools.com/xml/default)

Besides providing persistence, we actually have the ability, through the use of a technology called XSLT, to present our saved information in many different ways. XSL is a stylesheet language for XML. XSLT is XSL Transformations, a language for translating one XML document into another.

This is definitely a topic better left to another course. At Okanagan College, that would be a third-year (junior) or fourth-year (senior) course.
Chapter 15 – Persistence, part 3

Learning objectives

By the end of the chapter you will be able to:

- Implement persistence using traditional file techniques
- Implement persistence using CSV files

Persistence via traditional files

There is yet another option to providing persistence, by writing data into a series of traditional files. A file is just a sequence of bytes, living on an external storage device. To process a file, you simply read the bytes, breaking them into fields as appropriate. But breaking them into fields is where the complications lie.

BlueJ comes with several example programs. We will use one of those examples to see how to process files.

Input

Look in the `<bluej-home>/examples/file-reader` folder within your BlueJ installation. You will find FileReader.java, shown below.

```java
import java.io.*;
import java.net.URL;

/**
 * This is a little demo showing how to read text files. It will find files
 * that are situated anywhere in the classpath.
 *
 * Currently, two demo methods are available. Both simply print a text file to the
 * terminal. One returns exceptions in case of a problem, the other prints out
 * error messages.
 *
 * @author Michael Kölling
 * @version 1.0 (19. Feb 2002)
 */
public class FileReader {
```
/**
 * Create a file reader
 */
public FileReader()
{
    // nothing to do...
}

/**
 * Show the contents of the file 'fileName' on standard out (the text terminal).
 *
 * @param fileName The name of the file to show
 * @throws IOException if the file could not be opened
 */
public void showFile(final String fileName)
throws IOException
{
    InputStream fstream = openFile(fileName);

    // wrap the stream into an InputStreamReader, so that we read characters
    // rather than bytes (important for non-ascii characters); then wrap it into
    // a BufferedReader, so that we can read lines, rather than single characters
    BufferedReader in = new BufferedReader(new InputStreamReader(fstream));

    // okay, we're ready to go...
    System.out.println("File: " + fileName);
    String line = in.readLine();
    while(line != null) {
        System.out.println(line);
        line = in.readLine();
    }
    System.out.println("<end of file>");
}

/**
 * Same as 'showfile', but don't throw exceptions. If an error occurs,
 * write an error message to the terminal.
 *
 * @param fileName The name of the file to show
 */
public void checkedShowFile(final String fileName)
{
    try {
        showFile(fileName);
    }
    catch (IOException exc) {
        System.out.println("There was a problem showing this file.");
        System.out.println("The error encountered is:");
        System.out.println(exc);
    }
}
/**
 * Open a text file and return a stream to read from that file.
 * The file can reside anywhere in the classpath.
 *
 * @param fileName  The name of the file to open
 * @return An open stream to read from the file
 * @throws IOException if the file could not be opened
 */
public InputStream openFile(final String fileName)
    throws IOException
{
    if(fileName == null)
        throw new IOException("Cannot open file - filename was null.");
    URL url = getClass().getClassLoader().getResource(fileName);
    if(url == null)
        throw new IOException("File not found: " + fileName);
    return url.openStream();
}

This class begins with the notation that it is used for reading text files. A text file is one which is organized into lines of data, with each terminated by an end-of-line character. Knowing that character exists allows us to read the text file one line at a time. There are other types of files; they are read in different ways, and will not be considered here.

To see the class in use, open the FileReader project, compile the class, and right-click the class.

Execute the constructor, placing an object on the workbench.

Right-click the object and select the showFile method. It will request the name of a file. There are several files immediately available in the current folder – FileReader.java, README.TXT, test.txt, and package.bluej. Read one of them and see how the output appears in the terminal window.

What happens if you use the showFile method and specify a file which does not exist? It throws an exception, as the documentation describes.

Try opening some files using the checkedShowFile method. This method uses try and catch blocks. Notice how checkedShowFile invokes showFile, but does so in a try block.

So how does showFile work?

    InputStream fstream = openFile(fileName);

This line establishes a connection between the file on a disk and a stream, a sequence of characters.

    // wrap the stream into an InputStreamReader, so that we read characters
// rather than bytes (important for non-ascii characters); then wrap it into
// a BufferedReader, so that we can read lines, rather than single characters
BufferedReader in = new BufferedReader(new InputStreamReader(fstream));

This line (only one of the four is executable) converts the unstructured stream of characters into a collection of lines, which we will read one at a time.

    // okay, we're ready to go...
    System.out.println("File: " + fileName);
    String line = in.readLine();

These lines display the name of the file in the Terminal window and read the first line from the file. If the file is empty, that first line will be null. Testing for a null line is the way we stop processing the file as we see in the loop below.

        while(line != null) {
            System.out.println(line);
            line = in.readLine();
        }

As long as there is a line which has been read, the loop displays it and reads another line. This is exactly the same logic we used with iterators over collections.

Once all the lines of the file have been processed, line becomes null, the loop terminates, and the method displays a message to let you know the listing is complete.

        System.out.println("<end of file>");

It is always a good idea to clearly mark the end of a listing so you know the listing is complete.

In the public relations world, press releases often end with a line saying -30-.

This method uses many classes and methods. From whence come InputStream, InputStreamReader, BufferedReader, and the readLine method? The class file begins with two import statements.

    import java.io.*;
    import java.net.URL;

The first is the one that provides access, via a wildcard, to the three classes mentioned. The readLine method is a method within which of them?

Since the statement in which readLine appears is

        line = in.readLine();

the readLine method must be a method within the class BufferedReader, since in is a BufferedReader. Check that is so.
How does `checkedShowFile` work? As noted earlier, it simply encloses a call to `showFile` in a try block, which throws (and catches) an exception if there is a problem opening the file.

But how is the file opened? How do you establish a connection between a stream, which a Java program can process, and a file on disk? `FileReader` contains a method, `openFile`, which `showFile` uses to establish the connection.

```java
/**
 * Open a text file and return a stream to read from that file.
 * The file can reside anywhere in the classpath.
 *
 * @param fileName  The name of the file to open
 * @return An open stream to read from the file
 * @throws IOException if the file could not be opened
 */
public InputStream openFile(String fileName) throws IOException {
    if(fileName == null)
        throw new IOException("Cannot open file - filename was null.");
    URL url = getClass().getClassLoader().getResource(fileName);
    if(url == null)
        throw new IOException("File not found: " + fileName);
    return url.openStream();
}
```

This method uses the URL class. The Java documentation says that “Class URL represents a Uniform Resource Locator, a pointer to a "resource" on the World Wide Web. A resource can be something as simple as a file or a directory, or it can be a reference to a more complicated object, such as a query to a database or to a search engine.”

`openFile` checks that the file name has been provided, showing an exception if it does not. If the file name has been provided, `openFile` tries to open the file, throwing an exception if it cannot.

The documentation for the class refers to classpath. The classpath is an environment variable which is set to indicate the folders in which Java should look for system libraries. It generally refers to a collection of folders, not just the project folder. To see its value (in Windows), open a command prompt and issue the command `set | find "CLASSPATH"

If nothing appears, do not be upset. Classpath is not always used.

A single dot in the classpath refers to the current folder.

**Output**

But how do we write to a text file?
Here is a very simple example.

```java
import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.IOException;

public class WriteFileExample {
    public WriteFileExample() {
    }

    public void writeFile () {
        // some data to write to the file
        double hours = 12.5;
        double rate = 24.87;
        double grossPay = hours * rate;

        // The file on the disk is named write.txt. The connection between that file and the
        // stream in the method is through a FileWriter object
        try {
            // establish the connection
            FileWriter fstream = new FileWriter("write.txt");
            BufferedWriter out = new BufferedWriter(fstream);

            // write some text and then a line separator
            out.write("Salary calculations");
            out.newLine();
            out.write("Hours = " + hours);
            out.newLine();
            out.write("Rate = " + rate);
            out.newLine();
            out.write("Gross pay = " + grossPay);

            // everything has been written so close the file
            out.close();
        }
        // if anything goes wrong, display an error message in the terminal
        // window
        catch (IOException ioe) {
            System.out.println(ioe.getMessage());
        }
    }
}
```

Note the use of the `newLine` method. As its documentation says, “Writes a line separator. The line separator string is defined by the system property `line.separator`, and is not necessarily a single newline (`\n`) character.”

But how do we use this information in our project?
A digression - BlueJ definition files

We have already seen files which contain keywords at the beginning of each line in chapter 3, where we looked at bluej.defs, which contains the following lines.

```
# Syntax colour definitions
# =========================

# Key to values
# -----------

# comment Single line comments (/!) and standard multi-line comments (/** */)
# javadoc Multi-line javadoc comments (/** */)
# keyword1 Standard Java keywords (e.g. abstract, final, do, if, else, new, catch etc.)
# keyword2 Class creation keywords (package, import, class, interface, extends, implements)
# keyword3 Remaining Java keywords (this, null, super, true, false)
# primitive Java primitives (int, float, double, char)
# string String literals (anything in "quotes")
# label Labels for loops or in switch/case statements
# invalid Unclosed string literals or other detected errors
# other Anything else
# background Editor background colour

# Any of the values above that are not defined are given the BlueJ default colours.

# Key to colours
# ------------

# Each colour should be given a six digit hexadecimal value of the from rrggb where
# the pairs of digits refer to the red, green and blue values respectively.

comment = 999999
javadoc = 000099
stand-out = ee00bb
keyword1 = 660033
keyword2 = cc0000
keyword3 = 006699
primitive = cc0000
string = 006600
label = 999999

The first few lines are comments. There are two types of comments: those which have an # as the first character of the line and those which are an empty line.

But look at the last few lines. They each begin with a keyword followed by a space, an equals sign, and another space. When BlueJ reads this file, it identifies the first few characters of each line and performs appropriate actions.

End of digression.
Saving a MyDate object using keywords

We will use a similar technique to save a object. Let’s do a simple object, a MyDate object.

Recall that a MyDate object contains instance variables named year, month, and day. In English, the obvious keywords are year, month, and day. Even if English is not your native language those may still be the obvious keywords. Thus the lines in the file may look like this.

```java
month = 5
day = 10
year = 2015
```

if we adopt BlueJ’s practice of surrounding the equals sign with spaces or they may look like this.

```java
month=5
day=10
year=2015
```

Note that the order does not matter since the code that extracts the data from the file will look at the keywords and decide which instance variable is involved.

A method to create a MyDate object in this format follows.

```java
/**
 * Convert a date to a String suitable for placement
 * in a text file using keywords to identify each instance
 * variable.
 * @return a String in the correct format
 */
public String toStringKeyword() {
    String result = ""
    result += "year=" + year + "\n";
    result += "month=" + month + "\n";
    result += "day=" + day;

    return result;
}
```

Yes, this method produces a different order, the international standard order. Some other method will write this String to a file. In my case, that will be a unit test.

```java
@Test
public void testWriteStringKeyword() {
    // write the file
    try {
        String fileName = "MyDateWithKeywords.txt";
        FileWriter fstream = new FileWriter(fileName);
        BufferedWriter out = new BufferedWriter(fstream);
```
out.write(d.toStringKeyword());
out.close();
}
catch (IOException ioe) {
    System.err.println(ioe.getMessage());
}
catch (Exception e) {
    System.err.println("Unexpected error " + e.toString());
}
}

Once you run that test, you will have the object in a file. To restore the MyDate object from a file containing this data, consider this method.

/**
 * To retrieve a MyDate object from a file containing
 * the keyword structure
 * @param fileName the file containing the object
 * @throws IOException if there is a problem reading
 * the file
 */
public void restoreViaKeyword(final String fileName) throws IOException {
    FileReader fstream = new FileReader(fileName);
    BufferedReader in = new BufferedReader(fstream);
    String data = in.readLine();
    while (data != null) {
        String ucData = data.toUpperCase();
        if (ucData.startsWith("YEAR="))
            this.year = (new Long(data.substring(5))).longValue();
        if (ucData.startsWith("MONTH="))
            this.month = (new Long(data.substring(6))).longValue();
        if (ucData.startsWith("DAY="))
            this.day = (new Long(data.substring(4))).longValue();
        data = in.readLine();
    }
}

Here is the unit test which shows that the restoration works.

@Test
public void testRestoreViaKeyword() {
    try {
        String fileName = "MyDateWithKeywords.txt";
        FileReader fstream = new FileReader(fileName);
        BufferedReader in = new BufferedReader(fstream);
        MyDate temp = new MyDate();
        temp.restoreViaKeyword(fileName);
        in.close();
    }
}
assertEquals(d, temp);
}  
catch (IOException e) {
    fail(e.getMessage());
}  
catch (Exception e) {
    fail("Unexpected error " + e.toString());
}
}

CSV

A traditional file with a special format is a CSV, comma-separated variables, file. In such a file, as you might expect, each value is separated from the next by a comma. A missing field is indicated by the presence of two consecutive commas.

Why would such a file be used?

It is the traditional way of importing data into (or exporting data out of) a spreadsheet.

Convert a Student object to CSV format

Let's examine how to write a Student object to a CSV file.

The Student object contains an identifier, a Name object, and an Address object (which might be null), and a MyDate object (which might be null), all inherited from Person. It also contains a second Address object.

Thus we need a method to convert a Name object to CSV format, a method to convert an Address object to CSV format, and a method to convert a MyDate object to CSV format.

The first is here.

/**
 * Convert a Name object to CSV format.
 * @return Name object in CSV format
 */
public String toStringCSV() {
    String result = name1 + "," + name2 + "," + name3 + ",";
    result += nickname + "," + preferredName + "," + fullName;
    return result;
}

The first attempt at converting an Address object to CSV format is here.

/**
**Convert an Address object to CSV format.**

* @return an Address object in CSV format

```java
public String toStringCSV() {
    String result = number + "," + suffix + "," + name + ",";
    result += type + "," + direction + "," + city + ",";
    result += province + "," + country + "," + postalCode;
    return result;
}
```

The first attempt to convert a MyDate object to CSV format is here.

```java
/**
 * Convert a MyDate object to CSV format.
 * @return a MyDate object in CSV format
 */
public String toStringCSV() {
    return year + "," + month + "," + day;
}
```

Test them! Don't just take my word for it that they work.

With these methods created and tested, we can think about creating the method to convert a Student to CSV format. Recall that a Student object is a Person object. Hence, it makes more sense to place a toStringCSV method within Person.

```java
/**
 * Convert a Person object to CSV format.
 * @return Person object in CSV format
 */
public String toStringCSV() {
    String result = identifier + ",";
    result += theName.toStringCSV() + ",";
    result += homeAddress.toStringCSV() + ",";
    result += birthDate.toStringCSV();
    return result;
}
```

Then we can write the method to convert a Student object to CSV format.

```java
/**
 * Convert a Student object to CSV format.
 * @return Student object in CSV format
 */
public String toStringCSV() {
    String result = super.toStringCSV() + ",";
    result += localAddress.toStringCSV();
    return result;
}
```
Note that the method needs to insert commas after the calls to its parent's toStringCSV method. Why?

When I tested these toStringCSV methods, the test failed with a NullPointerException. I had tried to use a Student object with no local address.

Modify the toStringCSV methods in both Person and Student.

```java
/**
 * Convert a Person object to CSV format.
 * @return the line
 */
public String toStringCSV() {
    String result = identifier + "",";
    result += theName.toStringCSV() + "",";
    if (homeAddress != null)
        result += homeAddress.toStringCSV() + "",";
    else
        result += ",,,,,,,," + ",";
    if (birthDate != null)
        result += birthDate.toStringCSV();
    else
        result += ",";
    return result;
}
```

For a truly advanced topic, see http://download.oracle.com/javase/tutorial/reflect/index for a description of how you can use reflection to determine the number of instance variables in a class.

Notice that a Person object has two instance variables either of which may be null. A Student object has one.

```java
/**
 * Convert a Student object to CSV format.
 * @return the line
 */
public String toStringCSV() {
    String result = super.toStringCSV() + "",";
    if (localAddress != null)
        result += localAddress.toStringCSV();
    else
        result += ",,,,,,,,";
    return result;
}
```

This created the correct result, but it has introduced a style problem.

In a later chapter we will talk about coupling. Coupling is the term used to describe how much information one object needs to know about another. Too much (or tight) coupling is a bad idea.
Here, a Student object needs to know there are nine instance variables in an Address object and a Person object needs to know there are three instance variables in a MyDate object.

If we avoid reflection, the solution is to create another method in Address.

```java
/**
 * Create the correct CSV format for a null address
 * @return a null address in CSV format
 */
public static String toStringCSVNull() {
    return ",,,,,,,,,",
}
```

The method is static; there is only one such method within the Address class and it is not attached to an object; it is attached to the class itself. There are nine instance variables; hence there must be eight commas.

Create a similar method within MyDate.

```java
/**
 * Convert a null MyDate object to CSV format
 * @return a null MyDate object in CSV
 */
public static String toStringCSVNull() {
    return ",,",
}
```

And then modify the toStringCSV methods in Person and Student to use these new methods, Person first.

```java
/**
 * Convert a Person object to CSV format.
 * @return the Person object in CSV format
 */
public String toStringCSV() {
    String result = identifier + ",,
    result += theName.toStringCSV() + ",,;
    if (homeAddress != null)
        result += homeAddress.toStringCSV() + ",,;
    else
        result += Address.toStringCSVNull() + ",,;
    if (birthDate != null)
        result += birthDate.toStringCSV();
    else
        result += MyDate.toStringCSVNull();
    return result;
}
```

And then Student.
public String toStringCSV()
{
    String result = super.toStringCSV() + ";
    if (localAddress != null)
        result += localAddress.toStringCSV();
    else
        result += Address.toStringCSVNull();
    return result;
}

When you examine the result, you will see that the data is there, but there is no indication of what the data is. Remember that the main purpose of CSV files is to facilitate the importing of data into a spreadsheet, and spreadsheet columns should have labels.

We'll add the labels as we write the file.

Writing a CSV file

It makes no sense to import only one row into a spreadsheet, so we should examine what happens when we write a collection of Student objects to a file, a task which should take place within College.

/public
* Create a CSV file containing all students registered at the college.
* The file is saved as students.csv
*/
public void studentsCSV(final String fileName) {
    // write the file
    try {
        FileWriter fstream = new FileWriter(fileName);
        BufferedWriter out = new BufferedWriter(fstream);

        // first the headings
        String result = "Identifier,Name1,Name2,Name3,Preferred Name,"
            + "Nickname,Full (Legal) Name,"
            + "Home Address - Number,Suffix,Name,Type,Direction,"
            + "City, Province,Country,Postal Code,"
            + "Birthdate – Year, Month, Day,"
            + "Local Address - Number,Suffix,Name,Type,Direction,"
            + "City, Province,Country,Postal Code";

        // headings form one line, each student forms another
        for (Student s:theStudents) {
            result += "\n" + s.toStringCSV();
        }

        out.write(result);
    }
}
out.close();

}  
catch (IOException ioe) {  
    System.err.println(ioe.getMessage());
}  
catch (Exception e) {  
    System.err.println("Unexpected error " + e.toString());
}  
}

**Reading a CSV file into a spreadsheet**

Examine the file in a text editor.

Load it into a spreadsheet. Notice that you may need to alter the column widths in the spreadsheet to have the data (particularly the identifier) displayed properly.

**Reading a CSV file into a Java program**

When we read a CSV file back into a Java program, we will be using it to create the collection it contains. Thus we need to ignore the first line in the file, containing the title, but we do need to know the fields in subsequent lines and how they are used.

Consider the following method.

```java
/**  
 * Process a CSV file containing all students registered at the college.  
 */  
public String restoreStudentsFromCSV(final String fileName) {
    String data = "";
    String identifier;
    Name theName = null;
    Address homeAddress = null;
    MyDate birthDate = null;
    Address localAddress = null;
    Student s = null;
    // read the file
    try {
        FileReader fstream = new FileReader(fileName);
        BufferedReader in = new BufferedReader(fstream);
        // skip the headings
        data = in.readLine();
        // read the next line of data
        data = in.readLine();
        while (data != null) {
            // but how do you break the line into its portions?
            Scanner sc = new Scanner(data);
            sc.useDelimiter(",");
```
// get the identifier
identifier = sc.next();
// get the parts of the name
String n1 = sc.next();
String n2 = sc.next();
String n3 = sc.next();
String n4 = sc.next();
String n5 = sc.next();
String n6 = sc.next();
try {
    theName = new Name(n1, n2, n3, n4, n6, n5);
} catch (NameException ne) {
    System.out.println("Name error " + ne.getMessage());
}
// get the parts of the home address
String a1 = sc.next();
String a2 = sc.next();
String a3 = sc.next();
String a4 = sc.next();
String a5 = sc.next();
String a6 = sc.next();
String a7 = sc.next();
String a8 = sc.next();
String a9 = sc.next();
homeAddress = new Address(a1, a2, a3, a4, a5, a6, a7, a8, a9);

// get the parts of the birthdate
int d1 = 0;
int d2 = 0;
int d3 = 0;
try {
    d1 = Integer.valueOf(sc.next()).intValue();
} catch (NumberFormatException nfe) {
    d1 = 0;
} try {
    d1 = Integer.valueOf(sc.next()).intValue();
} catch (NumberFormatException nfe) {
    d1 = 0;
} try {
    d2 = Integer.valueOf(sc.next()).intValue();
} catch (NumberFormatException nfe) {
    d2 = 0;
} try {
    d3 = Integer.valueOf(sc.next()).intValue();
} catch (NumberFormatException nfe) {
d3 = 0;
}

try {
    if ((d1 != 0) && (d2 != 0) && (d3 != 0))
        birthDate = new MyDate(d1, d2, d3);
} catch (MyDateException mde) {
    birthDate = null;
}

// get the parts of the local address
a1 = sc.next();
a2 = sc.next();
a3 = sc.next();
a4 = sc.next();
a5 = sc.next();
a6 = sc.next();
a7 = sc.next();
a8 = sc.next();
a9 = sc.next();
localAddress = new Address(a1, a2, a3, a4, a5, a6, a7, a8, a9);

try {
    s = new Student(identifier, theName, homeAddress, localAddress, birthDate);
} catch (IdentifierException ie) {
    System.out.println("identifier error " + ie.getMessage());
} catch (NameException ne){
    System.out.println("Name error " + ne.getMessage());
};

// get the next line from the CSV file. if there is no next line, data becomes null
data = in.readLine();
in.close();

} catch (IOException ioe) {
    System.err.println(ioe.getMessage());
} catch (Exception e) {
    System.err.println(data + "Unexpected error " + e.toString());
}

You will notice that the method is tightly-coupled to both the Address and MyDate classes. However, the method can be rewritten to remove much of this coupling. Before we see how to do that, I draw your attention to the use of the Scanner class.
**Scanner**

There are several ways to break a string into its constituent parts. The Scanner class is the one I have chosen to use here.

For other techniques, look at the Pattern and StringTokenizer classes.

The documentation for Scanner describes it as “A simple text scanner which can parse primitive types and strings using regular expressions.

“A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various next methods.”

Regular expressions are the subject of entire books. We have seen them before. A regular expression is a way of describing a pattern. The pattern describes the end marker which separates parts of the string. In our case, the pattern will be a comma.

A token is a portion of a string. It may be a character or a series of characters, depending on the separator you specify. The default is whitespace, essentially a synonym for spaces and punctuation.

Thus, we can use a Scanner to divide a string into portions. We need to identify the string.

Scanner sc = new Scanner(data);

We need to identify the separator.

sc.useDelimiter(",");

We need to use methods to extract the “next” token from the string. There are many such methods, depending on the type of data contained in the token. We are usually extracting strings, so we can use the next method. For example, when we are extracting the parts of the name from the CSV file, we use

String n1 = sc.next();
String n2 = sc.next();

The complication arises when we are extracting numbers from the CSV file. If the birth date is missing, the CSV file contains two commas, with nothing between. See the MyDate.toStringCSVNull method above.

If we read spaces or nulls using the nextInt method, we would throw an exception. Instead, we use next to read a String and try to convert it to an Integer from which we extract the underlying int.

try {
    d1 = Integer.valueOf(sc.next()).intValue();
The Scanner class may be useful when reading from a keyboard when we are not within the BlueJ environment, a situation you may find in a later lab assignment. How to read from the keyboard is shown in the Scanner documentation.

**Decoupling**

As written, the method needs to know the structure of the CSV file. It needs to know how many fields are required to construct a Name object, for example. But consider this method, from within the Name class.

```java
/**
 * Rebuild a name from CSV data.
 * @param sc The Scanner from which to extract the parts of the name
 * @return the rebuilt name
 */
public static Name rebuildFromCSV(Scanner sc) {
    Name result = null;
    String n1 = sc.next();
    String n2 = sc.next();
    String n3 = sc.next();
    String n4 = sc.next();
    String n5 = sc.next();
    String n6 = sc.next();
    try {
        result = new Name(n1, n2, n3, n4, n6, n5);
    }
    catch (NameException ne) {
        // cannot happen
    }
    return result;
}
```

We see that the method uses the Scanner object already created and retrieves from it enough information to rebuild a Name object. Thus the information about the instance variables remains hidden within the Name class.

In a similar manner we can create a method to rebuild a MyDate object and a method to rebuild an Address object. These two methods will keep secret the number of instance variables in the object.

Once these methods are created, we can modify restoreStudentsFromCSV so that the coupling is much reduced.

```java
/**
 * Process a CSV file containing all students registered at the college.
 */
```
public void restoreStudentsFromCSV(final String fileName) {
    String data = "";
    String identifier;
    Name theName = null;
    Address homeAddress = null;
    MyDate birthDate = null;
    Address localAddress = null;
    Student s = null;
    // read the file
    try {
        FileReader fstream = new FileReader(fileName);
        BufferedReader in = new BufferedReader(fstream);
        // skip the headings
        data = in.readLine();
        // read the next line of data
        data = in.readLine();
        while (data != null) {
            // use a Scanner to break the CSV data into its portions
            Scanner sc = new Scanner(data);
            sc.useDelimiter(",");

            // get the identifier
            identifier = sc.next();

            // rebuild the name
            theName = Name.rebuildFromCSV(sc);

            // rebuild the home address
            homeAddress = Address.rebuildFromCSV(sc);

            // rebuild the birth date
            birthDate = MyDate.rebuildFromCSV(sc);

            // rebuild the local address
            localAddress = Address.rebuildFromCSV(sc);

            // create the Student object
            try {
                s = new Student(identifier, theName, homeAddress, localAddress, birthDate);
            } catch (IdentifierException ie) {
                System.out.println("identifier error " + ie.getMessage());
            } catch (NameException ne) {
                System.out.println("Name error " + ne.getMessage());
            }

            // get the next line from the CSV file. if there is no next line, data becomes null
            data = in.readLine();
        }
        in.close();


```java
} catch (IOException ioe) {
    System.err.println(ioe.getMessage());
} catch (Exception e) {
    System.err.println(data + "Unexpected error " + e.toString());
}
```

### Manifest files

An example of files organized into lines is a manifest file, a term from the structure of jar files. For specific details, please see [https://docs.oracle.com/javase/tutorial/deployment/jar/manifestindex.html](https://docs.oracle.com/javase/tutorial/deployment/jar/manifestindex.html)

Open a jar file, using a program like 7-Zip or WinZip. Open the file, do not extract files from it!

You will see a jar file contains many files and two folders. The type of each file is given by its extension. MyDate.java contains the source code you wrote; MyDate.class contains the bytecode the compiler produced from MyDate.java.

One folder is named doc. This folder contains all the documentation you have generated about your project.

The other folder is named META-INF. It contains a single file, named MANIFEST.MF. This is a text file, openable with any editor or word processor.

You might think that the manifest file would contain all the files in your project. If it did, it would perhaps need to indicate the type of the file, but that is shown by the extensions. In our example, we will not use that technique.

For our example, consider the problem of saving all the objects we have created into a series of files, each object in a file or all the objects of one class in one file. How do we provide a list of the files we have used?

But first we should consider the contents of the files. Do we wish to have many small files (one object per file) or do we wish to have a few files (with many objects of the same type in each). With many files, the processing may be simpler; with few files, the processing may be more complicated. Let’s see if the data provides any hints.

What collections does the College class contain? Looking at the code (below and in other places) we see collections of Professor, Student, and Course objects.

Consider the Professor and Student classes. We can consider them together as they are connected through their parent class Person. These objects have a unique piece of data, the professor or student number, which could be used as the name of a file. (What would happen if a professor were also a student? I suspect that would cause a problem if we used the number as the name of
the file.) We really have only two choices: simple files with names like Professor1, Professor2, 
* etc. or a large file containing many objects.

For the Course class, we see that each object has a subject abbreviation and a course number. Together these two uniquely identify a course so could be used to form a unique file name. Alternatively we could create unique file names of the form Course1, Course2, * etc. A third possibility is to create one large file that contains all the Course objects.

But a Course object also contains a collection of Section objects. How do we deal with them?

The College documentation is incomplete since it neglects to mention that Section also contains a collection of Student objects; in fact, it should contain a collection of student identifiers. Section contains instance variables (identifier, or subject code, course number, and section number) which could be used to create a unique file name. It seems that a Section object could be stored as one file, with each object being in its own file.

Perhaps that is adequate reason to decide that if one class will be stored with one object per file then objects form all classes will be stored that way.

We can create a manifest file that contains the names of all the files, and there will be many files. As we saw above, each file can be used to restore an object. But first, we must ensure that the College class, the class which will do the backup and restore is correctly built. Consider the code below, in which the backup and restore methods are stubs; that is, they are included but they do nothing.

```java
import java.util.Map;
import java.util.HashMap;
import java.io.*;

/**<br> * @author (Rick)<br> * @version (a version number or a date)<br> * Modified extensively Spring 2015<br> */
public class College {
    // the singleton
    private static College theCollege = null;

    // college-wide constants
    private final int MAX_MEETINGS_PER_WEEK = 5;
    private final int MEETING_START_TIME_LENGTH = 4;
    private final int MEETING_END_TIME_LENGTH = 4;
    private final int ROOM_NUMBER_LENGTH = 4;
    private final int DEPT_ABBREVIATION_LENGTH = 4;

    // the collections the college maintains
    // Professor
    // Student
```
// Course
private static Map<String, Professor> theProfessors;
private static Map<String, Student> theStudents;
private static Map<String, Course> theCourses;

// Course contains a collection of Sections
// Section contains a collection of Meetings

/**
 * How many times a week can a section meet?
 * @return that number
 */
public int getMaxMeetingsPerWeek() {
    return MAX_MEETINGS_PER_WEEK;
}

/**
 * How many digits are in the start times?
 * @return that number
 */
public int getMeetingStartTimeLength() {
    return MEETING_START_TIME_LENGTH;
}

/**
 * How many digits are in the end times?
 * @return that number
 */
public int getMeetingEndTimeLength() {
    return MEETING_END_TIME_LENGTH;
}

/**
 * How many characters are in the room identifiers?
 * @return that number
 */
public int getRoomNumberLength() {
    return ROOM_NUMBER_LENGTH;
}

/**
 * How many characters are in the department abbreviations
 * or course codes?
 * @return that number
 */
public int getDeptAbbreviationLength() {
    return DEPT_ABBREVIATION_LENGTH;
}

/**
 * Provide access to the singleton College object.
 * @return that singleton
 */
When Professor, Student, or Course objects are created, they are added to the appropriate collection. Thus, College must contain a number of remember methods.

/**
 * Remember a Professor to the collection.
 * @param p The Professor object to be remembered.
 */
public void remember(final Professor p) {
    theProfessors.put(p.getIdentifier(), p);
}

/**
 * Remember a Student to the collection.
 * @param s The Student object to be remembered.
 */
public void remember(final Student s) {
    theStudents.put(s.getIdentifier(), s);
}

/**
 * Remember a Course to the collection.
 * @param c The Course object to be remembered.
 */
public void remember(final Course c) {

theCourses.put(c.getSubjectAbbreviation() + c.getCourseNumber(), c);
}

Of course, these remember methods must be called by the appropriate constructors.

Let’s now flesh out what the backup method does.

Within College, it must go through all the Professor objects in the collection, write each to a separate file, and write the name of that file to the manifest. Each Professor object needs a method to write itself to a file and, since each Professor object inherits from Person, Person needs a method to write to a file.

Here is the method in College so far.

/**
 * Backup the collections the College maintains.
 * @param fileName the name of the manifest file to be generated
 */
public void backup(final String fileName)
throws IOException {
    // the manifest file
    PrintWriter manifest = new PrintWriter(
        new BufferedWriter(
            new FileWriter(fileName)));
    // the individual files
    PrintWriter out;
    int sequence = 0;

    // the Professors
    for (Professor p: theProfessors.values()) {
        // create a new name for the file to which the object
        // is written
        sequence ++;
        String profFileName = "Professor" + sequence + ".txt";
        // create the file
        FileWriter fstream = new FileWriter(profFileName);
        out = new PrintWriter(new BufferedWriter(fstream));
        // write to the file
        p.writeManifest(out);
        out.close();
        // write to the manifest file
        manifest.println(\"Professor=\" + profFileName);
        // continue to the next professor
    }

    // close the manifest file when all is completed
    manifest.close();
}

The method in Professor looks like this.
/**
 * Write a Professor object to the specified output device.
 * @param out The file used to persist the Professor
 */
public void writeManifest(PrintWriter out) {
    super.writeManifest(out);
    out.println("DATE_HIRED=" + dateHired.toString());
    out.println("RETIREMENT_DATE=" + retirementDate.toString());
}

The method in Person looks like this.

/**
 * Write a Person object to the specified output device.
 * @param out The file used to persist the Person
 */
public void writeManifest(PrintWriter out) {
    out.println("IDENTIFIER=" + identifier);
    out.println("FIRST_NAME=" + firstName);
    out.println("MIDDLE_NAME=" + firstName);
    out.println("LAST_NAME=" + firstName);
    out.println("FULL_NAME=" + firstName);
    out.println("PREFERRED_NAME=" + preferredName);
    out.println("ADDRESS=" + addr.toString());
    out.println("BIRTH_DATE=" + birthDate.toString());
}

When problems become too hard or when textbooks become too long, authors often say “it is left as an exercise for the reader to …” In this case, the exercise involves the following:

- Complete the backup method so it processes Student and Course objects.
- Create a restore method within the College class.
- Ensure that the restore method works. In particular, the birth date is written out as a String but is restored into a MyDate object. Ensure that addresses, which are written out on several lines, are restored correctly.
- Ensure that all collections in all classes are backed up.
- It would be nice if the manifest file were to contain a line stating when the file was created.

Further study

If you are interested in exploring the subject of file input and output further, look at the Java IO tutorial at [http://download.oracle.com/javase/tutorial/essential/io/index](http://download.oracle.com/javase/tutorial/essential/io/index).
Persistence via databases

There is another option to providing persistence, by writing data into a database. If you are interested in databases, start exploring Java Database Connectivity at http://www.oracle.com/technetwork/java/javase/tech/index.

This topic is much too complicated to consider in an introductory course.

Summary

The chapter covered what is perhaps the most-advanced topic in the book. But it is a topic which will become more and more important as the role of XML increases.

Now that we know how to create classes, and how to persist them, it is time to look at how we allow the users to provide the data to create classes. In particular, we need to see how to build data entry screens.
Exercises

1. Implement the clone method in the Student and Meeting classes. You will need it for the Student class when you use XML to persist the College class.

2. Check your knowledge of XML serialization by using it with some of the other classes we have in our college model.

   Remember that we have about 10 different classes from which you can select. Of course, if this were a real model, you would need to implement XML serialization for all of the classes.

3. In previous chapters, we discussed modelling a bank account.

   Implement the XML serialization of a Transaction.

   Implement the XML serialization of a BankAccount.

4. In previous chapters, we discussed modelling the sightings a birder makes.

   Implement the XML serialization of a Sighting.

   Implement the XML serialization of a YardList.

5. Explore the topic of Scalable Vector Graphics. A good place to begin is http://www.w3.org/Graphics/SVG/

6. Explore the topic of XSL and XSLT. A good place to begin is http://www.w3.org/Style/XSL/

7. Write a method to process a folder and determine how many files it contains. Display that number in the Terminal window.

   For a more challenging task, write a method to process a folder and determine how many files it and all its subfolders contain.

8. Write a method to input a text file and determine the number of unique words (ignoring capitalization) in the file. You will probably want to use a Set to help you. For a source of large text files, consider http://www.gutenberg.ca/.

   For a more challenging task, write a method to input a text file and determine not only the unique words but also how many times each appears. You will probably want to use a Map to help you.
9. Write a method to input a text file and write its contents to another file, converting all the text into uppercase.

10. Write a method to input a file containing the source code for a Java class, count the number of lines in the file, count the number of comments in the file, and determine the percentage of the file which is comments.

11. Write a method to input the text file created by the writeFile method of WriteFileExample. Ensure that the numbers in it are extracted correctly.

12. Explore what happens if the fields within a CSV file themselves contain commas. How would you adjust your methods for writing the file?

13. Explore the topic of using databases to persist Java objects. A good place to begin is http://www.oracle.com/technetwork/java/javase/tech/index.
Chapter 16 – Creating a GUI: data entry screens, part 1

Learning objectives

By the end of the chapter you will be able to:

- Describe the parts of a Graphical User Interface (GUI).
- Use Swing components to create data entry screens
- Describe some of the considerations involved in data entry screen design
- Describe the role of a layout manager in creating data entry screens
- Use a layout manager to create data entry screens
- Use ActionEvents and ActionListeners
- Use InputVerifiers

Warning to the reader

This chapter and the two following it contain information which some people feel should not be in an introductory course. I disagree. You may omit these three chapters should you wish.

However, since we have been discussing saving and retrieving objects, I feel we should also look at how the objects receive their values in the first place. After all, they won’t always have their values set in unit tests.

Introduction

You may have noticed that the way we have been running our program is not the way you normally run a program. While BlueJ uses windows and buttons, the code we have been creating has no windows, no buttons, and no labels. We need to develop a way in which the program can present a prettier face to the world.

We need to explore the world of the graphical user interface. An alternative name which is becoming common is “user experience.”

GUI

A graphical user interface (usually abbreviated GUI and pronounced “gooey”) is one way by which we can enter data that exists in the real world into the program we have created, which models that real world.
BlueJ uses a GUI as do most of the programs you run. The major parts of a GUI are:

- A frame which contains all the pieces of the interface,
- Panels, which divide the interface into logical sections,
- Labels, which identify data you should provide,
- Textboxes, which provide a space where you enter data,
- Buttons, which provide a way to enter commands using a mouse or other pointing device,
- Radio buttons, which allow you to make a single choice from a collection,
- Checkboxes, which allow you to make zero or more choices from a collection, and
- A layout manager, which helps you position all the parts of the interface.

These parts are standard (although the terminology may change somewhat) regardless of the language you are using to create the GUI and regardless of the GUI.

GUIs are built using a toolkit of predefined widgets. “Widget” is a generic term, with several meanings. When I looked up the word in Wikipedia (on 2006-05-23), I found two appropriate meanings. “A placeholder name for any unspecified device or good” and “a component of a graphical user interface that the user interacts with.” The two names are actually related; rather than refer to “labels, radio buttons, and checkboxes”, you might simply call them widgets.

The toolkit you use to build a GUI provides the widgets, but it also provides a “look and feel.” That is, it provides a specific style of widget. Windows programs have a different look and feel from those designed to run on a Macintosh. Even within Windows, some progress bars have square ends while others have rounded ends, and different versions of Windows have a different look and feel.

The toolkit we will use is called Swing, and is available in the javax.swing package. To quote from the documentation, the package “Provides a set of 'lightweight' (all-Java language) components that, to the maximum degree possible, work the same on all platforms.”

An earlier toolkit (Abstract Window Toolkit, or AWT) contained widgets which used code that was dependent on the processor you were using.

Within the Swing toolkit, the widgets mentioned above are called JFrame, JPanel, JLabel, JTextField, JButton, JCheckBox, and JRadioButton.

In addition to these widgets we will (in later chapters) use JApplet, JMenuBar, JMenu, JMenuItem, and JPanel.

There are other widgets too, but we won't be using them.
In case you missed it, the widgets in Swing all have names beginning with the letter J.

Layout managers, which we will see in a few moments, are not widgets.
Some HCI considerations

HCI is an abbreviation of Human-Computer Interaction. Some prefer to call it CHI. (One such group is the ACM’s Special Interest Group on Computer-Human Interaction, SIGCHI. 
http://www.sigchi.org/.)

Whatever the abbreviation, the field involves decisions you should consider when building an interface which will be used by a person to interact with a computer.

Here is a brief list of some of the HCI issues you should consider.

- Most Western cultures read from left to right, and top to bottom.

  That means that the important information a user must enter should be at the top, and any buttons should be at the bottom. It also means that those rules don’t apply when your market is a culture that is not Western. It also means that a product designed for international use should be customizable.

- A cluttered screen is not good; it confuses people. Your mother was right, neatness counts.

- Columns are good, random (or non-columnar) arrangements are bad.

- Balance is good.

  The left and right sides of a screen should be balanced with approximately the same number of visible widgets and the same amount of whitespace (the background) on each. It’s okay when the top of the screen contains more than the bottom.

- Menus, which we will explore in the next chapter, should use standard terms, and they should be in standard places. Commonly, File is the first item on a menu, Help is the last.

  Java is designed to run on many different operating systems, so the GUI for your program should respect that. Don’t present an interface suitable for a Macintosh unless the computer actually is a Macintosh.

- Colours are nice, but garish is not nice.

  Many people, particularly men, are colour-blind, so pay attention to the colours you use. When you are colour-blind, you may not see colours the same way others see them. A common problem is an inability to distinguish between red and green. The Wikipedia article on color blindness goes into a great deal of detail on the subject of colour-blindness.
Layout managers

Some of the rules earlier affect the positioning of widgets within your frame. A *layout manager* helps us to follow these rules. What is a layout manager?

To once again quote the Java tutorial, a layout manager “[d]efines the interface for classes that know how to lay out Containers.” A container is something (like a JPanel or a JFrame) into which you can place widgets, including other containers, and a layout manager controls the placement of those widgets.

There are many different layout managers, some very simple and some very complicated.

FlowLayout is one of the simplest layout managers. It places its widgets in left-to-right (or right-to-left, should you prefer) order. This can be very useful when you are placing a series of buttons side-by-side. It can be useful if you are placing checkboxes beside each other.

While this layout manager is often used for widgets in a single line, it will also serve for multi-line layouts. If there are too many widgets on one line, the next widget will simply continue onto the next one. Of course, if you resize the container, widgets may move from one line to another. This could be confusing for many of the people using your GUI.

Of course, FlowLayout also works for columns of widgets. Consider the buttons on the left side of the main BlueJ window. Could they be laid out using a FlowLayout layout manager. Since the BlueJ source code is available from [www.bluej.org/download/source-download.html](http://www.bluej.org/download/source-download.html), check it yourself.

We will use FlowLayout only in special situations, typically for the buttons at the bottom of a JFrame.

BorderLayout breaks its container into five regions, North, South, East, West, and Center, and allows you to place widgets in each. This can be useful; place the buttons on your screen in the South region, for example. We will use BorderLayout in a later chapter, where we design a screen that has three sections vertically, which we can conveniently place in the North, South, and Center regions.

GridLayout might be useful should you be creating a Minesweeper®-clone, a calculator, a telephone, or a chess game. It divides the frame into equal-sized rectangles. For example, if you have a JFrame called *jf* and you use GridLayout in the following statement

```java
jf.setLayout(new GridLayout(5, 4));
```

you will find the screen divided into 20 small rectangles, all of the same size, five rows each containing four rectangles.

Because the rectangles are the same size, and that doesn’t lend itself to the type of GUI we will be building, we won’t be using GridLayout.
GridBagLayout is useful when you are creating a more complicated screen, but it is very complicated to use. Ideally, you will find a tool which lets you draw the screen and then creates the appropriate Java code for you. (Search the web for Java GUI builder to get an idea of what is available. Yes, there are builder tools which will assist us in laying out a GUI. However, we won’t use them here. The experience of laying out a GUI yourself is something everyone should do at least once, if only to understand the underlying principles.)

GridBagLayout is possibly the most-complicated layout manager.

I prefer a layout manager which lets you place widgets in relation to one another, or in relation to their container. Java provides a couple of choices.

 GroupLayout is described in the Java tutorial as “… a layout manager that was developed for use by GUI builder tools, but it can also be used manually.”

Java also provides the SpringLayout class, but its documentation clearly advises against writing code using SpringLayout. “Due to the painstaking nature of writing layout code by hand, it is strongly recommended that you use the SpringLayout layout manager combined with a builder tool to lay out your GUI.”

One of the strengths of Java is that you can add new features to it, simply by providing a new library which implements that feature. That’s why there are so many layout managers; everyone has his or her own idea of the way it should be done.

One layout manager, which matches the way I think but is not supplied by Oracle, is RelativeLayout, described several (eight!) years ago at http://www.onjava.com/pub/a/onjava/2002/09/18/relativelayout.html?page=1. A more-detailed explanation about how RelativeLayout works is at http://www.onjava.com/pub/a/onjava/2002/11/27/layout2.html

While this layout manager has many features, it is also a little challenging to use. A similar, but easier-to-use, layout manager is available from http://code.google.com/p/relativelayout/.

The webpage describes Google's the layout manager using these words. “RelativeLayout is a free layout manager for Java SE 1.5.0 and higher (with limited support for Java 1.4.2). Rather than relying on grids or nested boxes, RelativeLayout lets you lay out components relative to each other and the window that contains them. This turns out to be a much more intuitive way of going about it, and virtually any interface you can imagine can be easily created using this way.”

This chapter, and the subsequent two chapters, will use the layout manager from Google. However, like SpringLayout, GroupLayout, and GridBagLayout, using RelativeLayout requires attention to detail.
Installing the RelativeLayout layout manager

To use RelativeLayout, you need to download the two jar files Jama.jar and RelativeLayout-v1.0.jar from http://code.google.com/p/relativelayout/; remember where you save them. A jar file contains compiled Java classes. You can think of it as a zipped file of classes. In fact, you can open it with a standard compression program, if you want.

To let BlueJ know that the jar files are available, select Tools, Preferences, Libraries. Select Add. Navigate to the jar files. Click one and then select Open. The name appears in the list of libraries. Repeat for the other. Select Ok.

A message box appears; what it means is “You can’t use RelativeLayout until you restart BlueJ.” So exit from BlueJ and restart it.

Should you be a cautious person, select Tools, Preferences, Libraries again. Yes, RelativeLayout is shown as being available.

Actually, you should download a third jar file, RelativeLayout-v1.0-src.jar which is the one that contains the source code for the layout manager. We don’t need the source code (unless we are curious) but we need the documentation which comes with the source code. Download the jar file. I unzipped mine and saved its contents somewhere I could find it again, so I can refer to its documentation, just as I do with all other Java documentation.

Creating a data entry screen

For our first data entry screen we should begin with a simple example. Which is our simplest class? That is, which has the fewest instance variables?

I think MyDate provides a good example of the process of creating a simple GUI. Note that this GUI would not be part of a real program; it is being used simply as a teaching device.

You will want to review the website cited earlier to have some idea how to use RelativeLayout and its bindings (or constraints.) To summarize, what we will be doing is identifying widgets and then positioning each in terms of the container and/or the other widgets in the container.

How the data entry screen should appear

How should our data entry screen look? Let’s make something simple, JLabels in one column identifying the data to enter, and JTextFields in a second column providing space to enter the data. Some of the spacing may not look right; it’s determined by some defaults in the layout manager. Perhaps the data entry screen would look something like the following.
There are several things wrong with the illustration. The first is that there is probably too much space between a label and its text field. Can you spot the other errors? There at least two.

**Import statements**

To use the RelativeLayout layout manager, here are the import statements.

```java
import edu.cmu.relativelayout.*;
import edu.cmu.relativelayout.matrix.*;
```

This provides the layout manager and its associated components.

To use Swing components, providing the widgets, here are two additional import statements we need.

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

The first of these additional statements provides the widgets we will be using. These are “better” widgets than those in the Abstract Window Toolkit, or AWT.

The second provides the AWT events to which we can respond. More details on events will be given later.

**Creating the JFrame**

We begin by creating what we see - the JFrame (including a title bar, and minimize, maximize, and exit buttons), the labels (implemented as JLabels), and the text fields (implemented as JTextFields).
Below is a method to create the JFrame, the JLabels, and the JTextField. To summarize the process:

- Create the JFrame and attach a layout manager to it.
- For each widget you wish to add to the JFrame, create the widget and the constraints which specify its location. As we are using a relative layout, the constraints are relative to the JFrame or to other widgets.
- Attach the widgets to the JFrame and the constraints to the layout manager.

The method is explained line-by-line on subsequent pages.

```java
/**
 * create a JFrame to allow data entry.
 */
public static JFrame buildFrame() {
    // create a frame
    JFramejf = new JFrame("Enter MyDate");
    jf.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);

    // and set its layout manager
    jf.setLayout(new RelativeLayout());

    // create a BindingFactory to create various Binding objects (constraints) to control the
    // position of the widgets
    BindingFactorybf = new BindingFactory();

    // create the first label
    JLabellabelYear = new JLabel("Year – four digits");

    // and its bindings (constraints)
    // leftEdge is relative to the last leftMargin established. We will use the default left margin
    // which is the left edge of the frame.
    // ditto for top edge.
    BindingleftEdge = bf.leftEdge();
    BindingtopEdge = bf.topEdge();

    // create a RelativeConstraints object, a collection of constraints
    RelativeConstraintslabelYearConstraints = new RelativeConstraints();

    // add the constraints to the collection
    labelYearConstraints.addBinding(leftEdge);
    labelYearConstraints.addBinding(topEdge);

    // add the label and its collection of constraints to the frame
    jf.add(labelYear, labelYearConstraints);

    // second label
    JLabellabelMonth = new JLabel("Month – two digits");

    // and place the label below the first.
    topEdge = bf.below(labelYear);
```
// the leftEdge Binding makes the two labels share a common left edge.
leftEdge = bf.leftAlignedWith(labelYear);

RelativeConstraints labelMonthConstraints = new RelativeConstraints();
labelMonthConstraints.addBinding(topEdge);
labelMonthConstraints.addBinding(leftEdge);

jf.add(labelMonth, labelMonthConstraints);

// third label
JLabel labelDay = new JLabel("Day – two digits");
topEdge = bf.below(labelMonth);
leftEdge = bf.leftAlignedWith(labelMonth);

RelativeConstraints labelDayConstraints = new RelativeConstraints();
labelDayConstraints.addBinding(topEdge);
labelDayConstraints.addBinding(leftEdge);

jf.add(labelDay, labelDayConstraints);

// firstTextfield (year), with room for four characters
JTextField textYear = new JTextField(4);

// text fields will begin 200 pixels from the left edge of the frame. Set that
// as the default
bf.setLeftMargin(200);

leftEdge = bf.leftEdge();
topEdge = bf.topAlign(labelYear);

RelativeConstraints textYearConstraints = new RelativeConstraints();
textYearConstraints.addBinding(leftEdge);
textYearConstraints.addBinding(topEdge);

jf.add(textYear, textYearConstraints);

// ditto for a second text field (month), allowing space for two characters
JTextField textMonth = new JTextField(2);

topEdge = bf.topAlign(labelMonth);
leftEdge = bf.leftAlignedWith(textYear);

RelativeConstraints textMonthConstraints = new RelativeConstraints();
textMonthConstraints.addBinding(topEdge);
textMonthConstraints.addBinding(leftEdge);

jf.add(textMonth, textMonthConstraints);

// ditto for a third text field (day)
JTextField textDay = new JTextField(2);

topEdge = bf.topAlign(labelDay);
leftEdge = bf.leftAlignedWith(textYear);

RelativeConstraints textDayConstraints = new RelativeConstraints();
textDayConstraints.addBinding(topEdge);
textDayConstraints.addBinding(leftEdge);

jf.add(textDay, textDayConstraints);

// done
return jf;
}

Making the JFrame visible

Finally, here’s a method to create and make the JFrame visible, and run successfully within an event-based environment. We'll define event in a moment, when we look at the method in detail.

/**
 * show the Frame.
 */
public static void showFrame() {
    Runnable runner = new Runnable() {
        public void run() {
            JFrame box = buildFrame();
            box.setLocation(100, 100);
            box.setPreferredSize(new Dimension(500, 300));
            box.pack();
            box.setVisible(true);
        }
    };
    EventQueue.invokeLater(runner);
}

You can copy these methods and place them within the MyDate class if you'd like to see them in action.

Explaining the code

You may have noticed that I have adopted some naming standards:

- JLabel variables all have names beginning with the word label. For example, the label associated with the JTextField for the month is named labelMonth.
- JTextField variables all have their names beginning with the word text. For example, the JTextField into which you enter the month is named textMonth.
- RelativeConstraint variables all have their names beginning with the name of the variable they are constraining. For example, labelMonth is positioned using labelMonthConstraints.
- Individual Binding objects, which are placed in a RelativeConstraint object, have names leftEdge, topEdge, rightEdge, and bottomEdge, regardless of the widget they are describing.
I suggest you use a similar scheme and stick to it. Perhaps you'll use lbl instead of label, but that prefix still helps separate labels from other widgets. This naming convention is called Hungarian Notation. ([http://en.wikipedia.org/wiki/Hungarian_notation](http://en.wikipedia.org/wiki/Hungarian_notation))

Let’s examine these methods in some detail. We’ll give a few lines of code and then explain them.

Notice that creating or building a frame is in one method, and displaying it is in another.

**The JFrame**

Let’s begin with the creation of the JFrame.

```
final JFrame jf = new JFrame("Enter MyDate");
```

You have seen frames whenever you have used a computer with a windowing system. The Java tutorial says “A frame, implemented as an instance of the JFrame class, is a window that typically has decorations such as a border, a title, and buttons for closing and iconifying the window. Applications with a GUI typically use at least one frame.”

The JFrame we create has the title “Enter MyDate.”

```
jf.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);
```

When you click the X button in the top right, the JFrame will hide itself. Thus it will be available quickly should you need it again.

```
If you will be reusing JFrame objects, you may wish to use the Singleton pattern to ensure that duplicates are not created.
```

If you will not be reusing the JFrame, use this statement instead.

```
jf.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
```

**The layout manager**

```
jf.setLayout(new RelativeLayout());
```

Associated with the JFrame are the rules that govern the placement of widgets in the JFrame, as implemented by a layout manager. The RelativeLayout layout manager will interpret those rules for us.

**BindingFactory**

Think of a Binding as an instruction which the layout manager uses to control the positioning of widgets. The package documentation says “A Binding represents a dependency of some
component's position on the position either of another component or of the surrounding container.”

Think of a BindingFactory as a class which creates bindings to our specifications. It also tracks the relationship between widgets. The package documentation says “A factory for quickly creating Bindings to lay out simple interfaces.”

```java
// create a BindingFactory to create various Binding objects (constraints) to control the
// position of the widgets
BindingFactory bf = new BindingFactory();
```

When you look in the documentation for BindingFactory, you’ll find there is a default BindingFactory object and you can access it. Thus, the last statement above could be replaced with this one.

```java
BindingFactory bf = BindingFactory.getBindingFactory();
```

This is different from the Singleton pattern. Singleton allows you no access to the constructor; BindingFactory has two constructors you can use.

**The labels**

Continuing through the method, we create and place widgets, starting with a label. For each piece of data we wish to enter, there should be a label.

```java
// create the first label
JLabel labelYear = new JLabel("Year – four digits");
```

The way in which we create and place widgets is the same for all widgets.

- Create the widget which we wish to place on the frame.
- Create the bindings for the widget.
- Create a collection for the bindings and add the bindings to the collection.
- Add the widget and its collection of bindings to the container.

In the code above, we create a JLabel containing the words “Year – four digits”.

If you had only three widgets I suppose you could name them Huey, Dewey, and Louie, after Donald Duck’s nephews. (Perhaps it’s time for a joke about getting your ducks in a row.) If you had many widgets, I suppose you could name them after the letters in the Greek alphabet (alpha, beta, gamma, delta, and so on.) But neither naming strategy is a good one.

Note that you can get creative with the captions on your JLabels: you can embed HTML commands in the string you provide to the constructor. For example, new JLabel("<HTML><i>Cancel</i></HTML>") creates a JLabel whose text is italicized. The tag </HTML> is not required.
More details about embedding HTML in Java widgets is available in the Java tutorial, [http://download.oracle.com/javase/tutorial/uiswing/components/html.html](http://download.oracle.com/javase/tutorial/uiswing/components/html.html).

There are several constructors for JLabels, and they include ones which allow you to specify an image on a JLabel.

**The label Bindings**

We create two bindings to describe the position of the JLabel.

```java
Binding leftEdge = bf.leftEdge();
Binding topEdge = bf.topEdge();
```

The most common bindings are those which specify the top edge of the widget and left edge of the widget. The bottom and right edges are determined by the size of the widget; for labels, the size is determined by the words you wish to place in the label as well as the typeface/font you use.

Since RelativeLayout is a relative layout manager, we usually position widgets relative to other widgets. But one or more widgets must be anchored to the frame itself. We use this first JLabel as the anchor and then position other widgets relative to it.

The first binding (constraint) positions the left edge of the JLabel a certain number of pixels from the left edge of the JFrame. This margin is the default value; you could modify it if you wanted more or less space between the widget and the edge of the form by using the `setLeftMargin` method within the BindingFactory class.

What is the default value for the left margin?

The second binding positions the top edge of the JLabel a certain number of pixels down from the top edge of the frame. This could be modified.

What is the default value for the top margin?

The upper left corner of the JFrame is the origin (or 0, 0) in the layout manager's measurement system. If we specify numeric pixel values (as we will soon), a positive value represents measurement to the right or down, depending on the direction of motion. Negative values represent measurement to the left or up.

Also, note that we are positioning the widget in an absolute position; it may be better to position it, say, 10% of the way down the JFrame. This layout manager supports that method of placement but to do that it you must determine the size of the JFrame and then multiply that by 10%. See the details later.

We could create bindings describing the height and width of the JLabel but there is no need to do so. JLabels have a default height (depending on the typeface/font), and their width expands to display the contents.
Once we have created the bindings for the label, we place those labels in a collection.

```java
RelativeConstraints labelYearConstraints = new RelativeConstraints();
labelYearConstraints.addBinding(leftEdge);
labelYearConstraints.addBinding(topEdge);
```

Finally, we add the label and its collection of bindings to the frame.

```java
jf.add(labelYear, labelYearConstraints);
```

When the JFrame is made visible, the layout manager will process all the bindings and position the widget properly.

Now that we have our first widget anchored relative to the frame, we can place the other widgets relative to the anchor.

**The other labels**

Now we create another JLabel, one representing the month.

```java
JLabel labelMonth = new JLabel("Month – two digits");
jf.add(labelMonth, labelMonthConstraints);
```

We need to create a binding describing its top edge, saying, that it is below the year JLabel. How much below is determined by the rules incorporated in the BindingFactory. You can use the setVerticalSpacing method to change this if you wish.

```java
topEdge = bf.below(labelYear);
```

What is the default vertical spacing?

We create a second binding, ensuring its left edge lines up with the left edge of the previous label. Note that some people prefer to have the right edges of the labels line up. How would you do that? Hint: rightAlignedWith().

```java
leftEdge = bf.leftAlignedWith(labelYear);
```

We create the collection of bindings.

```java
RelativeConstraints labelMonthConstraints = new RelativeConstraints();
labelMonthConstraints.addBinding(topEdge);
labelMonthConstraints.addBinding(leftEdge);
```

I am creating different collections for the bindings of each widget, but I could reuse the same collection if I wished. How do I clear the bindings from the collection?

Finally, we add the label and its collection of bindings to the frame.

```java
jf.add(labelMonth, labelMonthConstraints);
```
The layout manager will tell us, via InconsistentConstraintExceptions and AmbiguousLayoutExceptions, whenever we provide inconsistent information (widget B is placed relative to widget A, but widget A is placed relative to widget B, for example) or too little information about the positioning of a widget. Too little information usually comes when we are sloppy in our cutting and pasting, and don’t change all the names we need to change, thus leaving one of the bindings attached to the wrong widget.

The code to create the third JLabel is almost identical to the code for the second, so is omitted here.

**JTextFields**

Once we have the JLabels in position, we deal with the JTextFields, where you enter data.

```java
JTextField yearText = new JTextField(4);
```

We create the JTextField, giving it enough space to accommodate four characters. This width of four characters is based on the average width of the characters in the font you are using; after all, a “W” is wider than an “I”. Thus the actual width (in pixels) will vary depending on the font you are using; a font is the combination of typeface and point size. In the same way, the width (and height) of a label is determined by the typeface and point size.

We create the appropriate bindings, and add them to the JTextField’s collection of bindings.

```java
// text fields will begin 200 pixels from the left edge of the frame
bf.setLeftMargin(200);

leftEdge = bf.leftEdge();
topEdge = bf.topAlign(labelYear);

RelativeConstraints textMonthConstraints = new RelativeConstraints();
textMonthConstraints.addBinding(topEdge);
textMonthConstraints.addBinding(leftEdge);
```

Note that RelativeConstraints contains two methods for adding Bindings. We have used the first, which adds one Binding to the collection. But there is also an addBindings method which allows us to add more than one Binding at once. Thus, we could use

```java
textMonthConstraints.addBindings(topEdge, leftEdge);
```

The order in which we add the bindings does not matter.

Finally, we add the text field and its collection of bindings to the frame.

```java
jf.add(textMonth, textMonthConstraints);
```
Other text fields

Similarly, we place the other two JTextFields on the content pane. Their top edges are aligned with the correct labels, and their left edges are aligned with previous text fields.

Note that a JTextField is designed for times when the input consists of only one line of input. If you need a widget which can accept several lines of input, consider a JTextArea, a JTextPane, or a JEditorPane.

A JTextArea allows many lines of type, all in the same font. Unfortunately, the JTextArea expands on the screen as you add more and more lines. To limit its size, place it within a JScrollPane.

A JEditorPane allows you to edit text while using bold or italic text in different fonts and colours.

A JTextPane is similar to a JEditorPane, but you'll have to explore the similarities and differences yourself.

There is a special type of JTextField called a JPasswordField. If you use a JPasswordField, the text you type is replaced with asterisks (stars) and you cannot cut or paste the text you entered, as you can from a JTextField. You can paste text into a JPasswordField.

Done!

Once we have created the JFrame, we return it to the calling method.

Summarizing building a data entry screen

This is a simplified version of the process. A more complete version appears in the next chapter.

The first step is to create a JFrame and associate a layout manager with it.

The second step is actually a series of steps, repeated for all widgets.

- Create the widget which we wish to place on the frame.
- Create the bindings for the widget.
- Create a collection for the bindings and add the bindings to the collection.
- Add the widget and its collection of bindings to the container.

Displaying the frame

Note that we have described the creation of a JFrame; we have not yet specified the size of the JFrame. We have not specified its location of the screen. We have not even made it visible.

You might say we have created the blueprint for the JFrame and now we need another method to size it, position it, and make it visible. The showFrame method does that.
Let’s examine its statements in a moment. First we need to say a little about threads.

**Threads – definition**

Wikipedia ([http://en.wikipedia.org/wiki/Thread](http://en.wikipedia.org/wiki/Thread)) defines a thread as “a way for a program to split itself into two or more simultaneously (or pseudo-simultaneously) running tasks.”

Have you ever been working in your word processor and made a spelling or grammar mistake? A word appears underlined (in my case in red) to indicate a spelling mistake. A phrase appears underlined (in my case in green) to indicate a grammar concern.

A spell-check thread is running at the same time as a grammar-check thread (or maybe one thread is doing both checks. I don’t know for sure.); both are running at the same time as the thread that allows me to type the document. Simultaneously, other processes are running on my computer, but these processes are part of another program, which itself may have threads (checking for appointments and incoming mail, for example.)

The Java interpreter uses a garbage collection thread to identify objects for which there are no references. It then reuses the memory for new variables.

The operating system on your computer gives each thread a small amount of time to run and then switches to another thread.

You’ll see more details about how threads and processes (and processors) interact in an operating systems course. For now, we need to consider how threads work in Java.

**Threads and Swing**

Threads are most relevant for our data entry screens, and, as we will see later, menus, and applets.

When we create a data entry screen, using Swing (or AWT) components, we respond to events which occurred to these components. In the background, an event-dispatch thread is listening for events and then notifying the appropriate handlers.

Events are placed in a queue (first in, first-out) and the event-dispatch thread removes and processes them as appropriate.

Thus, events may not be handled immediately if a previous event is being handled slowly or if many events are happening at the same time.

**Runnable**

The Runnable interface allows our program to run safely in an environment where multiple threads (activities) are happening simultaneously.
The thread we are most interested in is the event-dispatch thread. Common events on a data entry screen include a user clicking a button, or entering or leaving a JTextField. Almost everything to do with GUIs is handled through the event-dispatch thread.

By creating a Runnable object, and placing that object on the event-dispatch thread, we are ensuring that our GUIs will work properly, particularly when we have a large and complicated GUI, such as the one we are beginning to develop. By “run properly” we mean that events will be handled in the correct order.

A Runnable object implements the run method. The run method creates the GUI. We place the Runnable object in the event-dispatch thread by the final statement of showFrame,

```
EventQueue.invokeLater(runner);
```

The Runnable interface comes from java.lang so we do not need to import it. But EventQueue comes from java.awt.EventQueue, so you need an import statement to import java.awt.EventQueue; or import java.awt.*;

With that background, we can examine the rest of showFrame.

**Create the frame**

```
JFrame box = buildFrame();
```

First we create the JFrame using the method we just finished examining, giving it the name box.

**Position and size the frame**

```
box.setLocation(100, 100);
```

We specify the location of the top left-hand corner of the frame (100 pixels in from the left edge of the screen, 100 pixels down from the top of the screen).

```
box.setPreferredSize(new Dimension(500, 300));
```

We specify its preferred size (500 pixels wide by 300 pixels high.) It can be resized, of course. You’ll need another import statement here.

```
import java.awt.Dimension;
```

If you used import java.awt.*; to import EventQueue, Dimension will also have been imported by that statement.

As I write these words, I’m running my screen at 1280 by 1024 resolution, so the JFrame will be approximately 40% the width of the screen, and 35% of its height. If you are using a resolution
larger than this, your JFrame will occupy less of the screen. If you are using a resolution lower than this, your JFrame will occupy more of the screen.

```java
box.pack();
```

The `pack` method is from the Window class (A JFrame is a Frame is a Window.) and that class describes its function as “[c]auses th[e] Window to be sized to fit the preferred size and layouts of its subcomponents.”

This method takes all the widgets and constraints and processes them into what we see when we display the frame.

**Make it visible**

```java
box.setVisible(true);
```

And then we make it visible. There may be times when you wish to make it invisible, as something else becomes visible. Simply change the parameter to `false`.

**Displaying the frame**

Compile MyDate, right-click the class in the class diagram, and run the `showFrame` method.

Note that your frame will be a little different from the one shown above. In particular, the vertical spacing will be different. I used a larger spacing to make the contents of the frame easier to read. For example, I used

```java
topEdge = new Binding(Edge.TOP, 25, Direction.BELOW, Edge.BOTTOM, labelYear);
```

to specify the top edge of the `labelMonth` widget. We will see more bindings like this below and in subsequent chapters.

**Answering some questions**

**How do you place a widget 10% of the way down the JFrame?**

The short answer is that you cannot by specifying 10%. But you can calculate 10% of the height of a JFrame and then create the appropriate constraint.

To determine the size of a JFrame, you would expect to use `getSize`, `getHeight`, or `getWidth` methods. A JFrame does not have such methods. A JFrame is a Frame, which does not have these methods. A Frame is a Window, which does not have these methods. A Window is a Container, which does not have these methods. A Container is a Component, which does have these methods. Thus, Container, Window, Frame, and JFrame objects all have access to `getSize`, `getHeight`, and `getWidth` methods, through inheritance.

Thus `jf.getHeight()` is the height of the frame and `jf.getWidth()` is its width.
However, the JFrame does not have a size until we give it one in showFrame. To resolve the problem, you could add the height and width as parameters to buildFrame. Then you can use percentages of the frame height to specify top edges of widgets, and percentages of the frame width to specify left edges.

**How do we change the position of the JFrame when it appears?**

First, of course, we must set the preferred size of the JFrame.

Once that is done, the following method call places the frame in the centre of the screen.

```java
box.setLocationRelativeTo(null);
```

More correctly, that statement places the upper left hand corner of the frame in the centre of the screen with the frame itself to the right and below.

The following statement gives a specific location on the screen. The left edge of the JFrame is 700 pixels from the left edge of the screen, and the top of the JFrame is 400 from the top of the screen.

```java
box.setLocation(700, 400);
```

Normally, I would use setLocation with small values (like 100) for the arguments, placing the JFrame near the top left corner of the screen. Depending on your screen resolution and the size of the frame, having the left edge at 700 pixels may mean that part of the frame is invisible since it is off your screen.

Of course, should the person using your program not like the size or position of your frame, the mouse (or other pointing device) is available to resize and reposition it.

Should you really want to place the frame in the centre of the screen, you can do so using the setLocation method. First, specify the size of the frame. Note that we need to modify the constructor by providing the frame's width and height.

```java
int height = 300;
int width = 500;
JFrame box = buildFrame(width, height);
box.setSize(new Dimension(width, height));
box.setPreferredSize(box.getSize());
```

Then determine the size of the screen.

```java
Toolkit toolkit = Toolkit.getDefaultToolkit();
Dimension screenSize = toolkit.getScreenSize();
```

Those two statements return the size of the screen. To use them, you will need to include the statement
import java.awt.Toolkit;

Finally, you need the statements

```java
int x = (screenSize.width - width) / 2;
int y = (screenSize.height - height) / 2;
box.setLocation(x, y);
```

**Is it appropriate to have the labels contain prompts about the corresponding JTextField?**

No, it isn’t.

It’s much better to use JToolTip, small balloons of text that appear when the mouse hovers over the JTextField. For example, you may wish to remove the hint from the label and add the statement to your method.

```java
textYear.setToolTipText("Please enter the year - four digits");
```

Tooltips are usually single lines of text. If you need a multi-line tooltip, use HTML to describe the tip. For example `textYear.setToolTipText("<HTML>Please enter the year<br>- four digits</HTML>");` produces a two-line tooltip. The closing tag `</HTML>` is not required.

How long does the mouse need to hover before the tooltip appears?

`ToolTipManager.sharedInstance().initialDelay(23);` indicates a delay of 23 milliseconds. The default is 750 milliseconds. To use ToolTipManager class, you’ll need to import `javax.swing.ToolTipManager`.

How long will the tooltip remain visible?

`ToolTipManager.sharedInstance().dismissDelay(2000);` indicates the tooltip will remain visible for 2000 milliseconds (two seconds) unless you move the mouse first. The default is 4000 milliseconds.

What happens if you move the mouse off the JTextField and then move it back on?

`ToolTipManager.sharedInstance().reshowDelay(1000);` indicates that the mouse must be off the JTextField for at least 1000 milliseconds (one second) for the tooltip to appear again. If you move the mouse back onto the JTextField in less than 1000 milliseconds, the tooltip will not appear. The default is 500 milliseconds.

**How do we use hot-keys to move to a specific JTextField?**

Sometimes, you have many widgets on the screen and you wish to move to a specific one, not by tabbing through the others and not by taking your hands off the keyboard to click the one you want. You just want to press a combination of keys which will take you to the correct widget.

Hot-keys allow you to do this.
We first specify the mnemonic, the letter in the JLabel caption which is to be underlined (if your interface underlines mnemonics) and which will function as a hot-key. Here we assign Y as the mnemonic representing year. The documentation of the KeyEvent class lists all the mnemonics you may use.

```java
labelYear.setDisplayedMnemonic(KeyEvent.VK_Y);
```

Then specify the JTextField with which the JLabel is associated.

```java
labelYear.setLabelFor(textYear);
```

When you press `Alt` and the mnemonic, the cursor will move to the JTextField.

Note that if the mnemonic you prefer occurs several times in the JLabel, you may specify which occurrence is underlined by using the setDisplayedMnemonicIndex method. For example, if the JLabel named labelName3 contains “Full (Legal) Name” and you wish to use the letter 'l' (the letter ell) as the mnemonic, you have four occurrences of that letter from which to make a choice. I'd suggest the third one, the capitalized one. `labelName3.setDisplayedMnemonicIndex(6)` will accomplish that, since 'L' is the character in position 6, starting from 0 of course.

Note also that some symbols are not suitable for mnemonics; they are too narrow. In that group, I would include the letters 'l' (lowercase), 'i' (both upper and lowercase), and f (lowercase). The number 1 is also unsuitable unless it is followed by a space. Letters with descenders (the parts that hang below the line) are also poor choices; the underlining is obscured by the descender. Thus, you want to avoid lowercase g, j, p, and q. The uppercase versions of these letters are fine to use as mnemonics.

**How do I start the frame maximized? Minimized?**

In general, I feel it is not a good idea to start a frame full-size. It’s rude, intruding on, indeed obscuring, whatever a person is doing. Should you wish to ignore the advice (and many programs do), import the java.awt.Frame class and then use this statement.

```java
box.setExtendedState(Frame.MAXIMIZED_BOTH);
```

If you wish the frame to start minimized, import the java.awt.Frame class if you have not already done so, and then use the statement

```java
box.setExtendedState(Frame.ICONIFIED);
```

But these are minor questions. There are two biggies – what about buttons (did you notice they were missing from the frame we built?), and what about editing the data provided?
**JButtons**

The JFrame we created is quite unusual. It has JLabels and JTextFields, but it has no buttons, which we will implement as JButtons. We would expect at least an OK button and a Cancel button.

I will name my buttons using the Hungarian notation, with the names of JButton objects beginning with button. Others may begin their JButton names with btn or cmd.

**Cancel button**

Let’s create the Cancel button first. Note the idiom we have seen throughout the chapter.

- Create the widget which we wish to place on the frame.
- Create the bindings for the widget.
- Create a collection for the bindings and add the bindings to the collection.
- Add the widget and its collection of bindings to the container.

**Create a JButton**

```java
JButton buttonCancel = new JButton("Cancel");
```

When we create a button, we provide its caption. This caption may be specified here, or it may be retrieved from a resource bundle. Resource bundles are a tool used to internationalize your application.

Note that you can also specify an image or an icon which should appear on your JButton. See the setIcon method.

You can also embed HTML commands in a button's caption, as we did with a JLabel. This allows you to be adventuresome in colours and fonts.

**Create a JButton’s bindings**

Buttons are usually placed (in a Western culture) in the lower right corner of the frame. Thus, the bindings we should use describe the position of the button's right edge and the button's bottom edge.

Note that we need to declare rightEdge and bottomEdge since we have not used those variables so far; the JLabel and JTextField objects were positioned used leftEdge and topEdge.

```java
Binding rightEdge = new Binding(Edge.RIGHT, 25, Direction.LEFT, Edge.RIGHT, jf);
```

The button's right edge is 25 pixels to the left of the right edge of the frame.

```java
Binding bottomEdge = new Binding(Edge.BOTTOM, 10, Direction.ABOVE, Edge.BOTTOM, jf);
```
The button's bottom edge is 10 pixels above the bottom edge of the frame.

Notice how the parameters in the method calls match the wording in the English statements.

Create a collection to hold the bindings.

RelativeConstraints buttonCancelConstraints = new RelativeConstraints();

Place the bindings in the collection.

buttonCancelConstraints.addBinding(rightEdge);
buttonCancelConstraints.addBinding(bottomEdge);

**Add a JButton and its bindings to the frame**

jf.add(buttonCancel, buttonCancelConstraints);

This is exactly the same process we used with all our other widgets.

**Create a mnemonic**

Buttons have mnemonics too, here the letter C.

buttonCancel.setMnemonic(KeyEvent.VK_C);

**The default button**

And one of the buttons should be the default, the button which will be deemed to have been clicked when you press Enter.

jf.getRootPane().setDefaultButton(buttonCancel);

Now, we have only one button so of course it is the default button. But when we have many (two or more) buttons the default button is usually the one which causes the least damage when it is pressed accidentally.

**Button action**

What should happen when we click the Cancel button (or press Enter)? The same actions as when we click the X button to exit the JFrame. How we make it happen follows.

**The OK JButton**

We use similar statements to create an OK button.

JButton buttonOkay = new JButton("Okay");
Use these constraints to place it to the left of the Cancel button.

RelativeConstraints buttonOkayConstraints = new RelativeConstraints();
// right edge of the button is 10 pixels to the left of the right edge of buttonCancel
rightEdge = new Binding(Edge.RIGHT, 10, Direction.LEFT, Edge.LEFT, buttonCancel);
// tops of the two buttons are aligned
topEdge = bf.topAlign(buttonCancel);

buttonOkayConstraints.addBinding(rightEdge);
buttonOkayConstraints.addBinding(topEdge);

And then add the button and its constraints to the frame.

jf.add(buttonOkay, buttonOkayConstraints);

What is the mnemonic for the OK button?

**Alternative layout managers**

We have used RelativeLayout to place these buttons. A later example will show how to place them within a JPanel, using FlowLayout.

**ActionEvents and ActionListeners**

How do we tell that a button has been clicked? This leads us to the world of ActionEvents and ActionListeners.

When a button is clicked, an ActionEvent occurs.

But buttons are not the only widgets which can cause events to occur.

A JTextField which has focus allows input. When a JTextField gains or loses focus, an ActionEvent occurs.

ActionListeners are executed in response to the ActionEvent.

**Creating an ActionListener**

We will create an ActionListener for each of our buttons. Let’s begin with an ActionListener for the cancel button. My standard is to begin the name of an ActionListener with the word listener.

```
ActionListener listenerButtonCancel = new ActionListener () {
    public void actionPerformed(ActionEvent e) {
        // do something
    }
};
```
Linking an ActionListener to a JButton

We need to connect the ActionListener to each JButton. The ActionListener will be notified each time the JButton is clicked.

buttonCancel.addActionListener(listenerButtonCancel);

What does an ActionListener do?

When an ActionEvent occurs, the ActionListener is informed and it performs an action of some kind. What should happen when we click the cancel button?

For now, let’s have it just tell us that it has been clicked. Replace

// do something

with

JOptionPane.showMessageDialog(jf, "Clicked cancel", "Cancel", JOptionPane.INFORMATION_MESSAGE);

The intent is that the method should simply display a message for us. But it doesn’t compile!

The error message is “local variable jf is accessed from inner class; needs to be declared final.” Without going into the details of inner classes, we’ll just follow its advice. Find the line where you declared jf and add the word final to it. Now it reads

final JFrame jf = new JFrame("Enter MyDate");

We have seen the word final before, in connection with the declaration of constants and as a promise that a method will not change the value of a parameter. But we are not declaring a constant here nor are we working with a parameter. We are using final in a different sense, as a promise that we will not change a variable within an inner class.

This use of final as a promise is something Checkstyle checks and it may get upset with our setters and constructors. The parameters to all setters and constructors should be final, as a promise that the setters and constructors will not be changing the values provided to them.

Go back through your code and ensure the parameters for your setters and constructors are final.

Compile MyDate, right-click the class in the class diagram, run the showFrame method, and then click either JButton. What happens?

Nothing happens when you click the okay button (unless you thought ahead and gave it an ActionListener.) But go ahead and give the okay button an ActionListener with appropriate behaviour; I’ll wait.
Now click the okay button. An informational message appears.

Note the different colours or shading in the titles of the two frames. The dialog is *modal*; you must click the OK button within the Okay dialog before you can continue and do anything else with the application.

You should use modal dialogs when it makes no sense to allow you to continue without responding. A good example is when you are installing software. You probably should not be doing anything else while installing new programs.

How does the dialog appear and how is it related to the statement you copied?

- The first argument to `showMessageBoxDialog` is the name of the parent component. The message which appears will be centred on that component. Should the name be `null`, then the message is centred on the screen. This may not be a good idea since it may not show a clear association between the parent and the message.
- The second argument is the message which appears beside the icon (the circle containing the letter “i”). This icon will change depending on the value of the fourth argument.
- The third argument is the title which will appear in the message dialog frame.
- The fourth argument indicates that the message is one of information only, and will result in an information icon appearing in the message dialog frame.
Behind the Cancel Button

But we wanted the cancel button to cancel the frame, to close it, not display a message.

Replace the showMessageDialog call in the ActionListener with

```java
jf.setVisible(false);
```

and the frame will no longer be visible when you click the cancel button. showFrame exits.

In a normal application, the program would not terminate. It would continue running with some other frame visible and the invisible one (or ones) waiting their chance to reappear.

While this technique creates a snappier application, it does use up memory, and you must remember to close the data entry frame eventually, perhaps in a finally block. This is done by using the dispose method in the Window class.

When you dismiss the GUI, sometimes you leave the Java virtual machine running. If that is the case, the “barber pole” lying on its side in the bottom of the BlueJ window is moving. Right-click it and choose Reset Java Virtual Machine.

To prevent the problem, use

```java
jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Editing data

There are two times as you are entering data when it may be appropriate to test it is reasonable. One is as it is being entered; we will explore that first. The other is when an Okay button is clicked; we will explore that later.

Why do we need both times? Consider the frame we just created.

We can check that the JTextFields contain only numbers.

We cannot check that the month JTextField contains only the numbers from 1 through 12 until the JTextField is complete.

We cannot check that the day JTextField is appropriate to the month (30 days hath September …) until all three JTextFields (year, month and day) have been provided, and we do not know the order in which the data will be provided.

Modern authors suggest that you not intercept keystrokes as they are typed, but you verify the contents of a JTextField as you attempt to leave it. This is done with an InputVerifier.
If you prefer to restrict a user so only specific keys can be pressed while the focus is within a widget, there are a variety of techniques. I do not recommend them but you can find them by searching for, for example, JTextField numeric.

But how do we check that all required fields have been entered? That must be done when the Okay button is clicked.

Thus we see that there is some editing done as a field is entered, but some which must be delayed until all fields are entered.

**Verifying year and month text fields**

The main feature of an InputVerifier is a verify method, which accepts a JComponent as a parameter. A JComponent is the parent of the parent of JTextField. The verify method will return true if it finds no errors with the data provided in the JComponent, false if there is a problem with the data provided.

Here is an InputVerifier for the year. Notice that I begin the name of an InputVerifier with the word verifier.

```java
InputVerifier verifierYear = new InputVerifier() {
    public boolean verify(JComponent input) {
        int screenYear;
        // because an InputVerifier can be attached to many types of Components, we must
        // cast to the appropriate type of data to extract its contents
        String text = ((JTextField) input).getText();

        if (text.length() == 0) {
            // There was no data provided
            JOptionPane.showMessageDialog(jf, "Need to supply a year", "Missing data", JOptionPane.ERROR_MESSAGE);
            return false;
        } // length = 0

        try {
            // data was provided. Is it an integer?
            screenYear = (new Integer(text).intValue());
            // the data contains an integer. If not, the previous statement threw an exception.
            // is the integer reasonable? reasonable is defined as between 1000 and
            // 9999 inclusive
            if ((screenYear < 1000) || (screenYear > 9999)) {
                JOptionPane.showMessageDialog(jf, "Year should be four digits in length.", "Bad year", JOptionPane.ERROR_MESSAGE);
                return false;
            }
        } catch (Exception e) {
            JOptionPane.showMessageDialog(jf, "Year contains non-numeric characters.", "Bad year", JOptionPane.ERROR_MESSAGE);
            return false;
        }
    }
};
```


What could go wrong while entering a year?
   - You could neglect to enter anything.
   - You could enter non-numeric data.
   - You could enter a numeric value which is too small or too large.

The InputVerifier checks each, in the order listed.

Note the use of (new Integer(text).intValue()) to convert a String into an int.

The Integer class is a wrapper class for int; it takes the primitive datatype and converts it to a class. Of course there is a method (intValue) to extract the underlying int. We attempt to make the conversion, catching an exception should it be unsuccessful.

We have already used the Long class in our discussion of autoboxing and this is no different.

Once we have created the InputVerifier, we must associate it with (attach it to) the appropriate widget.

textYear.setInputVerifier(verifierYear);

InputVerifiers are often associated with only one widget. An obvious exception to such a generalization would occur on a data entry screen which requires the entry of several dates. All the year fields could share the same InputVerifier.

Test your data entry screen.

Note that you cannot use the cancel button to exit the frame unless a valid year has been provided. If you select the X in the top-right corner, you are able to leave the data entry screen with no problem. But if you select the cancel button, an error message appears.

Why does the error message happen? More importantly, how can you correct it? Hint, consider this statement.

buttonCancel.setVerifyInputWhenFocusTarget(false);

How is that for the name of a method? What does that method do?

The InputVerifier for the month is similar and is left as an exercise.
Verifying the day text field

The InputVerifier for the day is a little more complicated since the upper limit for the day depends on the month and the year.

```java
InputVerifier verifierDay = new InputVerifier() {
    public boolean verify(JComponent input) {
        int screenDay;
        String text = ((JTextField) input).getText();

        if (text.length() == 0) {
            // no day has been provided
            JOptionPane.showMessageDialog(jf, "Need to supply a day", "Missing data",
                JOptionPane.ERROR_MESSAGE);
            return false;
        } // length = 0
        try {
            screenDay = (new Integer(text).intValue());
            if (screenDay < 1) {
                // regardless of the month, 0 or a negative number is a bad day
                JOptionPane.showMessageDialog(jf, "Day number is too small.", "Bad day",
                    JOptionPane.ERROR_MESSAGE);
                return false;
            } // if the year, month, and day have been specified check maximum value for day
            int screenYear = 0;
            int screenMonth = 0;
            try {
                // do we have a year?
                screenYear = (new Integer(textYear.getText())).intValue();
                // do we have a month?
                screenMonth = (new Integer(textMonth.getText())).intValue();
            } catch (Exception e) {
                // some part of the date is missing. do no more day editing
                return false;
            }
            // we have all the parts of a date. Is the day no larger than allowed?
            // could use an array instead of if statements
            int maxDays = 31;
            if (screenMonth == 2)
                if (isLeapYear(screenYear))
                    // isLeapYear follows this method
                    maxDays = 29;
                else
                    maxDays = 28;
            if ((screenMonth == 9) || (screenMonth == 4) || (screenMonth == 6) ||
                (screenMonth == 11))
                maxDays = 30;
            // now we know the maximum number of days in the month
            if (screenDay > maxDays) {
```
The logic verifying the day is similar to that verifying the year and month (Is there anything provided? If so, is it numeric? If so, is it appropriate?). However the logic for checking the upper limit is a little more complicated since it depends on the year and the month.

First, the method checks that a year and month have been provided.

If so, it determines the correct number of days in the month and compares the value provided to that number.

Note the isLeapYear method.

/**
 * calculate if a year is a leap year.
 * @param year The year which we are checking
 * @return true if so, false if not
 */
public static boolean isLeapYear(int year) {
    return (new GregorianCalendar()).isLeapYear(year);
}

Rather than writing yet another leap year routine, embodying the logic necessary to decide when a year is a leap year (see, for example, http://en.wikipedia.org/wiki/Leap_year), I have chosen to use the GregorianCalendar class and its isLeapYear method. Don’t forget the import statement.

import java.util.GregorianCalendar;

Of course, we attach the InputIdentifier to its widget.

textDay.setInputVerifier(verifierDay);

And yes, (as you see when you compile the code) you will need to declare textYear, textMonth, and textDay variables as final.
Verification behind the Okay button

Display the data entry frame and then click Okay. Since there is an InputVerifier attached to the year text field, an error message appears stating that a year must be provided. Thus it is impossible to proceed without providing a year, a month, and a day of the month.

But what happens if a text field is required, but cannot be verified. Some would omit an InputVerifier then, but you could create an InputVerifier which returns true when some value is provided and false when none is provided.

What happens if a text field is required and needs to be validated against some other text field but neither text field has an InputVerifier? For example, one of the exercises asks you to create a data entry screen for an Address. All parts of the address are optional, so none have an InputVerifier, but if there is a postal code it can be validated against the province.

That validation cannot be done using an InputVerifier. It must be done using an ActionListener.

If you want to explore verifiers further, and you are interested in Canadian postal codes, look at information on forward sortation areas at [http://en.wikipedia.org/wiki/Postal_codes_in_Canada](http://en.wikipedia.org/wiki/Postal_codes_in_Canada) or [http://www.canadapost.ca/tools/pg/manual/PGaddress-e.asp](http://www.canadapost.ca/tools/pg/manual/PGaddress-e.asp).

Improvements

What can we do to improve our data entry screen?

Remove intrusive dialogs

The first improvement is to remove the intrusive dialogs and replace them with a message, implemented as a JLabel, which becomes visible when appropriate.

Creating the message

In your method, after you have created your buttons, insert the following statements.

```java
final JLabel statusMessage = new JLabel("Error messages go here.");
RelativeConstraints smConstraints = new RelativeConstraints();
bf.setLeftMargin(10);
Binding leftEdge = bf.leftEdge();
Binding topEdge = bf.topAlign(buttonCancel);
smConstraints.addBindings(leftEdge, topEdge);
jf.add(statusMessage, smConstraints);
```

This code must go after you have created your buttons since we are using the top of the buttons to position the JTextField.
The word final in the first statement is a promise that we may be changing the contents of the JLabel but we will not be changing the address of the area of memory to which it refers.

Why is the third line necessary? Recall that earlier one of the steps in aligning the left edges of the JTextFields was to set the left margin 200 pixels in from the left edge of the JFrame. Here we need the original margin.

Run the method and you’ll see the message appearing. Normally we want it to be hidden; add one more statement.

```java
statusMessage.setVisible(false);
```

Run the method again.

**Using the message**

Instead of displaying the intrusive dialogs, we will simply place the appropriate message in the JLabel and make it visible. When the InputVerifier tells us that the data is correct, we hide the JLabel once again.

Thus, we replace the statement

```java
JOptionPane.showMessageDialog(jf, "Need to supply a year",
"Missing data", JOptionPane.ERROR_MESSAGE);
```

in the InputVerifier for the year with these statements.

```java
statusMessage.setText("Need to supply a year");
statusMessage.setVisible(true);
```

We do the same for the other two messages in the InputVerifier for the year.

In addition, immediately before the InputVerifier returns true, we insert the statement

```java
statusMessage.setVisible(false);
```

We would not want a misleading error message to remain.

In a similar way, we replace the JOptionPanes in the other InputVerifiers with the status message.

**Disable the OK button until needed**

The second improvement is to ensure the OK button is disabled until all the necessary fields have been provided. It is a peeve of mine when a program allows you to attempt an action but then displays an error message. Why not simply disallow the action until it is meaningful?

To disable the OK button until all fields are provided correctly requires a little work. Jason S, one of my students, provided the idea for an interesting solution.
Create a class which contains a collection of boolean variables, one for each of the fields on the data entry screen. This class initializes them all with to false; that is, the fields are not acceptable with their current values. Set the booleans for the optional fields to be true.

As a required field has its value provided correctly, its corresponding boolean has its value changed to true.

As an optional field is provided, edit it if possible and set its boolean as appropriate. For example, assume an optional field may have any value except 1, 2, or 3. If you attempt to give it any of the disallowed values, its corresponding boolean flag will become false.

When all the booleans are true, the OK button is enabled.

This works well, except when the last data entry field is a required field and you have used the statement

cancelButton.setVerifyInputWhenFocusTarget(false);

as explained in the next chapter.

This statement causes the InputVerifier for the last field not to be executed, and thus the OK button is never enabled. A solution is to ensure that the last field on the data entry screen is not required.

The class used for this is shown below.

/**
 * A class, based on ideas from Jason S, that contains many boolean flags, used in editing data
 * on a data entry screen.
 */
class VerifierTracker {
    private boolean[] flags;

    /**
     * Constructor for VerifierTracker.
     * @param n the number of flags to create. All start as false.
     */
    public VerifierTracker(int n) {
        flags = new boolean[n];
        for (int i = 0; i < flags.length; i++)
            flags[i] = false;
    }

    /**
     * set the specified flag to the value specified.
     * @param i the flag number
     * @param b its new boolean value
     */

public void setFlag(int i, boolean b) {
    flags[i] = b;
}

/**
 * return the value of flag i.
 * @return flag[i]
 */
public boolean getFlag(int i) {
    return flags[i];
}

/**
 * Set all flags to the given value.
 * @param b the value for all flags
 */
public void setAllFlags(boolean b) {
    for (int i = 0; i < flags.length; i++)
        flags[i] = b;
}

/**
 * Examine the flags and see if all are set.
 * @return true if all the flags in the collection are true, false otherwise.
 */
public boolean allSet() {
    for (int i = 0; i < flags.length; i++)
        if (!flags[i])
            return false;
    return true;
}

Where do you place the class?

Since it can be used in many projects, it should be in a project of its own. Then you can access it from within any BlueJ project by choosing Tools, Preferences, Libraries, and adding the project which contains VerifierTracker.

How do you use the class?

First, create an instance of its datatype within your data entry screen. For the current data entry screen which has three fields, we use

VerifierTracker vee = new VerifierTracker(3);

Second, create constants to refer to each flag. That way you can use the constants rather than the actual numbers.

final int YEARFLAG = 0;
final int MONTHFLAG = 1;
final int DAYFLAG = 2;

Then within each InputVerifier, ensure the following statements appear before the return statement.

vee.setFlag(XXX, result);
okayButton.setEnabled(vee.allSet());

where XXX is YEARFLAG, MONTHFLAG, or DAYFLAG, as appropriate.

Remember to set all the flags back to false when the OK Button ActionListener completes its work.

Data entry for other classes

Using the ideas above, you can create simple data entry screens for the other classes which require them, Student, Professor, Course, Meeting, Mark, College, and Address. Most of these will be left as exercises for the reader.

Testing a GUI

We have used JUnit extensively to test the constructors, getters, setters, and toString methods we have created. Is there a similar tool to automate the testing of your GUI?

We will not go into testing GUI's in this chapter, but you will find there are such tools if you search the web for terms like GUI test Java.

Summary

In the chapter, we have covered the basics of creating data entry screens, and have seen how to create one simple data entry screen.

In the next chapter, we will see how to create more-complicated data entry screens. Section has some interesting aspects, so we’ll examine its data entry screen next.
Exercises

1. Complete the InputVerifier for the month text field.

2. Create a data entry screen for the Student class, collecting personal information, but not registration information.

3. Create a data entry screen for the Professor class, collecting personal information but nothing on the sections and courses the professor teaches.

4. Create a data entry screen for the Course class, but not collecting any information about the sections of the courses which are offered.

5. Create a data entry screen for the Meeting class, but not associating it with a section just yet.

6. Create a data entry screen for the Mark class, but not associating it with a student and course just yet.

7. Create a data entry screen for the Address class.

8. Create a data entry screen for the College class. It will allow you to enter its address and website, but nothing about employees, students, or courses.

9. Explore the subject of “look and feel”. What options do you have with the Swing toolkit? What are the differences between the options?

10. What are some of the other layout managers you can use? When would you use each? Could you use any of them to create the data entry screen we have seen in the chapter?

11. (Hard and maybe pointless) Use the GridBagLayout manager to recreate the data entry screen we have created in the chapter. Which layout manager do you prefer?

12. In previous chapters we discussed modelling a bank account. Create a data entry screen for entering the personal information for the holder of the account.

13. In previous chapters we discussed modelling a bank account. Create a data entry screen for entering a transaction, omitting the choice of the account to which the transaction applies. In the next chapter we will add the account.

14. In previous chapters we discussed modelling the sightings a birder makes. Create a data entry screen for entering a species of bird, including both the common and scientific name.
If you are serious about birding, you should know about the American Ornithologists’ Union and their website at http://www.aou.org/. This site makes available a list of over 2000 bird species seen in North America. It may be possible to use their list to make the data entry screen in this question redundant; you could perhaps link to the AOU site and select from a list.

You should also know about your local birding groups. These include, for example, the British Columbia Field Ornithologists (http://www.bcfo.ca/) and Bird Studies Canada (http://www.bsc-eoc.org).

15. In previous chapters we discussed modelling the sightings a birder makes. Create a data entry screen for entering a sighting, omitting the choice of the species which was sighted. In the next chapter we will add the species.

16. Find a GUI builder and use it to create a data entry screen.

17. Explore the use of the JTextArea and JScrollPane widgets.

18. Explore the use of the JPasswordField.

19. Explore the use of HTML within JButton, JToolTip, and JLabels.
Learning objectives

By the end of the chapter you will be able to:

- Use Swing components to create complicated data entry screens
- Describe the role of a layout manager in creating data entry screens
- Use a layout manager to create complicated data entry screens
- Use ActionEvents and ActionListeners
- Use InputVerifiers

Introduction

The data entry screen we created in the previous chapter is relatively simple. It contains some labels, some text fields, and some buttons. But more complicated data entry screens are very common.

They include panels to separate portions of the frame.

They include checkboxes and radio buttons.

They include lists of items.

Let’s see how we create these data entry screens.

Building the Section data entry screen

The data entry screen for a Section naturally divides itself into several portions: the course with which the section is associated, the meeting times for the section, and a collection of buttons.

To build the screen, additional Swing components are necessary, including the JRadioButton, JList, JScrollPane, JPanel, and JCheckBox components.

The data entry screen we will create follows.
You can see the three JPanels in the image above.

- The first JPanel is the portion of the screen containing course list and the section number. It is clearly separated from the rest of the screen by a border.

- The second JPanel contains the radio buttons, labels, text fields, and checkboxes associated with meeting times. It too has a border.

- The third JPanel contains the buttons. I have omitted the border on the panel because it seems to be superfluous. The bottom border of the second panel serves as a top border and the left, right, and bottom borders of the frame itself serve as the left, right, and bottom borders.

Let’s examine parts of the method necessary to build the screen, a few statements at a time.

But first, let's create a method to display the frame, so that we can test the code we are about to develop. Of course, the method is within the Section class.

```java
/**
 * show the Frame.
 */
public static void showFrame() {
    Runnable runner = new Runnable() {
        public void run() {
```
JFrame box = buildFrame();
box.setLocation(100, 100);
box.setPreferredSize(600, 500);
box.pack();
box.setVisible(true);
}
};
EventQueue.invokeLater(runner);
}
Yes, we are using exactly the same technique we used to display the MyDate data entry screen.

Now we are ready to begin creating the data entry frame itself.

Create the frame
The simplest version of the method is:

public static JFrame buildFrame() {
    // create the JFrame.
    final JFrame sectionInput = new JFrame("Enter Section Information");
    sectionInput.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);
}
We have created the frame (In the previous chapter, I used the nondescript variable name jf. Here I use the more informative sectionInput.) and specified that it will hide itself when we click the X icon in the top-left corner of the frame. These statements are common to most data entry screens you create.

Test it.
To the simple version of the method above add the following statements.

Create a column of panels

// use the BorderLayout to create a column of JPanels.
// Actually BorderLayout gives five sections of the frame but we will not use the EAST and WEST.
sectionInput.setLayout(new BorderLayout());

Recall that the BorderLayout divides the form into five regions, a central region and four surrounding regions. The comment serves as a reminder.

Create the three panels

JPanel topPanel = new JPanel();
JPanel middlePanel = new JPanel();
JPanel bottomPanel = new JPanel();
// and add the panels in the correct order, from top to bottom
sectionInput.add(topPanel, BorderLayout.NORTH);
sectionInput.add(middlePanel, BorderLayout.CENTER);
sectionInput.add(bottomPanel, BorderLayout.SOUTH);

The BorderLayout layout manager (imported from java.awt.BorderLayout. Don’t forget the import statement!) divides the frame into five sections. We are using three sections and our panels will expand to fill those section. The EAST and WEST sections will still exist, but will be empty (and thus disappear).

While I like the RelativeLayout layout manager, there are times where it is the best tool and times where other layout managers are the best tools. I am trying to show you other layout managers whenever their use is appropriate. As we build the frame, we will also use the BoxLayout and the FlowLayout managers.

Notice that we have created the three JPanels and placed them within the frame, but we have not yet specified what the JPanels contain. We’ll do that now, starting with the top JPanel.

**Populate the top panel**

A Box is a container into which you can place Objects either vertically or horizontally.

More correctly, a Box is a container which uses the BoxLayout layout manager, a manager which stacks widgets on top of each other, or beside each other, depending on the orientation of the Box. When you have columns of widgets, it may be easier to use a VerticalBox to contain and position them than to use some other layout manager.

**Could you use the BoxLayout manager to position the three JPanels on the screen?**

// the top panel contains two Boxes, one containing a JLabel and a JList to select the course,
// and one containing a JLabel and a JTextField to enter a section number

Box courseSelector = Box.createVerticalBox();
Box sectionSelector = Box.createHorizontalBox();

The first Box will have a caption above the list; it is a vertical box so the widgets will be placed vertically.

The second Box will have a caption beside a text field. They are horizontal boxes, so the widgets will be place horizontally.

First we create the box for the list of courses. Note that you will need to create Course objects for all of these courses.

// courseSelector Box
JLabel courseLabel = new JLabel("Course");
courseSelector.add(courseLabel);

// the course JList, a list of items from which the user can select. First prepare
// some default values, stored in an array. These values should come from a database,
// but we don’t know how to use databases yet so we will just provide some values here
String[] data = {"COSC 109", "COSC 111", "MATH 139", "NTEN 112", "BUAD 123", "PSYC 123",
                     "GEOG 111", "POLI 100", "BUAD 123", "BUAD 128", "BUAD 111"};
// create the list box and place the contents of the array in the JList
JList<String> coursesList = new JList(data);

// allow user to make only one selection
coursesList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

// embed the JList in a JScrollPane to allow for scrolling
JScrollPane courses = new JScrollPane(coursesList);

// add the JScrollPane to the panel
courseSelector.add(courses);

So the courseSelector Box contains a JList (which scrolls, and allows only one choice to be made). If you don’t have a database of courses, which we would normally use to populate the JList, you simply create an array containing some courses, and use that array to initialize the JList, using enough courses that we will see the scroll bars appear. If you have a collection of courses within the College singleton, use them.

We have seen arrays before, and will see them again.

Finally, we create we create the box for the section.

    // sectionSelector Box
    JLabel sectionLabel = new JLabel("Section ");
    sectionSelector.add(sectionLabel);

    // the section JTextField
    final JTextField sectionText = new JTextField(4);
    sectionSelector.add(sectionText);

Again, the JLabel’s caption contains some blanks for spacing,

**Place the boxes in the panel**

    // and add the three Boxes to the top JPanel
    top.add(courseSelector);
    top.add(sectionSelector);

And we place the Boxes in the JPanel. What layout manager is top using?
Decorate the panel

We would like to use borders separate the panels from the rest of the frame. So we need to create the borders.

```
// create borders for the JPanels
Border raisedBevel = BorderFactory.createRaisedBevelBorder();
Border loweredBevel = BorderFactory.createLoweredBevelBorder();
Border compound = BorderFactory.createCompoundBorder(raisedBevel, loweredBevel);
```

Border is an interface describing an object capable of rendering a border around the edges of a Swing component. Its use requires an import statement.

```
import javax.swing.border.*;
```

A BorderFactory is a class which creates objects implementing the Border interface. Its use requires an import statement.

```
import javax.swing.BorderFactory;
```

Explore BorderFactory to see the kinds of borders it can create. If you find a more appealing border, use it in place of what I have created.

Continuing with our method,

```
// decorate the JPanel with a border
top.setBorder(compound);
```

Now that we have a Border available, we place it around the JPanel and the appearance of the top JPanel is complete.

Could a TitledBorder be useful here?

Create the middle panel

This panel is more complex than anything we have seen. Because of its complexity, we will use RelativeLayout to design it. I suppose we could have built it using 11 vertical boxes, or six horizontal ones.

```
middle.setLayout(new RelativeLayout());
```
Labels

Note that there is a row of labels. For each label, we follow our usual practice; create a widget and its bindings, add the bindings to a collection, and then add the widget and its collection of bindings to the container (previously a frame but here a panel).

Here, the top edge of all the labels is the same, so we create a constraint for that, and then use it for each label.

```java
middlePanel.setLayout(new RelativeLayout());

BindingFactory bf = new BindingFactory();

JLabel dayLabel = new JLabel("Day of the week");
Binding leftEdge = new Binding(Edge.LEFT, 10, Direction.RIGHT, Edge.LEFT, middlePanel);
Binding topEdge = new Binding(Edge.TOP, 10, Direction.BELOW, Edge.TOP, middlePanel);
RelativeConstraints theBindings = new RelativeConstraints();
theBindings.addBindings(leftEdge, topEdge); // does it matter the order you add Bindings?
middlePanel.add(dayLabel, theBindings);

JLabel startLabel = new JLabel("Start time");
topEdge = bf.topAlign(dayLabel);
leftEdge = new Binding(Edge.LEFT, 275, Direction.RIGHT, Edge.LEFT, dayLabel);
theBindings = new RelativeConstraints();
theBindings.addBindings(topEdge, leftEdge);
middlePanel.add(startLabel, theBindings);

JLabel endLabel = new JLabel("End time");
topEdge = bf.topAlign(dayLabel);
leftEdge = new Binding(Edge.LEFT, 75, Direction.RIGHT, Edge.LEFT, startLabel);
theBindings = new RelativeConstraints();
theBindings.addBindings(topEdge, leftEdge);
middlePanel.add(endLabel, theBindings);

JLabel roomLabel = new JLabel("Room");
topEdge = bf.topAlign(dayLabel);
leftEdge = new Binding(Edge.LEFT, 75, Direction.RIGHT, Edge.LEFT, endLabel);
theBindings = new RelativeConstraints();
theBindings.addBindings(topEdge, leftEdge);
middlePanel.add(roomLabel, theBindings);
```

Notice something new in these statements. Rather than having separate collections for the bindings of each widget, we use only one collection, named `theBindings`. There is no way to remove bindings from a collection, so we simply allocate new memory for it each time we wish to erase the previous bindings. Garbage collection takes care of the previous, now unused, memory.
The meetings

The labels are followed by several similar rows for the different meetings of the section. The actual number of rows which appear is available from the College class.

For each meeting of the section (that is, for each of the rows), we need seven JRadioButtons (corresponding to the days of the week), and three JTextFields (the start and end times of the meeting, and the room in which the meeting will take place.) We’ll see how to do the checkboxes in a moment.

So we have several rows of widgets, each row of which appears identical. Intuition (another name for learning from past mistakes) tells me that a good way to deal with identicalness is to use a data structure of some kind, perhaps an array or a Vector or a List, with each element of the collection corresponding to one row on the data entry screen. Continuing with intuition, we realize there are seven radio buttons. They could make a collection too.

Since the data structure or collection is quite simple and there is no real processing of the elements in the collection, we can use an array. To do that, we should examine arrays more fully before we continue.

Arrays

We have used collections (and arrays) before. Generally we needed to go through all the elements in the collection, starting at the first and ending at the last. But what if we need to access the elements of the collection randomly? That is, we want the fourth element, then the eighth, then the first.

It would be very inefficient (slow) to have to process all preceding elements in the collection to retrieve a specific one. Fortunately, there is a very traditional data structure which supports random access. This is the array.

A Student object contains a collection of Mark objects but that collection is not an array. A Course object contains a number of Section objects but that collection is not an array. A Section object contains a number of student identifiers but that collection is not an array.

Those collections are not arrays because arrays are of fixed size.

An array has two parts. The first is the collection of data items. The second is the subscript or index, a number used to indicate which element of the collection we wish.

Suppose we have a collection of Student objects, which we wish to store in an array because we need to access the Student objects randomly. Consider the statement

Student[] theStudents = new Student[50];
This array, whose name is theStudents, contains space for 50 Student objects. The objects are numbered zero through 49. The index always starts at zero. In memory, these 50 objects are stored next to each other.

When we wish to retrieve the last Student object we retrieve theStudents[49]. If we wish to retrieve the first Student object, we retrieve theStudents[0].

When we wish to retrieve all the Student objects, we use a for statement.

```java
for (int i = 0; i < 50; i++) {
    // do something with theStudents[i]
}
```

But why stop at an array of Student objects? We can have an array of anything, primitive datatypes like int or char (Did you realize that you can think of a String as an array of char? You probably did since we inspected a String and saw its array of char.), or an array of arrays.

An array of arrays? Yes, each element of one array is itself a second array. You have probably seen such a structure often.

The board on which you play chess or checkers is a two-dimensional array. The board on which you play a game of Sudoku is a two-dimensional array.

The rows are elements of an array; but each row contains an array, each element of which is a column.

Each sheet of a workbook (a spreadsheet) is a two-dimensional array, whose elements are much more complicated than those of a chessboard.

That is, these are arrays in which you can think of each array element consisting of an array.

Were we to create a variable to represent a chessboard, we might use something like the following statement.

```java
int[][] chessboard = new int[8][8];
```
The model of a chessboard and its pieces could represent an empty cell with a zero, a cell containing a white piece with a negative number (different numbers correspond to different pieces), and a cell containing a black piece with a positive number.

If you stretch your mind you can imagine ragged arrays, in which each row may not have the same number of cells. Imagine the histogram turned on its side.

So, now that we know a little more about arrays, let’s continue building the data entry screen.

Create the middle panel, continued

Radio buttons

When I look at the screen we are building, experience helps me see that the radio buttons form a two-dimensional array (several rows, seven columns), while the start time, end time, and room each form one-dimensional arrays (several rows each). Most importantly, the arrays are associated. That is, an element in one row of one of the arrays is associated with the corresponding element in the other arrays.

When you look at the data in, for example, the third elements of these arrays, you will find the day of the week on which the third meeting takes place, the time it starts, the time it ends, and the room in which it occurs.

ButtonGroup

Note that radio buttons occur in groups. When you select one button in a group, all the others are deselected. That’s the traditional way you select which radio station to listen to on your vehicle radio. Push one button, you hear one station; push another button and you hear another station.

When you have more than one group of radio buttons on a screen, selecting and deselecting in one group does not influence your choices in the other groups. We isolate the buttons by placing the buttons for each of the possible meetings in different ButtonGroups.

    ButtonGroup aGroup = null;

This creates a reference to a ButtonGroup object. We will use the ButtonGroup constructor several times to actually create the many ButtonGroups we will actually create.
We will see how to add JButtons to ButtonGroups in a moment. Note that when first created, all the buttons in a button group will be deselected.

Note that there are some applications in which choosing a button in one group may affect the contents of another group.

For example, a washing machine I recently purchased lets you select the type of material you are washing. Depending on your choice of material, you may not be able to use hot water. The “hot water” choice in the temperature group is disabled.

An array of radio buttons

Since we have arrays, we need to have values which tell us the bounds of the loops which we will use to process the elements of the arrays.

```java
College c = College.getInstance();
final int MAX_MEETINGS = c.getMaxMeetingsPerWeek();
final int DAYS_IN_WEEK = 7;
```

And then we create the arrays.

```java
final JRadioButton[][] aRadioButton = new JRadioButton[MAX_MEETINGS][DAYS_IN_WEEK];
```

Note that we have created the possibility of the array; the individual array elements do not yet have value.

An array of text fields

Similarly, we create arrays of text fields for the start and end times, and the room numbers.

```java
final JTextField[] startText = new JTextField[MAX_MEETINGS];
final JTextField[] endText = new JTextField[MAX_MEETINGS];
final JTextField[] roomText = new JTextField[MAX_MEETINGS];
```

Checkboxes

In thinking about the screen, I realized it could be difficult to tell which of the five (oops, MAX_MEETINGS) meeting possibilities are actually being used and it would be inconvenient to change many values when a meeting is removed.

So I decided to place a column of checkboxes down the right side of the screen. Whenever the data entry person selects a day, or a time, or a room, the appropriate checkbox will be checked by the method. When the data entry person unchecks a checkbox, the appropriate time and room will be erased from the screen.

When the data entry person clicks the OK button, we will process only those rows which have the checkbox checked. This requires another array.
final JCheckBox[] hasMeeting = new JCheckBox[MAX_MEETINGS];

You may have noticed that all these arrays are declared to be final. This is so that the ActionListeners we will need are able to access them properly. We saw this use of final in the previous chapter.

**Placing the arrays in the frame**

At last we are able to begin creating the individual elements of the arrays and placing them on the screen.

```
topEdge = bf.below(dayLabel);
// for all the meetings
for (int i = 0; i < MAX_MEETINGS; i++) {
  // the radio buttons
  aGroup = new ButtonGroup();
  leftEdge = bf.leftEdge();
  for (int j = 0; j < DAYS_IN_WEEK; j++) {
    String text = "" + ("SMTWRFA").charAt(j);
    aRadioButton[i][j] = new JRadioButton(text);
    aGroup.add(aRadioButton[i][j]);
    theBindings = new RelativeConstraints();
    theBindings.addBindings(leftEdge, topEdge);
    middlePanel.add(aRadioButton[i][j], theBindings);
    leftEdge = bf.rightOf(aRadioButton[i][j]);
  }

  // and the text fields
  startText[i] = new JTextField(4);
  theBindings = new RelativeConstraints();
  leftEdge = bf.leftAlignedWith(startLabel);
  theBindings.addBindings(leftEdge, topEdge);
  middlePanel.add(startText[i], theBindings);

  endText[i] = new JTextField(4);
  theBindings = new RelativeConstraints();
  leftEdge = bf.leftAlignedWith(endLabel);
  theBindings.addBindings(leftEdge, topEdge);
  middlePanel.add(endText[i], theBindings);

  roomText[i] = new JTextField(4);
  theBindings = new RelativeConstraints();
  leftEdge = bf.leftAlignedWith(roomLabel);
  theBindings.addBindings(leftEdge, topEdge);
  middlePanel.add(roomText[i], theBindings);

  // and the checkbox
  hasMeeting[i] = new JCheckBox();
  theBindings = new RelativeConstraints();
  leftEdge = bf.rightOf(roomText[i]);
```
The core of this code is a pair of nested for loops.

```java
for (int i = 0; i < MAX_MEETINGS; i++) {
    for (int j = 0; j < DAYS_IN_WEEK; j++) {
        ...
    }
}
```

The first loop, the outer one deals with the rows to be displayed. The inner loop deals with the radio buttons for the seven days of the week.

The first ellipsis (the ...) indicates there are some statements that must be done to make the radio buttons appear.

The second ellipsis indicates there are some statements that must be done after the radio buttons are created, specifically related to the textboxes for the start and end times, and the room, and the checkboxes.

We definitely need to create the individual array elements, for all the arrays and we need to place them in the correct locations. How do we do that? In particular, how do we deal with the bindings?

We use a binding to describe the top of all widgets in a row. The first meeting row is below the row labels, all of which are aligned along their top edges.

```java
topEdge = bf.below(dayLabel);
```

Subsequent rows are below the previous row.

```java
topEdge = bf.below(aRadioButton[i][0]);
```

What do we do for the individual widgets in a row?

The radio buttons begin at the left edge of the panel, taking into account the default margin.

```java
leftEdge = bf.leftEdge();
```

Subsequent radio buttons are to the right of the previous radio button. This keeps the columns of radio buttons aligned on their left edge as all radio buttons are the same width.

```java
leftEdge = bf.rightOf(aRadioButton[i][j]);
```
The text fields line up with their respective labels. For example,

leftEdge = bf.leftAlignedWith(startLabel);

Notice that every widget has a left and top binding. The variable names leftEdge and topEdge are used over and over again.

The label on the radio button is determined for the English-speaking world.

The checkboxes are placed to the right of the last text field.

leftEdge = bf.rightOf(roomText[i]);

**InputVerifiers**

But wait a moment! We realize that we will need InputVerifiers for the text fields to check that the data we enter is reasonable.

We need one for the section (checking for the correct format. Perhaps we can use SectionException to help us.)

To indicate which item in the list is chosen, use the first element as the default choice.

coursesList.setSelectedIndex(0);

Here are the InputVerifiers for the times. The InputVerifier for the rooms follows.

```java
// InputVerifier for the times
InputVerifier verifierTime = new InputVerifier() {
    public boolean verify(JComponent input) {
        int screenTime;
        String text = ((JTextField) input).getText();
        if (text.length() == 0) {
            // no time specified
            JOptionPane.showMessageDialog(sectionInput, "Need to supply some time",
                    "Missing data", JOptionPane.ERROR_MESSAGE);
            return false;
        } // length = 0
        try {
            // is there a numeric time
            screenTime = (new Integer(text)).intValue();
            // and is it reasonable?
            int screenHour = screenTime/100;
            int screenMinutes = screenTime % 100;
            if ((screenTime < 0000) ||
                (screenTime > 2359) ||
```
The InputVerifier for the rooms, is incomplete, as we do not have any way of storing and retrieving a list of allowable rooms. The InputVerifier simply returns true, the room is acceptable, whenever we supply a room. This InputVerifier is simply a stub, a small piece of code which will be completed at a later time.

```
  // InputVerifier for the room
  InputVerifier verifierRoom = new InputVerifier() {
    public boolean verify(JComponent input) {
      int screenRoom;
      String text = ((JTextField) input).getText();
      if (text.length() != 0) {
        // should check if the room is in a database, but we don't have one yet
        return true;
      }
      return false;
    } // end verify
  }; // end verifierRoom
```

Of course each InputVerifier must be attached to the appropriate widget.

```
for (int i = 0; i < MAX_MEETINGS; i++) {
  startText[i].setInputVerifier(verifierTime);
  endText[i].setInputVerifier(verifierTime);
  roomText[i].setInputVerifier(verifierRoom);
}
```

What about an InputVerifier for the list of courses? We need to know that a course has been chosen.

An InputVerifier is not what we need. Instead, we need a MouseListener. Explore that topic on your own.
ActionListeners for radio buttons

Recall that the check boxes work in conjunction with the radio buttons and text fields. If we click a radio button or enter a start or end time, or a room, we would like the check box for that row to become checked.

Similarly, if a check box is unchecked, we would like to have the radio buttons become unchecked and the start time, end time, and room number to vanish.

How can we link these components together?

We have seen ActionListeners in the previous chapter and we need them here as well. We want to attach the same ActionListener to several widgets, but we will need to know which widget creates the ActionEvent.

But we need an additional feature of ActionListeners, the ability to identify a specific String with each widget. When an ActionListener is executed, we can check for a specific String with the getActionCommand method.

Instead of just creating a radio button

```java
aradioButton[i][j] = new JRadioButton(text);
```

we need an additional statement

```java
aradioButton[i][j].setActionCommand( );
```

with some unique String provided as a parameter. The String needs to identify the row in which the radio button is located, so we can use

```java
aradioButton[i][j].setActionCommand(Integer.toString(i));
```

Here is the appropriate ActionListener, one which applies to all the radio buttons. Don’t forget to add it to each radio button.

```java
// ActionListener for the radio buttons
ActionListener actionListenerRadioButton = new ActionListener () {
    public void actionPerformed(ActionEvent e) {
        try {
            int row = new Integer(e.getActionCommand()).intValue();
            hasMeeting[row].setSelected(true);
        } catch (Exception x) {
        }
    }
};
```
ActionListeners for checkboxes

We need ActionListeners for the checkboxes. When a checkbox is deselected, we will erase all the information associated with that meeting. For the technique to work, we need the ActionCommand set so we can tell which checkbox is checked.

hasMeeting[i].setActionCommand(Integer.toString(i));

Then we can associate the ActionListener with all the check boxes.

// ActionListener for the checkboxes
ActionListener listenerCheck = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // which button
        AbstractButton abstractButton = (AbstractButton)e.getSource();
        // was it checked or unchecked?
        boolean selected = abstractButton.getModel().isSelected();
        // if unchecked
        if (!selected) {
            try {
                int row = new Integer(e.getActionCommand()).intValue();
                startText[row].setText("");
                endText[row].setText("");
                roomText[row].setText("");
            } catch (Exception x) {
            }
        }
    }
};

But we have a small problem. This ActionListener erases the values we provided for the start time, end time, and room number, but leaves the radio buttons checked.

We can clear all the buttons in a group using the clearSelection method but to do that we need to know the ButtonGroup with which we are dealing.

We need an array of ButtonGroups, which we don’t have yet.

Make it so.

Decorate the middle panel

Place a border around the middle JPanel and we are done with it.

    middle.setBorder(compound);
The bottom panel

Let’s continue and build the bottom JPanel, the one containing the buttons.

Which layout manager should we select for the bottom panel?

The panel contains only two JButtons so RelativeLayout would be overkill for such a simple situation. Instead, we will use FlowLayout.

With FlowLayout, components are placed side by side, at the specified end of the panel, in the order in which they are added to the JPanel. That is, the okay button is added first, so will be moved to the left when we add the cancel button.

```
// build the bottom JPanel. It contains JButtons
// use FlowLayout placing components at the right.
// the standard order (left to right) is
// OK and then cancel
bottomPanel.setLayout(new FlowLayout(FlowLayout.RIGHT));

JButton okayButton = new JButton("OK");
okayButton.setMnemonic(KeyEvent.VK_K);
bottom.add(okayButton);

JButton cancelButton = new JButton("Cancel");
cancelButton.setMnemonic(KeyEvent.VK_C);
bottom.add(cancelButton);
```

And we make the cancel button the default button.

```
sectionInput.getRootPane().setDefaultButton(cancelButton);
```

Button ActionListeners

Of course, buttons need ActionListeners too. To check that the buttons respond correctly, we could use these ActionListeners.

```
ActionListener listenerOkay = new ActionListener () {
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(sectionInput, "Clicked okay.", "Okay", JOptionPane.INFORMATION_MESSAGE);
    }
};

ActionListener listenerCancel = new ActionListener () {
    public void actionPerformed(ActionEvent e) {
        sectionInput.setVisible(false);
    }
};

okayButton.addActionListener(listenerOkay);
```
cancelButton.addActionListener(listenerCancel);

But the OK button should do more than simply display a message; it should actually create a Section object.

First though, we must check that a course has been chosen. When coursesList.getSelectedIndex() is not -1, we know the user has made a selection.

All the other items have been verified, so we can make a call to the constructor to build the Section and then we can add the Meeting objects.

Oops, we can't create a Section since we have omitted the CRN. Add it to the data entry screen. When that is complete, we can create a Section object.

```
// data is okay, so create a new Section with zero capacity
try {
    int tempCRN = (new Integer(crnText.getText())).intValue;
    String tempCourse = (String) coursesList.getSelectedValue();
    String tempSubject = tempCourse.substring(0,4);
    String tempNumber = tempCourse.substring(5);
    Section tempSection = new Section(tempCRN, tempSubject, tempNumber, sectionText.getText(), 0);

    // create and add Meeting objects here
}
catch (CourseException ce) {
    // will not happen
}
catch (SectionException se) {
    // will not happen
}
```

Create and add the meeting objects, in the location shown above.

```
// Meeting objects
String days = "SMTWRFA";
for (int i = 0; i < MAX_MEETINGS; i++) {
    if (hasMeeting[i].isSelected()) {
        for (int j = 0; j < DAYS_IN_WEEK ; j++) {
            if (aRadioButton[i][j].isSelected()) {
                char day = days.charAt(j);
                tempSection.setTime(day + " " + startText[i].getText() + " - " + endText[i].getText());
                break;
            }
        }
        tempSection.setLocation(roomText[i].getText());
    }
}
```
Finally, reset fields on the data entry screen.

crnText.setText("");
coursesList.clearSelection();
sectionText.setText("");
for (int i = 0; i < MAX_MEETINGS; i++) {
    // should erase room, start, end and all days
    hasMeeting[i].setSelected(false);
}

Done!

And then the screen is complete.

    // done
    return sectionInput;
}

Test it.

Modify it as you think appropriate. Some companies have design standards to which you must adhere, but for now, you may customize the data entry screen to suit your preference. Change the colours (foreground and/or background). Change the font. Be creative!

**Summarizing building a data entry screen**

This is a longer summary than the one you saw in the previous chapter.

- The first step in building a data entry screen is to create a JFrame and associate a layout manager with it.

- The second step is to create any widgets (including panels, which will need their own layout managers) which you wish to place in the frame. You may need different layout managers to create the different parts of the screen.

- When using the RelativeLayout layout manager, create the bindings (constraints) associated with each widget. Other layout managers use different terms to control the position of the widgets.

- The third step is to add the widgets and their positioning information to the frame.

- The fourth step is to create any panels which we wish to place within other panels. Create the positioning information associated with each panel. Add the panels and their positioning information to their containers. “Their container” may be the frame or it may be another panel. Repeat the step as many times as necessary, particularly if there are nested panels.
• The fifth step is to examine all the panels. For each, create the (non-panel) widgets the panel contains. Create their positioning information and add them to the panel.

• The sixth step is to add InputVerifiers to all the text fields and ActionListeners to all the buttons, be they radio buttons or JButton.

• The seventh step is to test that everything works properly, something you have been doing throughout all the steps, haven’t you?

Some general comments on data entry

You may need to accept data in a special format, be it phone numbers, postal codes, Canadian Social Insurance Numbers, or American Social Security Numbers. You will find the JFormattedTextField makes your life easier. It is described in the online Java documentation.

When you are entering a large amount of data, you may find the JOptionPane that appear from the InputVerifiers when you make errors interrupt the flow of typing. If that is so, you may wish to avoid the use of JOptionPane.

Instead, do as we did in the previous chapter, add a normally-invisible label to the data entry screen; place an error message in it and make the error message visible whenever the input is in error. The InputVerifier method will need to return true always, or you will not be able to exit from the field.

If you adopt such a method, you will need to verify all the fields when you attempt to accept the data on the screen, perhaps by clicking the okay button.

I have begun placing a JPanel across the bottom of the screen. It is a little different from what we have done above as the panel contains two smaller panels. The first panel (positioned to the left of the screen) contains the error message, or a status message. The second panel (positioned to the right of the screen) is the panel containing the buttons.

I need two panels because I'm using FlowLayout in each I could accomplish the same result with one panel, using RelativeLayout.

Which is preferable? It's probably up to you.

Here are a couple of interesting problems, one of which we have already seen.

Attempt to leave the data entry screen without entering any data.

A person might do this when there were many data entry screens and the person accidentally chose the wrong one. If you select the X in the top-right corner, you are able to leave the data entry screen with no problem. But if you select the cancel button, an error message appears. Why does this happen? More importantly, how can you correct it? Hint, consider this statement.
cancelButton.setVerifyInputWhenFocusTarget(false);

How is that for the name of a method? What does that method do?

Here’s another problem: assume you have entered 2009/02/30 as the date on the Meeting data entry screen you created in the previous chapter. You meant to enter 2009/03/30.

As you leave the day JTextField you will receive an error message. You acknowledge it, and click in the month JTextField. But the day error message reappears. The only way to leave the day JTextField is to provide a valid day within February 2009. Then you may change the month to March, and return to the day field to enter the number 30. Why does such a thing happen? How can you avoid it?

It would be very convenient if Swing contained a date picker of some kind. Unfortunately, it does not yet.

However there are some commercial date pickers available, which may have free demonstration versions.

There are also some open source solutions. NachoCalendar is available from http://nachocalendar.sourceforge.net/. A different picker is available from http://www.roseindia.net/tutorial/java/swing/datePicker.html.

Similarly, suppose your Section data entry screen checks that the end time is after the start time, but you entered the start time incorrectly. How can you fix it? You may be forced to change to a time which is correct and then change it back. For example, a start time of 1130 with an end time of 1120, when you meant to enter 1030 and 1120.

Summary

But a complete system does not involve only data entry screens!

A complete system probably involves a splash screen (something to watch as the rest of the program loads), and screens containing a menu, allowing you access to other parts of the system.

Creating a menu is the topic of the next chapter.
Exercises

1. The section “Some general comments on data entry” speaks about JTextField.
   Explore the capabilities of that class and create a small data entry screen that illustrates those capabilities.

2. (Hard and possibly pointless) Use the GridBagLayout manager to recreate one of the data entry screens we have created in the chapter.
   Which layout manager do you prefer?

3. When you enter data into the start time, end time, or room number text field, the appropriate checkbox is not checked.
   Modify your program so that the appropriate checkbox is checked.

4. The section “Some general comments on data entry” speaks about using invisible fields which become visible to display error information.
   Modify a data entry screen to use that approach.

5. In previous chapters we discussed modelling a bank account.
   In the previous chapter you created a data entry screen for entering a transaction. Complete that screen, adding the choice of the account to which the transaction applies.

6. In previous chapters we discussed modelling a bank account. Assume that your solution has grown into a system which has many accounts. Develop a screen which allows you to display details on a specific account, whose number you must provide by choosing it from a list of account numbers.

7. In previous chapters we discussed modelling the sightings a birder makes. In the previous chapter you created a data entry screen for entering a sighting.
   Complete that screen adding the choice of the species which was sighted.

8. In previous chapters we discussed modelling the sightings a birder makes. Assume your solution has grown into a system containing many sightings.
   Develop a screen which allows you to display the sightings of a particular species, whose name you must provide by choosing it from a list of species names or abbreviations.
9. When a data entry screen first appears, where is the cursor?

Investigate the use of a WindowFocusListener to ensure the cursor is in the field you desire.

```java
sectionInput.addWindowFocusListener(new WindowAdapter() {
    public void windowGainedFocus(WindowEvent e) {
        crnText.requestFocusInWindow();
    }
});
```
Chapter 18 – Creating a GUI: menus

Learning objectives

By the end of the chapter you will be able to:

- Use Swing widgets to create menus
- Describe some of the considerations involved in designing a menu
- Use ActionEvents and ActionListener with menus

Introduction

Having data entry screens is fine, but how do you access them?

In modern computer applications, you often have a menu system that allows you to access the various parts of your system.

Here, we will focus on creating a relatively-traditional menu system. In particular, BlueJ has menus, and BlueJ is a Java application. How did the developers make those menus?

Creating the menus

Definitions

A horizontal menu appears across the top of a frame and is implemented through the JMenuBar class. The Java documentation says that the JMenuBar class is “An implementation of a menu - a popup window containing JMenuItem that is displayed when the user selects an item on the JMenuBar. In addition to JMenuItem, a JMenu can also contain JSeparators.”

That is, a JMenuBar object is the horizontal menu, containing a collection of JMenu objects. A JMenu object is the vertical list of JMenus (leading to another menu), JMenuItem (leading to a frame) and JSeparators which appears when you click an item in the JMenuBar. JSeparators are the visual dividers, usually horizontal lines, you see separating JMenuItem into logical groups.

If Java did not include a JMenu class, could you implement one?

How about using a JButton and a Box? The Box displays the JButton in a vertical layout.
Designing the menu system

So what do we want our menu system to contain?

We need to answer that question before we begin to implement the menu system. But note that we will not be implementing all the functionality behind the menus.

On the JMenuBar, I would suggest, as a minimum, these JMenu items.
- File, containing all choices relating to the files in which we are storing data
- Edit, containing all choices relating to data entry
- Help, containing a help system

On the File JMenu, I would suggest these items.
- New, to create a new college or a new semester. If we were to implement the JMenu in its entirety, we would need new professor, new student, and new section as well.
- Open (a JMenuItem), to work with an existing college or semester. This option would open the XML files saved the last time we exited.
- Save (a JMenuItem), to save any unsaved work, if any.
- Print (a JMenu), to allow us to display timetables, transcripts, list of professors, etc.
- Exit (a JMenuItem), to shut down our program and save all the data to XML files, ready for another time.

On the Edit JMenu, I would suggest the following JMenuItem:
- editing an existing professor
- editing an existing student
- editing an existing course
- editing an existing section

We don't have data entry frames for any of the editing options yet.

On the Help JMenu, I would suggest these JMenuItem.
- Help, to provide access to documents describing how to use our program
- About, to provide a brief blurb about the developers of the program

Implementing the menu


Create a class named CollegeMenu, as follows. It contains only a few methods.

The constructor uses a helper method to create the JMenuBar; the helper method exists to prevent the constructor from becoming excessively long.
A third method, createAndShowGUI, displays the JMenuBar within a JFrame. Note that the constructor is private; we don’t want a menu created except under the control of the createAndShowGUI method.

As in the previous chapters, we need the Swing library for the widgets and the AWT library for the events associated with them.

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

class CollegeMenu {

    private JMenuBar bar;

    private CollegeMenu() {
        bar = createMenuBar();
    }

    public JMenuBar getBar() {
        return bar;
    }

    private JMenuBar createMenuBar() {
        bar = new JMenuBar();

        JMenu fileMenu = new JMenu("File");
        fileMenu.setMnemonic(KeyEvent.VK_F);
        fileMenu.getAccessibleContext().setAccessibleDescription("Deal with files - open, save, etc");

        Recall that we used mnemonics when we established shortcuts in our data entry screens; we use them with menus as well. The accessibility methods are described below.
```
Once fileMenu has been created add it to the menu bar.

    bar.add(fileMenu);

In a similar way, we add JMenus for Edit and Help.

Now the (horizontal) menu bar contains three choices, but the (vertical) menu associated with each choice is empty.

We define a JMenuItem variable for those entries on the File menu which go directly to frames, and a JMenu variable which will be used for other menus.

    JMenuItem menuitem;
    JMenu menu;

And then we create the choices on the File menu; two of the choices lead us to other menus and one does an immediate action.

    // the New submenu
    menu = new JMenu("New");
    menu.setMnemonic(KeyEvent.VK_N);
    menu.getAccessibleContext().setAccessibleDescription("Create a new file");
    fileMenu.add(menu);

    // options on the New submenu
    menuitem = new JMenuItem("Section", KeyEvent.VK_S);
    menuitem.getAccessibleContext().setAccessibleDescription("Create a section");
    menu.add(menuitem);

    // the Print submenu
    menu = new JMenu("Print");
    menu.setMnemonic(KeyEvent.VK_P);
    menu.getAccessibleContext().setAccessibleDescription("Print");
    fileMenu.add(menu);

    // create and add JMenuItem to the print JMenu
    ...

    fileMenu.add(new JSeparator());

    menuitem = new JMenuItem("Exit", KeyEvent.VK_X);
    menuitem.getAccessibleContext().setAccessibleDescription("Close the program");
    fileMenu.add(menuitem);

    // repeat for other JMenuItem

    // repeat for other JMenus

    // done
    return bar;
As we did with the data entry screens, the method above creates the menu bar but does not display it. We need another method which will create the frame into which we can place the menu.

```java
/**
 * Create the GUI and show it.
 */
public static void createAndShowGUI() {
    Runnable runner = new Runnable() {
        public void run() {
            // provide a JFrame in which to display the menu
            JFramejf = new JFrame("COSC 111/121 example");
            jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

            // create the menu and attach it to the frame
            jf.setJMenuBar(new CollegeMenu().getBar());

            // display the frame
            jf.setMinimumSize(new Dimension(450, 260));
            jf.setLocation(100, 100);

            jf.setVisible(true);
        }
    };    
    EventQueue.invokeLater(runner);
}
```

Did you remember the import statements for Dimension and EventQueue?

The code as shown will work, but it is incomplete. It displays the choices but there is nothing behind them; nothing happens when you make a choice. First, though, test the menu system by compiling the CollegeMenu class and then executing its createAndShowGUI method. Here is what you see when it starts.
When you choose File, New, this is what you see.

![COSC 111/121 example](image)

When you choose File, the menu appears with the first item selected. That first item provides access to another menu so you see it.

**Accessibility**

We have used some new methods here, relating to *accessibility*.

Accessibility refers to making computers easier to use by those with different types of sensory challenges. Here, we are providing access for those with serious vision problems, who may use a screen reader to identify the mouse location and the text on which it is located.

An alternative to `getAccessibleContext().setAccessibleDescription` would be to use tooltips for those who don’t use a screen reader. Tooltips appear when you rest your mouse over a choice but do not click a button on the mouse; we saw them in an earlier chapter. An example is

```java
menuItem.setToolTipText("Create a new file");
```

To accommodate all users, you should probably use both `AccessibleDescriptions` and `ToolTips`.

**Completing the menu**

Now complete the other parts of the menu system.

As you complete the menu, you may run across an interesting challenge, the duplication of mnemonics. We have seen that a mnemonic is a single letter abbreviation you can use to make a selection from a menu. S for Save and P for Print are common mnemonics.
On the Edit menu, what mnemonics did you use for Student and Section?

As a general rule, you shouldn’t have the same mnemonic for two selections on the same menu. To avoid duplication, I used T for student and S for section. You could use S for student and E for section if you prefer. Or you could use S for both.

When you have duplicates and you cannot find suitable different mnemonics, pressing [Alt] and the mnemonic repeatedly should switch from one of the duplicate entries to the next, in a circular fashion.

The other problem you run across is that the vertical, drop-down, menus are divided into logical sections (New, Open, and Save all deal with files) and you usually wish to separate one logical group from another. Use JSeparators, adding one to the menu wherever appropriate.

```java
fileMenu.add(new JSeparator());
```

### Performing actions

**ActionListeners for each menu option**

Some of the choices on a menu simply lead to menus of other choices. Others, however, lead to data entry screens, or query screens (but we won’t have any of those). How do we invoke methods to do these actions?

We use ActionListeners again, one for each menu selection. Don’t forget the import statement!

Start with the easiest, the ActionListener for File, Exit.

```java
ell = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
};
menuItem.addActionListener(ell);
```

A similar ActionListener is the one for Help, About.

```java
ell = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null,
            "Created as an example in COSC 111/121.",
            "About", JOptionPane.INFORMATION_MESSAGE);
    }
};
menuItem.addActionListener(ell);
```

A more-complicated ActionListener is for File, New, Section.
ell = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        Section.showFrame();
    }
};

menuItem.addActionListener(ell);

As noted above, choosing File presents this image.

With the ActionListeners in place, clicking Section will present the data entry screen from chapter 16. Clicking Exit will close the application.

**One ActionListener for all menu items**

As an alternative, some designers will suggest that the CollegeMenu class itself will implement the ActionListener interface, rather than having ActionListeners for each menu selection. I prefer not to do this, but include it to expand your knowledge of tools available to you.

```java
public class CollegeMenu implements ActionListener
```

Each menuItem variable must register itself with the listener.

```java
menuItem.addActionListener(this);
```

As is usual, the word this refers to the current object, the menu itself, which is an ActionListener, since it implements the ActionListener interface.

When one object acts as a listener for many other objects, we must be able to distinguish between the “many other objects.” We use the setActionCommand method to identify each menu choice. For example,

```java
menuItem.setActionCommand("exit");
```
The listener uses the `ActionPerformed` method to identify which menu selection is made. This is a simple version, listening only for the Exit and About JMenuItems.

```java
public void actionPerformed(ActionEvent e) {
    JMenuItem source = (JMenuItem)(e.getSource());
    if (source.getActionCommand().equalsIgnoreCase("exit")) {
        System.exit(0);
    }
    if (source.getActionCommand().equalsIgnoreCase("About")) {
        // display message box
        JOptionPane.showMessageDialog(null,
            "CollegeMenu - demo program for COSC 111/121",
            "About", JOptionPane.INFORMATION_MESSAGE);
    }

    // do something for each other possibility
    return;
}
```

The `getActionCommand` method allows us to identify which menu item has been chosen. It doesn't use the word on the menu, so doesn't need to be changed should we internationalize the menu. It uses the word we have associated with the menu; in English they may be the same.

Note that using `equalsIgnoreCase` allows you to be careless in specifying the ActionCommands. Did you notice that I spelled exit in two different ways?

How do you make a data entry screen appear when the user makes the appropriate choice on the menu? Each data entry screen has a method to make it appear, so you simply invoke that method.

That is, suppose you have a JMenuItem whose `ActionCommand` is `newSection`; this corresponds to the menu item you will select to create a new Section object.

```java
    if (source.getActionCommand().equalsIgnoreCase("newSection")) {
        Section.buildFrame();
    }
```

In the previous chapter we saw how to create the `Section` data entry screen.

Here is an interesting observation. I have spoken about making data entry screens invisible so they will be available to be shown again if needed, without rebuilding them. But the code I have written rebuilds them every time.

How do we reuse hidden screens? The first step is to separate the construction of the screen from the display of the screen.

To prevent rebuilding screens, one possibility is to use the Singleton pattern. If that is the solution, we know how to do that since we have used Singleton many times already.
A second possibility is to for the CollegeMenu class to track the screens which exist. If the screen the user wants has been created, make it visible. If not, create it, remember that it has been created, and make it visible.

A question in the exercises asks you to pursue this subject further.

**Toolbars**

Modern applications duplicate many items from the menu on a toolbar. Common actions are placed on a toolbar which is always visible instead of forcing the user to make choices from a menu. The Swing package supports toolbars with the JToolBar class.

If you decide to use toolbars, you should look at http://java.sun.com/developer/techDocs/hi/repository/, which provides a collection of icons that are designed for use on toolbars. This URL may change in the future, now that Oracle has purchased Sun Microsystems.

**Help**

Implementing a help system is a complicated task, since it should be able to answer any questions the user has about your program. Rather than going into great detail about implementing a help system, I’ll refer you to the website for JavaHelp.

This is described as “an open source software, a full-featured, platform-independent, extensible help system that enables you to incorporate online help in applets, components, applications, operating systems, and devices.” Note that there has not been a lot of development work on this JavaHelp since 2007 but it has recently been revitalized. Perhaps a more-active product should be used.

For more details on JavaHelp, see http://java.net/projects/javahelp/ Other similar packages may be available. I haven’t used them.

**Internationalization**

**Definition**

Seeing data entry screens and a menu system in English raises the question “what does it take to use a different language?”

How would I translate your system into French? Into Russian? Into Japanese?
Let me introduce a common abbreviation – i18n. This is a common geeky abbreviation for internationalization, since there are 18 characters between the first and last letters of internationalization.

**Steps to follow**

So how do you translate a system into another language?

A good place to get general information is [http://download.oracle.com/javase/tutorial/i18n](http://download.oracle.com/javase/tutorial/i18n), the portion of the Java tutorials dealing with i18n. This trail goes into more details than I will here.

First, you need to identify the words which need to be translated. These will be removed from your program (replaced by generic terms) and stored in a properties file. These include labels, menu items, error messages, tooltips, accessibility descriptions, *etc*.

Second, you need someone to do the translation.

For example, your File menu contains the option New. Perhaps you will use generic term createAFile. The code that follows contains the exact statements you need.

In a properties file for a French translation, you will place a line that says createAFile = Nouveau. The word nouveau has been provided by your translator.

For the word Open, use the generic term openAFile.

In a properties file for a French translation, you will place a line that says openAFile = Ouvrir. Ouvrir has been provided by your translator.

Note that the mnemonic may change as well as the term. Thus you may need createAFileMnemonic = N and openAFileMnemonic = O in your properties file.

Note also that the translated term may be longer than the term it is replacing. This may have implications for the spacing between widgets on your data entry screens.

Third, insert the translations into the properties file.

Fourth, you need save your properties file. Its name should be MessagesBundle_xx_yy.properties. MessagesBundle is the name chosen to reflect the fact that the file contains a collection (colloquially called a bundle) of messages, although you could use some other name if you wish. The second and third parts of the name are the international abbreviation for the language and the international abbreviation for the country. The language abbreviations are available at [http://www.loc.gov/standards/iso639-2/php/code](http://www.loc.gov/standards/iso639-2/php/code) and the country abbreviations (in English) are available at [http://www.iso.org/iso/country_codes/iso_3166_code_lists/english_country_names_and_code_elements.htm](http://www.iso.org/iso/country_codes/iso_3166_code_lists/english_country_names_and_code_elements.htm).
Note that one language may be used in different countries, with the vocabulary differing between countries. For example, the French spoken in Québec (a province of Canada) has differences from the French spoken in France, from that spoken in Switzerland, and from that spoken in Burkina Faso (a landlocked country in West Africa which gained its independence from France in 1960.) Thus, if we were making a program that could be used in those four countries, we would need four properties files.

MessagesBundle_fr_CA.properties
MessagesBundle_fr_FR.properties
MessagesBundle_fr_CH.properties
MessagesBundle_fr_BF.properties

For languages which do not use the Latin alphabet (alphabetic characters whose ASCII values are between 0 and 127) you need an additional couple of steps before you create the properties file.

First you need to create a file (whose name doesn’t matter) containing the statements you would expect to see in the properties file, but with the translations in the native alphabet. For example, here is the line for New, in Russian, according to one of my students.

createAFile = Новый

Create the file, ensuring that it is saved using the UTF-8 standard. You do not need a fancy program to do this. Notepad, provided with Windows, will save in UTF-8 as long as you click UTF-8 in the dropdown list labelled “Encoding.”


Once the file is created, use the native2ascii program (available as part of the Java download) to convert its contents to a file which contains only ASCII characters and Unicode characters.

To use native2ascii, run it at the command line. In my case, I used the command

native2ascii sergey.properties MessagesBundle_ru_RU.properties

since Sergey was the student who did the translation for me. Thank you, Sergey.

MessagesBundle_ru_RU.properties now contains a line which says

createAFile = \u041d\u043e\u0432\u044b\u0439

All the Cyrillic characters have been replaced with Unicode characters.

The fifth and final step in internationalization involves writing Java code to replace the generic terms you chose (createAFile, for example) with the appropriate word.

This is all shown below.
The Java statements to make the magic happen

In the method which shows the menu (createAndShowGUI), declare two variables, one for the language and one for the country. Both are Strings.

```java
String language;
String country;
```

These variables need to be given values. The easiest way is through parameters to the method, as createAndShowGUI is the first method which should be run. That is, replace the method header

```java
public static void createAndShowGUI() {
```

with

```java
public static void createAndShowGUI(final String language, final String country) {
```

Use these two variables to create a Locale object and a ResourceBundle object. Remember to import java.util.Locale and java.util.ResourceBundle.

```java
Locale currentLocale = new Locale(language, country);
ResourceBundle messages = ResourceBundle.getBundle("MessagesBundle", currentLocale);
```

The `getBundle` method looks for the properties file that corresponds to the correct language and country.

The variable `messages` becomes a parameter to any method that has been internationalized, the first of which is the constructor.

```java
frame.setJMenuBar(new CollegeMenu(messages).getBar());
```

The constructor becomes

```java
public CollegeMenu(ResourceBundle messages) {
    bar = createMenuBar(messages);
}
```

The `createMenuBar` method (note that it now has `messages` as a parameter) then uses statements like

```java
JMenuItem openMenuItem = new JMenuItem(messages.getString("createAFile"));
```

to extract the appropriate text from the ResourceBundle.

`createAFile` is simply the symbols you have chosen to represent the word, the `messages.getString` method retrieves the appropriate translation.
Note that a ResourceBundle is actually an instance of a map. The generic term we used (createAFile, for example) is the key and the translation is the value.

When you display your menu and data entry screens, all the text now comes from the translation.

**Who did the translation?**

You didn’t have someone to do the translation for you? You could simply create a file for English as it is spoken in Canada. That is, language would be entered as “en” and the country as “CA.”


**Conclusion**

This is only an introduction to the subject of i18n. There is much more to using ResourceBundles and Locales.

For example, how do you deal with different formats for numbers; some cultures use one format for numbers, others use a different format.

How do you deal with error messages that are put together from smaller messages? The order in which the words occur may differ from culture to culture.

Many other cultural differences (colour, for example) need to be considered.

**Summary**

There is still a lot of work to do to make your system a complete, working system, but there are other ideas we should pursue. You have seen the basics (and more) of how to design a program, entering data, storing data, and manipulating data. You have seen a large amount of Java.

However, there are additional aspects of programming, and of Java, which do not fit well into the example of a college. Thus, we’ll set aside the college, its students, and its professors as we consider other ideas and examples.
Exercises

1. Complete the CollegeMenu class started in the chapter.

2. You will notice that I have placed my menu in one frame, and the data entry screen in other frames. How would you place the menu and data entry on the same frame?

3. In previous chapters we discussed modelling a bank account. Create a menu system for a banking system involving many accounts.

4. In previous chapters we discussed modelling the sightings a birder makes. Create a menu system for a birding system involving many sightings.

5. Explore the subject of internationalization. As the world’s economies become more and more interrelated, internationalization will become more and more important.

   In particular, how well do the translation websites I mentioned work? Find someone who speaks a language different from you and explore the accuracy and appropriateness of the translations. Remember there are English sentences like “She saw that saw.” in which the same word may appear as both a noun and a verb. Other languages may have such challenges.

   How does the length of your translation compare to the English words? Does that affect your screen design?

6. Examine the question of reusing hidden screens, as discussed in the chapter.

   Use the Singleton pattern for each data entry screen. Or, create within CollegeMenu a collection of the screens which have been created and can be reused.
Chapter 19 – Applets

Learning objectives

By the end of the chapter you will be able to:

- Define applet
- Use the predefined methods which applets provide
- Create applets to carry out a variety of tasks

Introduction

Up to now, we have been building parts of an application as a stand-alone program which runs on its own. The GUI is part of the application and so are all the classes.

But Java can also be used to create applets.

These are small programs, embedded within a web page. They are described in most books on Java, and online, for example at http://java.sun.com/applets. This site includes many sample applets.

Java is not the only language used to embed applications within web pages; Flash is an alternative, for example.

Let’s see how we can create an applet.
Creating an applet

Open BlueJ and create a new project.

Then create a new class in that project, but make sure it is an applet, not an ordinary class.

Traditionally, the first program students write is named HelloWorld, so we will use that name for the first applet you write.
Well, it looks like a class on its class diagram, but it is clearly marked as an applet.

**Running an applet**

I know we have added no code to the applet, but BlueJ has created a complete, workable, although minimally-functional, applet for us from one of its templates. To run the template, right-click the applet and select Compile the Applet, and then right-click the applet again and select Run Applet. The window that below appears.
The appropriate radio button has been chosen, and the suggested height and width are fine. Click Ok, and the applet opens, using an application called appletviewer. Or is it AppletViewer?

Much of what you see comes from AppletViewer; in particular, the menu allowing you to control the applet comes from AppletViewer. The output of the applet is two lines of text. Terminate AppletViewer.
The name in the title bar may be truncated; three dots at the end of the name (an ellipsis, which we have seen before) indicates truncation. Due to the font chosen and the width of the frame, there may not be enough room to display the full name. If the name is truncated, run the applet again, but increase its width.

Run the applet a third time, but now select “Run applet in web browser.” Then click Ok. The applet runs, but this time in your default web browser. In my case, it started on a new tab in Firefox, as Firefox was already running. If the browser was not running, it would have started.

How does the applet do its work? Let’s open it in the editor and see.

**An applet (as created by BlueJ) under the microscope**

Here is the code as it appears in the editor (formatted as usual to fit on the printed page.) This is much longer than most applets you’ll see in textbooks, but don’t be concerned. Most of the length is documentation. After the code, we’ll examine the parts of the applet in detail.

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

/**
 * Class HelloWorld - write a description of the class here
 * @author (your name)
 * @version (a version number)
 */
public class HelloWorld extends JApplet
{
    // instance variables - replace the example below with your own
    private int x;

    /**
     * Called by the browser or applet viewer to inform this JApplet that it has been loaded into
     * the system. It is always called before the first time that the start method is called.
     */
    public void init()
    {
        // this is a workaround for a security conflict with some browsers including
        // some versions of Netscape & Internet Explorer which do not
        // allow access to the AWT system event queue which JApplets do on startup
        // to check access. May not be necessary with your browser.
        JRootPane rootPane = this.getRootPane();
        rootPane.putClientProperty("defeatSystemEventQueueCheck", Boolean.TRUE);

        // provide any initialization necessary for your JApplet
    }

    /**
     * Called by the browser or applet viewer to inform this JApplet that it should start its
     * execution. It is called after the init method and each time the JApplet is revisited
     */
```
public void start()
{
   // provide any code required to run each time
   // web page is visited
}

/**
 * Called by the browser or applet viewer to inform this JApplet that it should stop
 * its execution. It is called when the Web page that contains this JApplet has
 * been replaced by another page, and also just before the JApplet is to be destroyed.
 */
public void stop()
{
   // provide any code that needs to be run when page is replaced by another page or
   // before JApplet is destroyed
}

/**
 * Paint method for applet.
 *
 * @param g the Graphics object for this applet
 */
public void paint(Graphics g)
{
   // simple text displayed on applet
   g.setColor(Color.white);
   g.fillRect(0, 0, 200, 100);
   g.setColor(Color.black);
   g.drawString("Sample Applet", 20, 20);
   g.setColor(Color.blue);
   g.drawString("created by BlueJ", 20, 40);
}

/**
 * Called by the browser or applet viewer to inform this JApplet that it is being reclaimed
 * and that it should destroy any resources that it has allocated. The stop method will always
 * be called before destroy.
 */
public void destroy()
{
   // provide code to be run when JApplet is about to be destroyed.
}

/**
 * Returns information about this applet.
 * An applet should override this method to return a String containing information about
 * the author, version, and copyright of the JApplet.
 */

public String getAppleInfo()
{
    // provide information about the applet
    return "Title: \nAuthor: \n" + "A simple applet example description. ";
}

/**
 * Returns parameter information about this JApplet.
 * Returns information about the parameters than are understood by this JApplet.
 * An applet should override this method to return an array of Strings
 * describing these parameters.
 * Each element of the array should be a set of three Strings containing the name, the type,
 * and a description.
 * @return a String[][] representation of parameter information about this JApplet
 */
public String[][] getParameterInfo()
{
    // provide parameter information about the applet
    String paramInfo[][] = {
        {"firstParameter", "1-10", "description of first parameter"},
        {"status", "boolean", "description of second parameter"},
        {"images", "url", "description of third parameter"}};
    return paramInfo;
}

Whew!

The applet begins with two import statements. As we have seen in the previous chapters, Java has two graphics libraries, the AWT (Abstract Window Toolkit) and Swing. Applets use both, as we did when we built our GUI. Events come from the AWT and widgets come from Swing.

JApplet is a Swing class that extends Applet, an AWT class, and the applet we are creating extends JApplet. That is, HelloWorld is a JApplet (as well as being an Applet.)

A JApplet must implement several methods. These are init, start, stop, paint, and destroy. We look at each in turn.

getAppleInfo and getParameterInfo should be implemented as a favour to people who wish to use the applet. We look at each in turn.

init

The documentation BlueJ creates states it all.

/**
 * Called by the browser or applet viewer to inform this JApplet that it has been loaded
 * into the system. It is always called before the first time that the start method is called.
Many times the init method will do nothing. In particular, the workaround that is shown appears to be unnecessary on my computer.

Should you have an applet that requires a specific version of a browser (or a specific browser or operating system), then the init method would be where you check. To see how to determine the browser or operating system, examine the getProperties method of the System class. The following statement will display perhaps more than you want to know.

```
System.getProperties().list(System.out);
```

assuming, of course, that you have imported java.util.Properties or you execute that statement in the Code Pad; then you don't need the import statement.

When you have any GUI components in your applet, init is the place to define them and add their listeners. Yes, applets can contain buttons and text fields and …

As one example among many, consider
https://www.valleyfirst.com/Personal/ToolsAndCalculators/Calculators/RetirementPlanner/ Yes, the author is nearing retirement. We will not do something so complicated. Instead, we will be drawing pictures.

In fact, much of what we saw in the previous three chapters can be applied to an applet. Just remember that an applet cannot save data to a file on your computer without asking your permission.

**start**

The documentation BlueJ creates gives the purpose of the start method.

```
/**
   * Called by the browser or applet viewer to inform this JApplet that it should start its execution.
   * It is called after the init method and each time the JApplet is revisited in a Web page.
   */
```

Applets cannot open files on the machine on which they are running without asking your permission, but they can contact other websites to download images, files, or other data. Should your applet do that, the code to do it is placed in the method.

Of course, the method may do nothing.

**stop**

The documentation BlueJ creates gives the purpose of the stop method.

```
/**
```
* Called by the browser or applet viewer to inform this JApplet that it should stop its execution.
* It is called when the Web page that contains this JApplet has been replaced by another
* page, and also just before the JApplet is to be destroyed.
*
Perhaps the applet is playing some music and it should stop.

Perhaps it is running an animation and it should stop.

This method may do nothing.

**paint**

This is the method which does all the heavy lifting in an applet. That is, here is the method in which you will write most (perhaps all) of your Java statements. The documentation BlueJ creates does not let you know this.

An applet is a graphics program. As such, it needs an area on which to draw. This area, called a **graphics context** (an object of type Graphics) is provided as a parameter and then you can issue whatever commands you need to produce the graphics output you wish.

As you see from the sample BlueJ provides, these commands allow you to specify colours, to draw filled rectangles (and other shapes as well), and to write text (beginning at a specified position).

```
g.setColor(Color.white);
g.fillRect(0, 0, 200, 100);
```

This creates a white rectangle. Its upper corner is in the upper left corner of the window (and has co-ordinates 0, 0). The rectangle has a width of 200 pixels and a height of 100 pixels. It is hard to see the rectangle, since it appears on a white background. To see the rectangle, change it to a different colour, perhaps Color.yellow.

```
g.setColor(Color.black);
g.drawString("Sample Applet", 20, 20);
```

This changes the foreground colour to black. Until we change the colour, everything we draw will be in black. The second statement displays some text. Of course it is in black. The lower left corner of the leftmost character is 20 pixels from the left edge and 20 down from the top.

```
g.setColor(Color.blue);
g.drawString("created by BlueJ", 20, 40);
```

This changes the foreground colour to blue and displays some more text, a little further down in the graphics context.
To see the other methods you can use to draw, look at the online documentation of the Graphics class. You will have the opportunity to use some of the other methods in the exercises for the chapter.

Note that the rectangle will always be 200 pixels wide by 100 pixels high, regardless of the size of the window in which the applet is running. You can determine the size of that window by using the getSize method. The expression this.getSize() returns a Dimension object that provides the height and width.

```java
Dimension windowSize = this.getSize();
double width = windowSize.getWidth();
double height = windowSize.getHeight();
```

Exercise 3 provides an alternative way to determine the height and width.

Can someone please explain why the constructor for a Dimension object takes int parameters, but the getters return double?

destroy

The documentation BlueJ creates gives the purpose of the destroy method.

```java
/**
 * Called by the browser or applet viewer to inform this JApplet that it is being reclaimed and that 
 * it should destroy any resources that it has allocated. The stop method will always be called 
 * before destroy.
 */
```

Perhaps the applet has allocated some memory and it should be released. Perhaps it has established a connection to another website somewhere and that connection should be severed.

Of course, the method may do nothing.

getAppletInfo

The BlueJ documentation gives the purpose of the getAppletInfo method. You can see that it is similar to the toString methods we have written in all of the classes we have developed earlier.

```java
/**
 * Returns information about this applet.
 * An applet should override this method to return a String containing information about the author,
 * version, and copyright of the JApplet.
 * @return a String representation of information about this JApplet
 */
```
It is only politeness that you should provide meaningful information from the method. The default value of null is not informative.

**getParameterInfo**

When you run an applet, you can provide it with input. This information is provided through parameters. You can see such behaviour in the window which appears when we run the applet.

![Applet Parameters](image)

If you wish to specify a parameter, specify its name in the box provided, specify its value in the box provided, and click Add. Repeat for as many parameters as you need, and then click Ok.

If someone provides you with an applet for which some parameters are already defined, simply click the parameter (in the window on the left) and it will appear in the text boxes on the right; you can change its value there.

generate a way to find out which parameters an applet will process.

The BlueJ documentation describes **getParameterInfo** in more detail.

```java
/**
* Returns parameter information about this JApplet.
* Returns information about the parameters than are understood by this JApplet.
* An applet should override this method to return an array of Strings describing
* these parameters. Each element of the array should be a set of three
* Strings containing the name, the type, and a description.
* @return a String[] representation of parameter information about this JApplet
*/
```
This method returns a two-dimensional array (the documentation does not make this completely clear. In fact, the documentation is in error! Perhaps it will have been changed by the time you read this.), the elements of which are Strings.

Interpret the two-dimensional array as a one-dimensional array, each of whose elements (think of them as rows) is a one-dimensional array whose three elements (think of them as columns) are the name of the parameter (element 0), the type of the parameter (element 1), and a description of the parameter (element 2).

Notice the way you can declare an array and give its elements at the same time, something we saw earlier. Thus

{"firstParameter", "1-10", "description of first parameter"}

represents an array, containing three elements, all Strings, and

String paramInfo[][] = {
  {"firstParameter", "1-10", "description of first parameter"},
  {"status", "boolean", "description of second parameter"},
  {"images", "url", "description of third parameter"}};

declares an array of three elements, each of which is itself an array of three elements.

You can refer to individual elements of paramInfo by giving the individual row and column subscripts. Thus paramInfo[1][2] is “description of second parameter.” paramInfo[2][1] is “url.” Trying to access paramInfo[3][3] will cause an exception (an ArrayIndexOutOfBoundsException) to be thrown. Remember that both row and column subscripts start at 0.

When you use AppletViewer to view the applet, you can specify those parameters. When you run the applet from a web browser, you can also specify those parameters.

But you can’t specify the parameters unless you know what they are called. The getParameterInfo method will tell you about the parameters.

The applets we create here do not use parameters, but you will have the opportunity to do so in the exercises. To complete those exercises, you will need to explore the getParameter method of the Applet class.

**Hello world**

Since Kernighan and Ritchie published their book on C, in which the first sample program was to display the words “Hello World”, beginning programmers have learned how to say hello in many languages; you will do so as well.

You are definitely not a beginning programmer if you have covered the previous chapters, but writing a Hello World program is something all programmers do.
To follow in the steps of those who have gone before, modify the applet BueJ provided to display the message Hello World, in green text on a purple background, in 32 point text.

Oh, purple is not one of the constants in the Color class!

All colours can be constructed from a mixture of red, green, and blue. Google the words RGB purple and you will find many sites which tell you that purple is something like 160-32-240. The colours (the first number refers to the amount of red in the colour, the second to the amount of green, and the third to the amount of blue) are measured on a scale from 0 (none) to 255 (all). Thus 0-0-0 is black, and 255-255-255 is white. 160-32-240 is interpreted by our brain as purple.

```
Color purple = new Color(160, 32, 240);
g.setColor(purple);
```

What is so special about the number 255?

Once we have the colour we want, we create a rectangle in which we can display the text.

```
g.fillRect(0, 0, 200, 100);
```

Before we display the text, we need to create the correct font, based on the default font.

```
g.setFont(g.getFont().deriveFont(Font.PLAIN,32));
```

Then we set the color of the text, using a predefined value.

```
g.setColor(Color.green);
```

You could use Color.GREEN; that is also a predefined value.

And we display the text.

```
g.drawString("Hello World", 20, 50);
```

Ugly, isn’t it? That is, green on purple is ugly, not the Java code to create green text on a purple background.

**A problem**

Sometimes, when I resize the applet in AppletViewer, the contents disappear. When I minimize AppletViewer and then restore it, the contents remain.

This behaviour does not happen when I run the applet in my web browser.

Why does it happen in AppletViewer and how can you fix it?
Squares

You can create a “rectangle painting” by placing a series of filled rectangles on the graphics context. Or you can create something like Piet Mondrian's *Composition with Yellow, Blue, and Red*, 1937–42, oil on canvas, 72.5 x 69 cm, apparently on display at the Tate Gallery. London

![Image of Mondrian's Composition with Yellow, Blue, and Red](image)

Make it so.

Canadian flag

Or you can create your national flag. Being Canadian, I’ll create a Canadian flag, but not a waving Canadian flag.

![Image of Canadian flag](image)

If I draw the flag as a rectangle, the bars and background are easy to draw. The maple leaf is the challenge.

You need to do is identify the points on the maple leaf where the line surrounding it changes direction and then provide that information to the `drawPolygon` method. This is a new-to-us method which allows us to draw a polygon, given the x and y co-ordinates of its corners. The x co-ordinates are given in one array, the y co-ordinates in the other. The arrays are associated since the same element in each represents the x and y co-ordinates of one point.
http://www.pch.gc.ca/pgm/ceem-ceed/symbl/df2-eng.cfm provides a useful image. I used a drawing program to identify the approximately 30 points I needed to define the vertices of the polygon adequately.

Make it so.

As an aside, what does the world think of Canada?

While looking for this image in Microsoft® Office Word 2003’s library, I found images of maple syrup, blueberry pie, hockey, skating, tuques, and Nanaimo bars, along with the maps, flags, and flowers of the provinces, and a loon!

Later versions of Word appear to have lost some of the images. I didn’t notice the maple syrup and blueberry pie, for example.

If your flag contains stars, you could use the same technique to identify the vertexes on the star, but it would be better to create a method to draw stars. An even better method is to have a method which draws stars having different numbers of points, so that it will be more useful.

We explore flags and stars more in exercise 6.

**Creating HTML to run an applet**

While we have used AppletViewer to run our applet, applets are more often viewed via a web browser. Thus you need to create a webpage which contains the applet. BlueJ allows us to create a webpage for an applet very easily.

Compile the applet; I’ve chosen the Canadian Flag applet. When you run the applet, don’t tell BlueJ to run it in AppletViewer or the web browser; tell BlueJ to “create webpage only.”
In my case, since the applet is called CanadianFlag.java, the webpage is called CanadianFlag.html and is located in the same folder as the applet.

If you wished to create the webpage with another name, you could just rename the automatically-created page. Or you could right-click the applet in the class diagram; select Run Applet, Generate web page only, and click Ok.

A “Save As” window appears, titled “Select HTML page destination.” Double-click the Applet folder name, and then provide a name for the HTML page. I chose CanadianFlagTest and the extension .html was appended automatically.

Here are the contents of the html file.

```html
<html>
<!-- This file automatically generated by BlueJ Java Development -->
<!-- Environment. It is regenerated automatically each time the -->
<!-- applet is run. Any manual changes made to file will be lost -->
<!-- when the applet is next run inside BlueJ. Save into a -->
<!-- directory outside of the package directory if you want to -->
<!-- preserve this file. -->
<head>
  <title>CanadianFlag Applet</title>
</head>
<body>
  <h1>CanadianFlag Applet</h1>
</body>
</html>
```
But much of the HTML code is unnecessary. The simpler version of the HTML file is:

```html
<html>
  <head>
    <title>CanadianFlag Applet</title>
  </head>
  <body>
    <h1>CanadianFlag Applet</h1>
    <hr>
    <applet code="CanadianFlag.class"
            width=500
            height=500 >
    </applet>
  </body>
</html>

A quick introduction to HTML

Should you not be familiar with HTML, here’s the quick introduction. You saw a similar idea if you saw the chapters using XML.

HTML (Hypertext Markup Language) is a language used to describe the way web pages will appear when viewed with a web browser.

The description of each web page begins with <html> and ends with </html>. These are called tags; capitalization doesn’t matter. You have seen different tags when we used XML to serialize objects.
Within a web page, there are two sections, the heading and the body. The heading is surrounded by <head> and </head> tags, and the body is surrounded by <body> and </body> tags.

Within the heading, you can specify a title, surrounded by <title> and </title> tags, which appears as the title of the window in which the web page is displayed.

Within the body of the web page, you may have headings; these control the appearance of headings (text) which you use to separate parts of your webpage. The <h1> and </h1> tags mark the largest type of heading, <h6> and </h6> (unused here) mark the smallest.

<hr> (without a matching </hr>) inserts a horizontal line across the screen.

<applet> and </applet> mark the instructions which identify the applet to run (code=), and the size of the window in which it is to appear. Should you omit the size, the applet will run in a window of default size but, in my case, the default is too small to show the entire flag.

Should you wish to add some text to the web page (not shown here), surround each paragraph you wish to add with <p> and </p> tags. For example <p>This is the national flag of the great country of Canada. Its national holiday is July 1, celebrating the day Canada became a nation in 1867.</p>

Should you wish to add comments within your web page (see the second line of the automatically-generated web page), comments begin with <!-- and end with -->, and may extend over many lines if you wish. These comments will not appear when you display the web page.

**Summary**

Applets are graphics programs designed to run in a browser. Should you develop websites, you find you use them extensively. If not, you may not use them at all.

I have included them here as many students enjoy creating visual artefacts, and because they can be fun. The exercises actually contain quite a bit of new material of applets. Enjoy them!

In the next chapter, we will step back from Java for a moment, and look at some of the underpinnings of good analysis and design. We will look at patterns, solutions which have worked for others and which you can adopt to make your own programs better.
Exercises

1. Create a simple applet which displays your name.

Modify the applet so it accepts parameters to specify the size of the frame. First, decide on the names of your parameters. I would suggest length and width should be the names. When you use BlueJ to run the applet, specify those parameters and their values. That is, provide the Name of the parameter and its Value. Then click Add.

If you don’t specify the size, your applet should default to some reasonable value.

To accomplish all this, begin by declaring instance variables for your applet. Create a loadParameters method and call it from the init method. Let’s start with height and width.

The declarations are

    private int width;
    private int height;

and the loadParameters method is

    private void loadParameters() {
        String parameter = getParameter("width");
        if (parameter != null)
            width = new Integer(parameter);
        else
            width = 300;
parameter = getParameter("height");
if (parameter != null)
    height = new Integer(parameter);
else
    height = 400;
}

This assumes the height and width are actually numeric. How should you write the applet to deal with input which is not numeric?

Modify the applet so it doesn’t always display your name. It should display the name provided as a parameter, with your name being the default value.

Accomplish this by declaring an instance variable and then establish its value in loadParameters. If the name you wish to enter contains one or more spaces, ensure you enter it in quotation marks. Why is this necessary?

Modify the getParameterInfo method to explain the parameters the applet accepts. To test this method, create a new applet on the workbench, and execute its getParameterInfo method.

Modify the applet so it uses the colour provided as a parameter. You should probably make sure the applet accepts both the name color (the American spelling) and colour (the Canadian and English spelling). Ensure there is an appropriate default colour.

2. Create a simple applet to draw some geometric shapes. You may draw circles and ovals using the drawOval and fillOval methods. An oval is a circle drawn within a non-square rectangle. You may draw squares and rectangles with the drawRect and fillRect methods.

Other polygons are drawn using the drawPolygon and fillPolygon methods. You will need to explore the Graphics class to use these methods.

3. Create an applet which generates a circle in the centre of its graphics context. To determine the centre, use something like the following statements.

    Rectangle r = g.getClipBounds();
    int x = (r.x + r.width)/2;
    int y = (r.y + r.height)/2;

    The Rectangle that getClipBounds returns has its top left corner, represented by x and y, as 0, 0. The Rectangle’s width and height are the values you (or BlueJ) specify when you run the applet. Assume for a moment that you use the height and width that BlueJ suggests.

    When you draw a circle, you are specifying the corners of the box surrounding it. Thus when you use the statements

        int width = r.width / 4;
        int height = r.height / 4;
to determine the size of the box and then use the statement

g.drawOval(x, y, width, height);

to draw the circle, you will not draw a circle in the centre of the graphics context. You will
draw a circle in an invisible box, whose **top left corner** is at the centre of the graphics
context.

To actually draw a circle whose centre is in the centre of the graphics context, you need to
use the following statement.

g.drawOval(x - width / 2, y - height / 2, width, height);

If you specify that the applet runs in a non-square frame, the calculation of the centre of the
frame is the same. But the statement above will draw an oval, not a circle. What value
should you use as the diameter of the circle if we really want to draw a circle?

I would suggest the minimum of the height and width you have calculated.

```java
int diameter = Math.min(height, width);
int radius = diameter / 2;
g.drawOval(x - radius, y - radius, diameter, diameter);
```

Make the circle move from the centre along a path whose angle you specify, in degrees (or
radians, if you prefer), through a parameter. This involves a little trigonometry.

When you move in a direction, any direction, there are at least two ways to do so. You
could go directly to your destination, or you could do it in two steps; move a certain
distance in one direction and move a certain distance in another direction. Think of a right-
angled triangle. You could follow a route along the hypotenuse (the long side), or you could
follow a route along the other two sides. Both routes get you to the same destination. They
just take different amounts of time.

We can calculate the distances in the x and y directions by using the cosine and sine
functions, applied to the angle we specified. Thinking again of the right-angled triangle, we
can’t determine the lengths of the sides unless we also know the length of the hypotenuse.
Consider the statements below. We have used a `loadParameters` method to provide a value
for angle, in degrees.

```java
// distance to move radially
int hypotenuse = diameter / 20;

double doubleAngle = Math.toRadians(angle);
int deltaX = new Long(Math.round(Math.cos(doubleAngle) * hypotenuse)).intValue();
int deltaY = new Long(Math.round(Math.sin(doubleAngle) * hypotenuse)).intValue();
```
The first two lines decide the maximum distance the circle will move in a radial direction, out from its current centre. I chose this distance since it is large enough to see, but small enough that it will take a reasonable number of steps to move the circle a long distance.

The trigonometric functions in the Math library expect the angle to be in radians, but we have specified the angle in degrees. The toRadians method converts degrees to radians, giving a double, which the trigonometric functions expect.

The distance the circle moves in the x direction is given by the hypotenuse times the cosine of the angle. The distance the circle moves in the y direction is given by the hypotenuse times the sine of the angle.

The conversions in the last statements arise because drawOval expects ints while the sine and cosine produce doubles. Thus we convert the double to a Long and then extract the int equivalent.

Finally we are at a point where we can make the circle move.

\begin{verbatim}
Color fg = g.getColor();
Color bg = Color.white;  // default background
g.drawOval(centreX, centreY, diameter, diameter);
for (int i = 0; i < 25; i++) {
    // draw the oval in the background colour
    g.setColor(bg);
    g.drawOval(centreX, centreY, diameter, diameter);
    // and redraw it somewhere else
    centreX = centreX + deltaX;
    centreY = centreY + deltaY;
    g.setColor(fg);
    g.drawOval(centreX, centreY, diameter, diameter);
}
\end{verbatim}

These statements determine the colours to use, and draw the first circle. Then they redraw it using the background colour (causing the first circle to disappear), calculate the new location of the circle and draw it there, repeating as many times as necessary.

When you run the applet, you may not see the circle move, unless you have a very slow computer. You may see it only at its last position. How do you slow this action down?

One solution is to use the Timer class defined in javax.swing.Timer. Explore that class. Alternatively, you create a for loop that does nothing, but does it for a long time. The time to increment and check the counter will provide a delay.

The circle should appear to vanish as it reaches the edge of the frame. But that happens automatically!
4. Create an applet which generates a small circle in the graphics context. Its centre is specified via parameters. Make the circle move along a path whose angle you specify through a parameter. The circle should bounce off the edge of the frame and continue on its path.

This exercise will use many of the ideas from the previous exercise, plus you will need to detect collisions between the circle (think of it as a ball) and the edge of the frame (think of it as a wall.)

If the edge of the frame is smooth (and it is) the angle at which the circle approaches the edge of the frame is the supplement of the angle at which it bounces off. Two angles are supplements if they total 180 degrees or $\pi$ radians.

Assume that you move the ball by deltaX and deltaY, as in the previous exercise. Then, hitting a vertical wall means changing the sign of deltaX while leaving deltaY unchanged. Hitting a horizontal wall means changing the sign of deltaY while leaving deltaX unchanged.

5. Create an applet which generates a triangle in the centre of the frame.

Rotate the triangle. This will use much of what you have seen in the previous two questions.

6. Create an applet which creates the flag of your country. Some flags are harder than others due to the curved shapes they contain or the images on the flags (the Welsh dragon, or the penguin on the flag of the British Antarctic Territory, for instance.)

Illustrations of flags are available in many places. I like to use the World Flag Database, at www.flags.net; you’ll quickly find some of these difficult-to-draw flags. For example, Afghanistan and Albania are probably impossible to create using Java. Some international organizations, for example the African Union, are also difficult to create. But the Olympic Movement flag should be easy to create.

Other easy-to-create flags are Aruba, Bangladesh, Burkina Faso, Libya (probably the easiest of all), Ukraine, and United Arab Emirates.

Create a method that allows you to draw a star. Then you can consider drawing the flags of the United States (easier) and Micronesia (harder, because the stars do not all point straight up.)

When I ask my students to draw flags, we usually draw the flag of the United Kingdom and then use it as part of the flag of other nations in the Commonwealth. See Australia and New Zealand, for example. They need a method to draw stars too.

Assume we are drawing a star having $n$ points. One of the points is pointing directly up. The distance from the centre of the star to the outer points is $r_1$. The distance from the
centre of the star to the inner corner is $r2$. The star will be centred at a point $(x, y)$ in the applet's co-ordinate system.

Thus the method to draw the star has the possible signature

```java
public void drawStar(int n, double r1, double r2, int x, int y)
```

The mathematics behind the method is simple trigonometry.

Consider a right-angled triangle. It has a hypotenuse and two other sides which meet at a right angle. The trigonometric functions sine and cosine of an angle are defined as the ratio of the length of these sides (sine uses the opposite side, cosine uses the adjacent side) to the length of the hypotenuse.

That is, the sine of angle $A$ is the ratio opposite/hypotenuse. The cosine of angle $A$ is the ratio adjacent/hypotenuse.

The connection between this and stars is that we are interested in finding the $x$ and $y$ co-ordinates of the points of the star. Imagine a right triangle with one vertex at $(0, 0)$, the sides parallel to the $x$- and $y$-axes, and the hypotenuse being $r1$ (or $r2$). From the geometry of the star, we can calculate the angle $A$. Thus we can work backwards and determine the $x$ and $y$ co-ordinates of the point on the star.

The Java Graphics class has methods for drawing a polyline and a polygon. A polyline is a collection of connected line segments, described by an array of $x$ values and an array of $y$ values. A polygon is a closed polyline.

To draw a star we will use `drawPolygon`, providing it the appropriate arrays.
A circle is 360 degrees, or $2\pi$ radians, measured counter-clockwise. Since the star has $n$ points, they must be separated by $360/n$ degrees or $2\pi/n$ radians.

The Math class has sine and cosine methods which work in radians so we will use radians for our angles.

If one point of the star is pointing straight up, then its angle is $\pi/2$. Its neighbour is at an angle of $\pi/2 + \pi/n$. More correctly, its neighbour is at $\pi/2 + 2\pi/n$ but the twos in the second part of the expression cancel each other out.

The other angles can be calculated from these starting points by adding $2\pi/n$ inside a loop.

Important note – Graphics has (0, 0) in top left, x increasing to the right, y increasing down. Mathematics has (0, 0) in the centre, with x increasing to the right and y increasing up.

Hence, relative to the centre of the star, the first x and y co-ordinates are given as

```java
polyX[0] = (int) d1 * Math.cos(angle1);
polyY[0] = (int) -d1 * Math.sin(angle1);
polyX[1] = (int) d2 * Math.cos(angle2);
polyY[1] = (int) -d2 * Math.sin(angle2);
```

The negative signs are a reflection of the difference in direction of increasing y.

Relative to the co-ordinate systems of the applet, the co-ordinates are

```java
polyX[0] += x;
polyY[0] += y;
polyX[1] += x;
polyY[1] += y;
```

The angles are then changed, using the constant Math.PI instead of the Greek symbol for pi, $\pi$.

```java
angle1 = angle1 + 2 * Math.PI / n;
angle2 = angle2 + 2 * Math.PI / n;
```

But we wish to place these calculations inside a loop, as the number of points on the star (or iterations of the loop) may vary.

This gives rise to the code which generates the arrays.

```java
// the angular separation between the points at // distance r1 (and between the points at distance r2) double separation = 2 * Math.PI / n;
// the angle for the first point at distance d1, straight up double angle1 = Math.PI / 2;
```
// the angle for the point at distance d2
double angle2 = angle1 + separation / 2;

// the arrays of co-ordinates
int[] polyX = new int[n * 2];
int[] polyY = new int[n * 2];

// the element in polyX and poly which is being calculated
int j = 0;
for (int i = 0; i < n; i++) {

    /*
    * Note the negative sign in the calculation of the y co-ordinate.
    * This is because Graphics measures y from top to bottom, but mathematics
    * measures it from bottom to top.
    */
    polyX[j] = (int) r1 * Math.cos(angle1);
    polyY[j] = (int) -r1 * Math.sin(angle1);
    j++;
    polyX[j] = (int) r2 * Math.cos(angle2);
    polyY[j] = (int) -r2 * Math.sin(angle2);
    j++;
    angle1 = angle1 + separation;
    angle2 = angle2 + separation;
}

Now we have the co-ordinates relative to the centre of the star.

A second loop will translate these co-ordinates to the applet's co-ordinate system.

    for (int i = 0; i < 2 * n; i++) {
        polyX[i] += x;
        polyY[i] += y;
    }

Once the arrays are calculated, we need to draw the polygon in the Graphics area provided. But that Graphics area is not available to the method; it must be. Hence the method signature needs to be changed.

public void drawStar(int n, double r1, double r2, int x, int y, Graphics g) {

With that change, the drawStar method can include the commented statement

    // and draw the polygon
    g.drawPolygon(polyX, polyY, n * 2);

Recall that all our stars are upright; that is, they have a point that points to the top of the screen. If you wish to rotate stars, you should examine http://en.wikipedia.org/wiki/Transformation_matrix for the mathematics necessary.
Allowing for rotations requires us to modify the signature of the method as follows.

```java
* @param angle an optional parameter which specifies the angle (in radians) by which
  * to rotate the star. The angle is measured increasing in a clockwise direction, the opposite
  * of the math used to derive the co-ordinates of the star.
*)
public void drawStar(int n, double r1, double r2, int x, int y, Graphics g, double... angle)
```

The ellipsis indicate an optional parameter or parameters. These optional values are provided in an array of the type indicated.

Using the angle requires us to insert some code to do the rotation. This code must appear after we have created the polyX and polyY arrays, but before we translate the co-ordinates.

```java
// rotate the star if necessary. use the logic from
if (angle.length > 0) {
  // a rotation angle was specified so retrieve its value. This code assumes the angle
  // was provided in radians.
  double theta = angle[0];
  double s = Math.sin(theta);
  double c = Math.cos(theta);
  for (int i = 0; i < 2 * n; i++) {
    // calculate the new co-ordinates
    double xNew = polyX[i] * c - polyY[i] * s;
    double yNew = polyX[i] * s + polyY[i] * c;
    // save the new co-ordinates
    polyX[i] = (int) xNew;
    polyY[i] = (int) yNew;
  }
}
```

A star of the same size may appear several times on a flag. As examples, consider the flags of the United States of America and the European Union, or the many flags of nations in the South Pacific whose flags include the stars of the Southern Cross. It would be more efficient if we were to create the co-ordinates of the star once and then repeatedly display the star, but in different locations. To do that we need to consider a few changes.

- **drawStar** should not actually draw the star; it should simply create the arrays of co-ordinates and return them, so the method should be renamed **createStar**. How does a method return two arrays? It can't but it could return an array of **Point2D.Double** objects. Explore what a **Point2D.Double** object represents.

- The rotation of co-ordinates should be done in a separate method, so different copies of the star could be rotated by different amounts. Consider the flags of Venezuela and Micronesia.

- The translation of co-ordinates should be done in a separate method.
• Rather than using `drawPolygon`, we should use `fillPolygon`.

7. Investigate the `getProperties` method with the `System` class. How do you determine the operating system of the computer running your applet?
Chapter 20 – Design Patterns

Learning objectives

By the end of the chapter you will be able to:

- Define design pattern
- Use design patterns to help you design classes
- Use design patterns to help you understand the design of existing classes

Introduction

This chapter is not about programming. It is about the ideas that you should use to create a design before you begin programming.

In his book on Design Patterns, Steven John Metsker states “A pattern is a way of doing something, or a way of pursuing an intent. This idea applies to cooking, making fireworks, developing software, and to any other craft. In any craft that is mature or that is starting to mature, you can find common, effective methods for achieving aims and solving problems in various contexts. The community of people who practice a craft usually invent jargon that helps them talk about their craft. This jargon often refers to patterns, or standardized ways of achieving certain aims. Writers document these patterns, helping to standardize the jargon. Writers also ensure that the accumulated wisdom of a craft is available to future generations of practitioners.”

Among the crafts in which patterns are used are sewing, knitting, and woodworking. The name may be pattern, or recipe, or blueprint, or jig, or template.

The idea of patterns in computer science comes from the world of architecture, popularized by Christopher Alexander.

An architectural example is the pattern called Capturing Light. Many people like to have a quiet place to have their morning coffee. Often they prefer to sit in the sunlight, so the quiet place should be on the east side of the building, since the sun rises in the east. Look at the house described at http://www.architectureweek.com/2003/0423/patterns.html and, you can see nine additional architectural patterns at http://www.architectureweek.com/topics/patterns.html. They have names like Inhabiting the Site, Sheltering Roof, and Parts in Proportion.

The word pattern became known in computing with the publication of Design Patterns: Elements of Reusable Object-Oriented Software in 1995. The authors are Erich Gamma, Richard
Helm, Ralph Johnson and the late John Vlissides; collectively they are known as the “Gang of Four” or simply “GoF”).

They define a pattern as having four essential elements. “The pattern name is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. … The problem describes when to apply the pattern. … The solution describes the elements that make up the design, their relationships, responsibilities, and collaborations. … The consequences are the results and trade-offs of applying the pattern.”

We will see some of the GoF patterns below, but we will also see patterns from other sources.

Regardless of the field, a pattern is a named solution to a problem in design, one which has been used by many people to solve a problem. In architecture, we have Capturing Light; in software engineering we have Singleton.

Metsker’s paragraph which I quoted appears near the beginning of his book, Design Patterns Java Workbook. In his book he shows how to apply the design patterns first publicized in the GoF book.

In software engineering, a pattern is not a piece of Java code; that would be an implementation artefact. The GoF book describes many patterns; Metsker’s book shows how to implement the patterns in Java. Throughout this book we have been using patterns without usually naming them. It’s now time to do so.

Wikipedia provides details on the history of patterns within computer science, and lists many of the patterns that have been published, at http://en.wikipedia.org/wiki/Design_pattern. The article begins with the words “In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.”

A few of these patterns are independent of the others, but many work with other patterns. This is clearly shown in a pattern map from the GoF book, illustrating the connections between the GoF patterns.
Interface

Here is a pattern that underlies much of the philosophy behind Java. This pattern is not directly named in the GoF book since the authors were not using Java examples; they used Smalltalk and C++ and those languages don’t support interfaces as Java does.

We have used many Java interfaces in our coding, including Comparable, Set and Map. The basic idea behind interfaces pattern is to describe some capabilities a class may have, capabilities which many classes may be expected to have, and then leave the classes to provide the details of how those capabilities are implemented.

For the Comparable interface, we saw that the capabilities it specifies are the ability to compare two objects of the same type. The details of how that is accomplished depend on the specific class.
The Student class implements Comparable. Thus, a Student object is also a Comparable object. If we have a method which, for some reason, requires a Comparable object to be used as an argument, a Student object may be used. But so too may other Comparable objects.

If a class implements several interfaces, then an object of that class has several types.


**Marker interface**

A specialization of the Interface pattern is the marker interface pattern.

With Marker, an object which implements a marker interface indicates that it will support the capabilities of the interface. Sometimes this capability will require us to write code, sometimes it will not. Serializable (where we may not implement code) and Cloneable (where we will implement code) are both marker interfaces we have used which involve this pattern.

More details on Marker are at http://en.wikipedia.org/wiki/Marker.

**Creator**

This pattern is a simple one. It helps you answer the question “Which class should create new objects?” As such it is a pattern that you use while designing your system. It is not a pattern you use in solving problems that arise while implementing the system.

For example, in our college model, we have students and courses. Which class should create a new Student object? Which should create a new Course object?

Creator suggests that since students are registered at the college, College should create new Student objects.

Courses are taught by professors. Does that mean Professor objects should create Course objects? No, a course is taught by a professor, but the course continues to be available after the professor has left the college. Thus, Creator suggests College should create Course objects.

The Creator pattern was publicized by Larman as part of his General Responsibility Assignment Software Patterns, GRASP.

**High Cohesion**

This is another of Larman’s patterns, and gives a name to a practice used for many years. Many would note that High Cohesion is a design principle and not a pattern. A design principle is “less important” than a pattern. No, that is not quite right. A design principle is “less sophisticated” than a pattern.
The problem it is solving is how to write high quality code, code which can be understood by a single programmer.

Something which gets in the way of understanding is complicated code, code which attempts to do too much.

This pattern (or design principle) states that a class should do a limited number of things and it should do them well.

It also applies to methods. The `showFrame` methods we have seen that build a data entry screen and show it are not cohesive since they do two different tasks. The High Cohesion pattern suggests the use of two methods, one to build the data entry screen, and the other to display it.

**Low Coupling**

This too is another of Larman’s patterns, and gives a name to a practice used for many years. Many would note that Low Coupling is a design principle and not a pattern.

The problem it is solving is how to minimize the connections between classes, so that a change in one class has minimal effects on another.

This pattern (or design principle) states there should be no more connection between classes than is necessary, and that those connections be as simple as possible. Note that the Checkstyle extension interprets Low Coupling to mean that a method should have no more than seven parameters.

This pattern is related to the GoF Mediator pattern.

**Observer or Event listener**

The GoF defines the problem Obserer solves as “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.”

This is another pattern which is clearly reflected in the philosophy of Java, particularly in the `ActionListeners` and `InputVerifiers` we used while building our data entry screens.

An object listens for events and acts on them. To do this, it must identify itself to some other object which also listens and lets other objects know of the events.

This has also been called a “publish and subscribe” pattern. One object publishes, or sends, messages about events, while others (the listeners) subscribe to receive those messages.

This pattern is the basis behind the success of Twitter and Facebook.

**Delegation**

This is another pattern which is clearly reflected in the philosophy of Java; objects appear to possess behaviours but delegate those behaviours to other objects.

We saw Delegation most clearly when using the XMLEncoder; we replaced the DefaultPersistenceDelegate with one which knew about the collections our object contained.


**Factory**

Like delegation, Factory is a pattern which is clearly reflected in the philosophy of Java. Not all classes have the constructors you would expect. In some cases they have methods which act as constructors. More correctly, the methods are factories, which build objects.

We have seen Factory with the Integer.valueOf method, the various BorderFactory.create methods, and with BindingFactory.


**Lazy Initialization**

Perhaps Lazy Initialization is a design principle rather than a pattern but it deserves mention.

Suppose you have a program which involves many data entry screens. One school of thought says you should create all the screens (or at least the most-commonly used ones) before the program displays its main menu. Simply display a splash screen and a “waiting” icon until all the screens are ready to use. That could involve a lengthy wait as the program loads.

Or suppose you have a program that takes some time to create an index for a document (or some other large, complicated data structure.) Some programs would create the index while you are typing the document, slowing down the program. Other programs would create the index only when requested.

Both of these could be improved by paying attention to the lazy initialization pattern. This pattern says don’t create something until it’s needed. When it is needed, perhaps you could use the factory pattern to decide how it is created.

Note that Lazy Initialization is one of the principles behind eXtreme Programming.

Dynamic programming, which we will use with some of the mathematical functions in the next chapter, is another use of the lazy initialization pattern.

**Singleton**

The GoF book defines the problem Singleton solves as “Ensure a class only has one instance, and provide a global point of access to it.”

We used Singleton when we created the College class, a class which exists to ensure there is only one set of rules which applies to all students. For our college, the marks for a student in a course are stored as a percentage. But sometimes, you want to display them as a letter grade. The college needs to have one set of rules for translating percentages to letter grades, independent of the student, the course, or the department.

We would need a class which creates a method to translate these percentages into letter grades. But we wish only one such method. Thus the constructor would need to see it there already was such a method created.

A singleton College also ensures there is only one Student object for each student, one Professor object for each professor, one Course object for each course, and one Section object for each section.

Similarly, were we to examine bank accounts as an example of objects, we would find that the Singleton pattern helps us keep the interest rates paid in only one place. There would be only one rule to determine the interest paid on a bank account.

As an aside, many older programming languages used global variables, variables which were known and changeable in all parts of a system. This allowed for all sorts of problems when values were accidentally changed. The Singleton pattern allows the retrieval of global values, but it can limit the changing of those values. As such, it is a more powerful technique than using global variables.

It also allows changing values on the fly, by reading something in from a resource bundle, without recompiling the system.


**Model-view-controller (MVC)**

This is quite an old pattern, dating back to the days when Smalltalk, one of the first object-oriented languages, was new. Of course, the term “pattern” was not used then.

MVC is the pattern we have tried to use in creating the GUI for the College.

The model represents the data and how it is stored. Note that we have looked at several candidates for the model, including serialization and XML. We have also mentioned the use of databases.
The view represents the interface we have created to manipulate and present the data.

The controllers are the glue that connects the model and the view.

Controllers respond to user events and modify the model. By using controllers, the interface is isolated from the details of the model implementation. If we write the controllers correctly, they too are ignorant of the model implementation.

For example, the model could have a writeData method which the controller could invoke. That method would know whether XML or a database is being used. This is also an example of the application of the Low Coupling pattern.

A famous example of MVC divides the screen into four quadrants. In the first you enter data into the model. In the second, that data is displayed as a histogram. In the third, it is displayed as a pie chart. In the fourth it is displayed as a table of data. I wish I could find it! Perhaps it is apocryphal.

More details on MVC are at http://en.wikipedia.org/wiki/Model

**Summary**

Patterns are a very important part of design. This chapter has provided a very brief introduction to patterns. We are not able to go into them further as this book, while it touches on design, is mainly dealing with implementation.

If you wish to look into patterns in more detail (and you may do so in further courses), good places to start online are http://en.wikipedia.org/wiki/Design_Patterns and http://c2.com/cgi/wiki?DesignPatternsBook, and the exercises on the next page.
Exercises

1. Look in the GoF book and explore the patterns they discuss. Their catalogue contains 23 patterns with which you should become familiar. These include Abstract Factory, Adapter, Bridge, Builder, Chain of Responsibility, Command, Composite, Decorator, Façade, Factory Method, Flyweight, Interpreter, Iterator, Mediator, Memento, Observer, Prototype, Proxy, Singleton, State, Strategy, Template Method, and Visitor. You will become a better designer and programmer when you gain familiarity with these patterns. Don’t attempt to become familiar with all the patterns at once.

Note that the GoF book is available on CD as well as in printed format. For learning the patterns, the CD is perhaps better.

2. Look in Metsker’s book and see how the GoF pattern may be implemented in Java.

I suggest you use Metsker’s book in combination with the GoF book. One provides the theoretical underpinnings for the pattern, and the other provides a Java implementation.

3. Look in Larman’s book and explore the patterns he discusses. These include Information Expert, Creator, Controller, Low Coupling, High Cohesion, Polymorphism, Pure Fabrication, Indirection, and Protected Variations.


Explore the links they provide.
Chapter 21 – Mathematical examples

Learning objectives

By the end of the chapter you will be able to:

- Define rational number
- Define complex number
- Use Java to create classes which were omitted from the Java libraries

Introduction

The classes we have created in earlier chapters were all related to computer representations of things in real life. Sometimes a class can represent mathematical concepts instead.

I’d like to look at some mathematical examples now, one of which is available as part of a Java package, and two of which we can create ourselves.

BigInteger

What is the largest integer?

Since the time of the Greeks it has been known there is no largest integer. If there were, what would that integer plus one be?

Unfortunately computers don’t work well with large numbers whose size has no limits. We have already seen the limits Java imposes on the size of integers. A byte contains 8 bits (which could represent integers from -128 to 127), a short contains 16 bits (which could represent integers from -32768 to 32767), an int contains 32 bits (which could represent integers from -2147483648 to 2147483647), and a long contains 64 bits (which could represent integers from $-2^{63}$ to $2^{63} - 1$).

Suppose you need numbers larger than that. The national debt for a large country may exceed even the limit for a long. Certainly the gross national product for many countries will exceed the limit for a long.

Public key encryption using the RSA algorithm involves factoring large numbers. In early 2007, researchers reporting factoring a 307-digit number into its two prime factors, one an 80-digit number, the other a 227-digit number. It took several supercomputers about 11 months to accomplish the factorization.
So how do we represent even larger numbers?

The BigInteger class comes to the rescue. The online Java documentation says “BigInteger provides analogues to all of Java's primitive integer operators, and all relevant methods from java.lang.Math. Additionally, BigInteger provides operations for modular arithmetic, GCD calculation, primality testing, prime generation, bit manipulation, and a few other miscellaneous operations.”

To understand that statement, we should look at java.lang.Math first.

**The java.lang.Math package**

The online Java documentation says “The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.”

Of course method is the term we use in Java, and function is the term we use in mathematics.

We have used a few of these mathematical functions in our calculations so far, particularly as we drew stars in our applets. But Java is used in the world of science and engineering so many people need these mathematical functions.

The absolute value method (abs) strips the signs from numbers, converting negative numbers to positive ones.

Logarithm and exponential functions are available.

So are trigonometric and inverse trigonometric functions. Of course, you may convert degrees (a circle represents 360 degrees) to radians (a circle may also be measured as $2\pi$ radians), and vice versa.

Maximum and minimum functions are available. The functions all have only two numeric parameters. How would you write a method to find the maximum of three numbers?

How about the following solution?

```java
public double max(double a, double b, double c) {
    return Math.max(a, Math.max(b, c));
}
```

A square root function is available for non-negative numbers.

All of these functions are available for floats and doubles, and, in many cases, ints and longs. shorts and bytes are generally not welcome in the Math class.

The Math class also provides values for the mathematical constants $e$ (Math.E) and $\pi$ (Math.PI).
Math.PI has only a few digits. In the fall of 2010, a team computed π to the two-quadrillionth bit. More details are available at http://arxiv.org/abs/1008.3171.

Finally, the Math class provides a way to generate pseudo-random numbers, ones which meet tests of randomness but are actually created non-randomly. For details of how pseudo-random numbers are generated, please consult any statistics textbook.

**BigInteger, continued**

The GCD calculation referred to above is the greatest common divisor calculation, a nice recursive method which we examine in more detail in the next chapter.

When you look through the methods available in the BigInteger class you see their variety.

When I last checked (2010-05-21, via http://www.brillig.com/debt), the USA national debt it was $12,982,626,754,668.02. You would need a BigDecimal to represent the debt exactly, but we can round it to the BigInteger value of $12,982,626,754,668. To test BigInteger, I created an applet. In it I placed the following statements.

```java
import java.math.BigInteger;

private BigInteger usDebt = new BigInteger("12982626754668");

g.setColor(Color.red);
g.drawString("US debt " + usDebt.toString(), 20, 40);
```

The Canadian national debt (as reported via a Java applet at http://www.ndir.com/SI/education/debt.shtml on the same date) was approximately $702,564,000,000. While this too is a large number, it is not quite comparable to the American figure as the Canadian number includes federal, provincial, and municipal debt, while the American number includes only the federal debt.

I am using these websites as a source of large numbers. I am neither agreeing with nor disagreeing their points of view.

Between first writing these words and editing them three years later, the American debt rose by approximately $3,000,000,000,000 while the Canadian dropped by approximately $35,000,000,000. Of course, a recession happened during the last of those three years.

By May 2012, the corresponding numbers were approximately $15,727,115,000,000 and $584,685,470,000.

Note that the constructor for a BigInteger expects the number to be passed to it as a String, one without any commas or any other punctuation signs. No dollar or other currency signs are allowed. If you wish to create a BigInteger from a value already stored in an int or a long, you will
find there is no constructor which accepts an int or a long. Instead, you must use the \texttt{valueOf} method.

\begin{verbatim}
int n = 40;
BigInteger b = BigInteger.valueOf(n);
\end{verbatim}

\texttt{valueOf} is a static method within the \texttt{BigInteger} class, the same way \texttt{abs} is a static method within the \texttt{Math} class. In both cases, that means we use the name of the class, followed by a period and the method. This is an example of the Factory pattern.

The \texttt{BigDecimal} class, on the other hand, has the constructors you would expect. For example, it has a constructor which accepts a double as its argument.

\section*{Definitions – class and object - revisited}

Recall that our definition of a class is “A class is a model of something in the real world or it is a way of implementing a concept.”

The \texttt{Math} class is neither. It represents a collection of methods and constants. All of its methods are static. Its two constants are static. There is no constructor, no getter, no setter, and no \texttt{toString} method.

Thus we need to expand our definition of a class still further. “A class is a model of something in the real world, or it is a way of implementing a concept, or it is a collection of related methods and constants.”

\section*{Rational numbers}

There are two types of numbers which \texttt{java.lang.Math} does not support. First we’ll look at the rational numbers. Then we’ll look at complex numbers.

What is a rational number?

\begin{quote}
A rational number is any number which can be expressed as the ratio of two integers. \( \frac{3}{4} \) is a rational number, as is \( \frac{36}{48} \). In fact, those two rational numbers are the same. The numerator (36) may be written as \( 12 \times 3 \). The denominator (48) may be represented as \( 12 \times 4 \). After you remove the common factor (12) from the numerator and denominator, you are left with \( \frac{3}{4} \).
\end{quote}

Note that there are some numbers which are not rational numbers. These are called irrational numbers. Perhaps the simplest irrational number is the square root of 2.

For several proofs of that) see \url{http://en.wikipedia.org/wiki/Square_root_of_2}.

Some rules we will adopt for our rational numbers.

\begin{itemize}
\item Common factors will be removed from the numerator and the denominator.
\item A zero denominator is not allowed.
\end{itemize}
A negative rational number will have its numerator negative and its denominator positive.
A rational number with both numerator and denominator negative will be replaced by and
equivalent one with both denominator and numerator positive.
Zero is represented as 0/1.

So let’s create a rational number class. Yes, I know that you can find many examples of a
RationalNumber class on the Internet. This is a standard programming example in many
introductory programming courses.

Following my usual practice, we begin with a constructor and a toString method.

```java
/**
 * class RationalNumber.
 *
 * @author rick gee
 * @version may 2006
 */
public class RationalNumber {
    // instance variables
    private int n;  // the numerator
    private int d;  // the denominator

    /**
     * Constructor for objects of class RationalNumber.
     *
     * @param n numerator
     * @param d denominator
     */
    public RationalNumber(int n, int d) throws NumberFormatException {
        int num = n;
        int denom = d;
        if (d == 0) {
            throw new NumberFormatException("RationalNumber: denominator may not be zero");
        }
        if (num == 0) {
            denom = 1;
        } else {
            int divisor = gcd(Math.abs(num), Math.abs(denom));
            num = num / divisor;
            denom = denom / divisor;
        }
        // initialize instance variables
    }

    // methods
}
```

this.n = num;
this.d = denom;
}

/**
 * @return a String representation of a rational number
 */
public String toString() {
    return n + "/" + d;
}

/**
 * compute the greatest common divisor of the two
 * positive parameters. Uses Euclid's algorithm.
 */
private int gcd (int num1, int num2) {
    while (num1 != num2) {
        if (num1 > num2)
            num1 = num1 - num2;
        else
            num2 = num2 - num1;
    }
    return num1;
}
}

We’ll look at the gcd method in more detail later. For now, trust me that it works.

To test the methods in the class, write unit tests. You’ll need quite a few since the constructor makes many decisions. Consider the following values for the numerator and denominator; the constructor should work properly for each.

- 0 and 5 (result should be 0 and 1)
- 0 and 0 (result should be an exception)
- 0 and -3 (result should be 0 and 1)
- 1 and 7 (result should be 1 and 7)
- -1 and -8 (result should be 1 and 8)
- 2 and -9 (result should be -2 and 9)
- 2 and 8 (result should be 1 and 4).

Have we missed any combinations? Yes. -5 and 3, -2 and 0, 8 and 0 immediately come to mind.

**Adding rational numbers**

How do you add together two rational numbers?

The mathematical definition
\[
\frac{a}{b} + \frac{c}{d} = \frac{a \cdot d + b \cdot c}{b \cdot d}
\]

translates into

/**
 * Add two rational numbers.
 * @param r The second rational number
 * @return this + r as the new value of this
 */
public void add(final RationalNumber r) {
    int a = this.getNumerator();
    int b = this.getDenominator();
    int c = r.getNumerator();
    // cannot use d for r.getDenominator() as that is the name of the instance variable
    int d1 = r.getDenominator();

    // change the values of the instance variables
    RationalNumber tempR = new RationalNumber(a * d1 + b * c, b * d1);
    n = tempR.getNumerator();
    d1 = tempR.getDenominator();
}

Notice that we need two getter methods, one for the numerator and one for the denominator.

Notice also that by using the constructor, the add method does not need to concern itself with removing common factors at the end, nor does it need to concern itself with negative signs.

**Subtracting, multiplying, and dividing rational numbers**

We leave it as an exercise to implement subtraction

\[
\frac{a}{b} - \frac{c}{d} = \frac{a \cdot d - b \cdot c}{b \cdot d}
\]

multiplication

\[
\frac{a}{b} \cdot \frac{c}{d} = \frac{a \cdot c}{b \cdot d}
\]

and division.

\[
\frac{a}{b} \div \frac{c}{d} = \frac{a \cdot d}{b \cdot c}
\]
What should happen when you attempt to divide by the rational number 0/1? Does it?

**Inverting a rational number**

Implement a method to invert (or calculate the reciprocal of) a `RationalNumber`. That is, if you have a rational number `a/b`, the reciprocal is `b/a`.

What should happen if you try to invert 0/1? Does it?

**Converting a rational number to a double**

Implement a method to convert a `RationalNumber` to its equivalent double value.

**What if ints are not big enough?**

In the `RationalNumber` class, we have assumed the numerator and denominator are both ints. If necessary, we could always change them to longs, without disturbing the code of anyone who uses the class. Being able to make changes like this without disturbing code which uses our code is a result of a design principle called information hiding. It is also due to the use of the design pattern Loose Coupling.

What changes would you need to make to the `RationalNumber` class to use `BigInteger`, while at the same time inconveniencing users of this class as little as possible?

**Complex numbers**

Another type of numbers that `java.lang.Math` does not handle is complex numbers.

Recall that a complex number has two parts, the real part and the imaginary part. For example, 6 + 4i is a complex number, as are -5 + 5i, 0 + 0i, and 2 – 7i.

The letter i represents the square root of -1, a number which in some sense does not exist, hence is called an imaginary number.

Complex numbers are usually displayed as a String containing the real part, the sign of the imaginary part, the absolute value of the imaginary part, and the letter i. But in some mathematics an imaginary number can be very useful.

Complex numbers sometimes are represented as (6, 4), (-5, 5), (0, 0), and (2, -7). Thus we have two choices for the implementation of the `toString` method.

**Arithmetic operations and complex numbers**

Various operations are defined on complex numbers.

\[(a + bi) + (c + di) = (a + c) + (b + d)i\]
\[(a + bi) - (c + di) = (a - c) + (b - d)i\]

\[(a + bi) * (c + di) = (a * c - b * d) + (b * c + a * d)i\]

Division is a little more complicated. The real part of \((a + bi) / (c + di)\) is \((a * c + b * d) / (c^2 + d^2)\) and the imaginary part is \((b * c - a * d) / (c^2 + d^2)\)

**Implementing ComplexNumber**

Make it so.

Note that while my first examples show integer values for both the real and imaginary parts, they should actually be doubles.

**Summary**

Java is useful for many types of applications. One of its strengths is the ease with which you may add libraries, thus extending the areas in which it is applicable.

We have seen its versatility by creating `RationalNumber` and `ComplexNumber` classes.
Exercises

1. Complete the RationalNumber class.
   Change the datatype for its instance variables to long. Describe the effects.
   Change the datatype to BigInteger. Describe the effects.

2. Complete the ComplexNumber class.

3. Complex numbers may be represented by x and y as we have done, but they may also be represented as an angle and a radius. The angle may be calculated as the inverse tangent of y (the complex part) over x (the real part), and the radius is the square root of x squared plus y squared.
   Supplement the ComplexNumber class to include accessors for the angle and the radius. That is, add two new getters, one for radius and one for angle.
   You cannot write a ComplexNumber class that has two constructors, one which accepts x and y, and one which accepts the angle and the radius. Why not?

4. Just as complex numbers are the extension of real numbers from 1-space into 2-space, quaternions are the extension of complex numbers into 4-space.
   Learn something about quaternions and implement a Quaternion class.

5. Just as complex numbers are the extension of real numbers from 1-space into 2-space and quaternions are the extension of complex numbers from 2-space into 4-space, octonions are the extension of quaternions from 4-space into 8-space.
   Learn something about octonions and implement an Octonion class.
Chapter 22 – Mathematical methods

Learning objectives

By the end of the chapter, you will be able to:

- Define algorithm
- Write methods to implement several mathematical algorithms
- Define iteration.
- Define recursion
- Write iterative methods
- Write recursive methods
- Write methods that use formulas.
- Use dynamic programming

Introduction

Sometimes a class can be a collection of useful constants or methods. Imagine a class that contained the various constants used in physics and chemistry; the speed of light, Planck constant (Wikipedia - a physical constant reflecting the sizes of energy quanta in quantum mechanics), Avogadro constant (the number of atoms/molecules in a standard amount of a substance), etc.

Imagine a class that contained information on all the chemical elements.

But I’d like to look at a collection of simple algorithms and the methods which implement those algorithms. Let’s create a class called Algorithms. (Yes, I know that once upon a time I said the name of a class should be a singular noun. If that really bothers you, use the name AlgorithmCollection.)

What is an algorithm?

An algorithm is a set of rules that you can follow to solve a problem. An algorithm will solve the problem in a finite number of steps and it will solve the problem in a finite amount of time. Frequently there are several algorithms which solve the same problem. This is similar to the way we have seen different Java statements to accomplish the same result.
“Analysis of Algorithms” is the study of algorithms (understanding which are better and when) and the development of new algorithms. It is often a course taken in the second half of your post-secondary (tertiary) study.

**Greatest common divisor (gcd)**

While creating the `RationalNumber` class, we needed a method to find the greatest common divisor of two numbers. The code we used is repeated below (but I have changed the name of the method. You’ll see why in a moment.)

```
/**
 * compute the greatest common divisor of the two positive parameters. Uses Euclid's algorithm.
 */
private int gcdIterative (int num1, int num2)
{
    while (num1 != num2) {
        if (num1 > num2)
            num1 = num1 - num2;
        else
            num2 = num2 - num1;
    }
    return num1;
}
```

This algorithm is the Euclidean (or Euclid’s) algorithm. Euclid was a Greek mathematician and the algorithm is over 2300 years old; it is described in [en.wikipedia.org/wiki/Euclidean_algorithm](http://en.wikipedia.org/wiki/Euclidean_algorithm).

This method implements an iterative solution to the problem. An iterative solution is one in which you perform one step after another, under the control of a loop of some kind, here a while loop, until you reach your solution.

Note something interesting with the method. The two parameters are not declared as final. We declare objects final when passed as parameters to prevent inadvertent changes to them, but primitive datatypes are handled differently. When you pass a primitive datatype into a method, a copy is made of it, and that copy is manipulated.

Explore what happens if you make `num1` and `num2` final in the method header.

**A recursive greatest common divisor method**

However, there is also a recursive solution to the problem of calculating the greatest common divisor as well.

In a recursive solution, a method develops the solution by making a smaller case of the problem and calling itself again to solve the smaller case. This continues until a **smallest** case is reached in which you know the solution. All recursive methods have these two parts:
• One or more (smallest) stopping case(s) for which we know the answer, and
• A statement that calculates an answer by calling the method with a smaller argument.

We saw a recursive method in an earlier chapter, to reverse a String.

Here is a recursive solution to the gcd problem.

```java
public static int gcdRecursive(int num1, int num2) {
    if (num2 == 0)
        return num1;
    return gcdRecursive(num2, num1 % num2);
}
```

Remember that the modulus operator (%) calculates the remainder when you divide the first integer argument by the second. For example, 11 % 2 is 1, since 2 goes into 11 five times, with a remainder of one.

**Sample greatest common divisor calculations**

Perhaps we should work out an example to see how the iterative and recursive methods work.

Wikipedia uses 1071 and 1029 as the two numbers for which we wish to find the greatest common divisor.

For the iterative solution,

• We start with num1 = 1071 and num2 = 1029. The numbers are not the same, num1 is the larger, and so num1 becomes the difference, 42, while num2 remains 1029.
• num1 and num2 are not the same, but now num2 is the larger, so it becomes 1029 – 42 or 987 while num1 remains 42.
• num1 and num2 are not the same, and num2 is the larger, so it becomes 987 – 42 or 945 while num1 remains 42.
• This continues, removing 42 from num2 each time, until it becomes 63.
• num1 and num2 are not the same, and num2 is larger, so num2 becomes 63 – 42 or 21 while num1 remains 42.
• Now num1 is larger than num2, so becomes 42 – 21 or 21.
• Now both num1 and num2 are 21 so the algorithm terminates and the greatest common divisor is 21. In fact, 1071 = 3 * 17 * 21 and 1029 = 7 * 7 * 21 so 21 is the largest integer which divides both. Hence the name, greatest common divisor.

For the recursive solution,

• We start with num1 = 1071 and num2 = 1029.
- The second number is not zero, and 1071 % 1029 is the remainder when we divide 1071 by 1029, or 42. Before we can return an answer, we need to calculate gcdRecursive(1029, 42).
- The second number is not zero, and 1029 % 42 is 21. Before we can return an answer, we need to calculate gcdRecursive(42, 21).
- The second number is not zero, and 42 % 21 is zero. Before we can return an answer, we need to calculate gcdRecursive(21, 0);
- The second number is zero, so we return the first number, 21.
- Now all the waiting, unterminated, methods can return 21.

Both forms of the algorithm produce the same answer (That’s good!) but the second requires more memory, since it has a number of unterminated methods in memory at one time. The first may require more time, since it requires more steps.

Once you have the algorithm explained in a recursive form, it seems like magic the way the method works.

As we have already seen, declaring a method static says that the method belongs to the class. It is not a method which belongs to instances of the class. In combination with declaring the method public, other classes may call the method using the Algorithms.gcdRecursive(int, int).

Note that the iterative method was a private method. If we are going to use it in outside our class, it will need to become a public static method also.

Mathematicians tell us that the worst performance for the gcd function occurs when the parameters are consecutive Fibonacci numbers, which are described below.

**Fibonacci numbers**

If you read or saw The da Vinci Code, you have seen Fibonacci numbers. They were the first line of numbers written on the floor of the Louvre, functioning as the answer to the secret question if you forget your account number.

This series of numbers occurs in many places in the world, both natural and mathematical. Wikipedia has some interesting examples at [http://en.wikipedia.org/wiki/Fibonacci](http://en.wikipedia.org/wiki/Fibonacci), dating back 2200 years.

The series derives its name from Leonardo of Pisa. His father was nicknamed Bonaccio (“good natured” or “simple”). His son was posthumously given the nickname Fibonacci (derived from the Latin words *filius Bonacci*, meaning son of Bonaccio).
The values in the series are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, etc., where the first number (F₀) is zero, the second (F₁) is one, and every subsequent Fibonacci number is the sum of the previous two. That is, in functional notation:

\[ F(0) = 0; \]
\[ F(1) = 1; \]
\[ F(n) = F(n - 1) + F(n - 2) \text{ when } n > 1 \]

Sometimes, the series starts at 1, 1, 2. This is because Fibonacci was dealing with questions about the breeding behaviour of pairs of rabbits and it is difficult to breed more rabbits if you have no pairs. Of course, if you have one pair, you will soon have many. Fibonacci wanted to know how many pairs you would have, given certain assumptions about the age at which rabbits begin breeding and how frequently they breed.

**Iterative Fibonacci method**

Here is an iterative method to find the \( n \)th Fibonacci number.

```java
public static int iFib(final int n) {
    int f0 = 0;
    int f1 = 1;
    int t;
    for (int i = 0; i < n; i++) {
        t = f0 + f1;
        f0 = f1;
        f1 = t;
    }
    return f0;
}
```

We know that there is a limit to the size of ints so we would expect there is a limit to which Fibonacci numbers we can calculate. `iFib(46)` looks okay, but `iFib(47)` is negative. How can two positive numbers added together become negative?

It has to do with the way numbers are stored as bits. One of the bits is the sign bit. When the calculations in the other bits result into a carry into the sign bit, it may change from a positive number to a negative one. That’s what happens here. The result is obviously wrong.

Changing from `int` to `long` only delays the problem. How much does it delay the problem?
Recursive Fibonacci method

Here is a recursive method to find the n\textsuperscript{th} Fibonacci number.

```java
/**
 * recursive Fibonacci.
 * @param n The position of the number we wish
 * @return the nth Fibonacci number
 */
public static int rFib(final int n) {
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    return rFib(n-1) + rFib(n-2);
}
```

Recall that recursive method work by trying to solve successively smaller problems using the same method, and there are one or more “smallest” problems for which we know the answer. The definition of Fibonacci numbers provides a blueprint for a recursive implementation.

If you run the method, you’ll find that it works, but it gets very slow for larger values of n.

If we are going to use these Fibonacci methods for large values of n, we will need to go to BigIntegers, and we will have to do something about the speed of the recursive method. We’ll see how to do that later.

The Wikipedia article mentioned earlier also has some interesting generalizations of the Fibonacci series.

One that I find particularly interesting was explored by Divakar Viswanath. The random Fibonacci sequences are defined by:

\[
T(0) = 1; \\
T(1) = 1; \\
T(n) = \pm T(n - 1) \pm T(n - 2)
\]

The plus or minus signs are equally likely and are chosen independently, perhaps using a coin-toss. His article (available at [http://www.ams.org/mcom/2000-69-231/S0025-5718-99-01145-X/home.html](http://www.ams.org/mcom/2000-69-231/S0025-5718-99-01145-X/home.html)) points out that it is not clear what will happen to the numbers in the series, but he proves that they have an interesting behaviour.

Note that the Fibonacci numbers we discussed have an interesting, but different, behaviour themselves. When you calculate the ratio $F(n + 1) / F(n)$ for a number of values of n, you will find that the ratio comes closer and closer to the golden ratio ($\phi$, pronounce “feye” or “fee”) or approximately 1.618034.
In addition, it is possible to calculate $F(n)$ via a formula. $F(n)$ is the largest integer less than $(\phi^n/\sqrt{5}) + 0.5$.

### The factorial function

How many ways are there of arranging 10 people in a line?

The answer is $10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$ or 3628800. This is an example of the factorial function, written in mathematics as $n!$.

The iterative definition of factorial is:
- Zero factorial is defined, for convenience, as 1.
- For any other positive value $n$, $n$ factorial is defined as the product of all integers from 1 up to and including $n$.

The recursive definition of factorial is:
- Zero factorial is defined, for convenience, as 1.
- For any other positive value $n$, $n$ factorial is defined as $n \times (n - 1)$ factorial.

The first of these definitions (iterative) leads to the following implementation of the factorial function.

```java
/**
 * iterative factorial.
 * @param n  The value whose factorial we wish to calculate
 * @return n factorial
 */
public static int iFactorial (final int n) {
    if (n == 0) {
        return 1;
    }
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result = result * i;
    }
    return result;
}
```

The second definition (recursive) leads to the following implementation of the factorial function.

```java
/**
 * recursive factorial.
 * @param n  The value whose factorial we wish to calculate
 * @return n factorial
 */
public static int rFactorial(final int n) {
    if (n == 0) {
        return 1;
    }
    return n * rFactorial(n - 1);
}
```
Both of these calculate 16! successfully and then overflow when you attempt to calculate 17!. If we used long instead of int, how much further could we go?

**The choose function**

How many ways can you choose five volunteers from a group of 15 people? Assume that the order of the volunteers doesn’t matter.

The answer is 15 choose five, written \( \binom{15}{5} \). This unusual expression is evaluated as 15! / (10! * 5!).

**A Formula!**

In Java, the method is very simple to write. It is neither iterative nor recursive. It is simply a formula, but a formula that uses one of the factorial methods.

```java
/**
 * the number of ways to choose n items from m.
 * @param m the size of the pool
 * @param n the number to be chosen from the pool
 * @return m choose n
 */
public static int choose(final int m, final int n) {
    return iFactorial(m) / (iFactorial(n) * iFactorial(m - n));
}
```

Unfortunately it does not give correct answers in many cases, due to the overflow problems we have noticed earlier. How could we modify it to work properly?

I suppose we could use longs or BigIntegers, or we could use a more-intelligent algorithm, a better way to solve the problem. We’ll use a more-intelligent algorithm.

**Recursive choose method**

One such more-intelligent algorithm is a recursive one. Note that \( \binom{n}{k} \) is also defined as:

1. if \( k = 0 \), or \( n = 0 \), or \( n = k \), and \( \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \) otherwise.

See [http://en.wikipedia.org/wiki/Choose](http://en.wikipedia.org/wiki/Choose) for a discussion on this form of the algorithm, which gives rise to this recursive method.

```java
/**
 * recursive choose - the number of ways to choose k items from n.
 * @param n the size of the pool
 * @param k the number to be chosen from the pool
 * @return n choose k
 */
public static int rChoose(final int n, final int k) {
```
if (k == 0)  
    return 1;
if (n == 0)  
    return 1;
if (n == k)  
    return 1;
return rChoose(n - 1, k - 1) + rChoose(n - 1, k);
}

This method gives valid answers in many of the cases for which the previous version did not. Addition is less likely to overflow than multiplication since the numbers involved are smaller.

The Ackermann function

I find the Ackermann function a truly fascinating function. It has its origin in the design of operating systems and was designed specifically to produce problems while running programs.

**Warning** - Do not try to calculate the Ackermann function for large values of m and n. A(4,1) is too large, unless you have some time to wait and a very fast computer.


Look at the table on that web page to see why you should not try to calculate the Ackermann function for large values of m and n. First, it has been done. Secondly, the numbers become very large very quickly.

Definition of the Ackermann function

The Ackermann function A(m, n) is defined for non-negative m and n as follows:

- A(0, n) = n + 1
- A(m, 0) = A(m - 1, 1)
- A(m, n) = A(m - 1, A(m, n - 1))

Recursive implementation of the Ackermann function

This leads to the obvious recursive implementation.

```java
/**
 * Ackermann function.
 * @param m The first parameter
 * @param n The second parameter
 * @return A(m, n)
 */
public static int A(final int m, final int n) {
    if (m == 0)  
        return n + 1;
    if (n == 0)  
```
return A(m-1, 1);
return A(m-1, A(m, n-1));
}

But the table referred to above does offer some ideas on how to calculate this function more efficiently. Sometimes there are closed forms for the function; that is, there is a formula which gives you an answer immediately.

The definition of the Ackermann function tells us that $A(0, n)$ is $n + 1$ for any non-negative $n$.

The table gives us closed forms for $A(1, n)$, $A(2, n)$, and $A(3, n)$ as well.

\[
A(1, n) = n + 2 \\
A(2, n) = 2 * n + 3 \\
A(3, n) = 2^{(n+3)} - 3
\]

This last equation would be implemented as Math.pow(2.0, (double)(n + 3)) - 3.0

Note that the Math.pow method expects both the base (2.0) and the exponent (n + 3) to be doubles. 2.0 is a double; n + 3 is not, so we must cast it to a double. We could have the same effect by using 3.0 instead of 3.

For other values of $m$ and $n$, there are no simple formulas, although there are a few values calculated which we could use. That idea of calculating and saving values is the idea behind dynamic programming.

**Dynamic programming**

*Dynamic programming* is a technique which remembers the results of previous calculations and is described at [http://en.wikipedia.org/wiki/Dynamic_programming](http://en.wikipedia.org/wiki/Dynamic_programming), a description which even mentions the Fibonacci numbers we have examined already.

Why does it mention them?

To calculate Fib(100) you need all the Fibonacci numbers less than Fib(100). So if you follow up the calculation of Fib(100) with a request for Fib(50), it would be much more efficient to simply look up the value in a table rather than to recalculate it.

Dynamic programming involves calculating and keeping the table.

This statement conceals several questions. What type of data structure do we use for the table? When do we create it? Do we want to save it and if so, how do we save it?
Storing the data – Fibonacci

We want a data structure that allows us to store two ints (n, and Fib(n)). We want a data structure that will grow without any special effort on our part. We want a data structure that is efficient in returning values.

One such data structure is the ArrayList, a data structure we have used earlier. We will use n as the index and Fib(n) will be the value in the element. But we can’t create an ArrayList<int>. We can, however, create an ArrayList<Integer>. As we have seen, Integer is a wrapper class for the primitive datatype int. Thus, the Algorithms class needs the following statement.

private static ArrayList<Integer> fibData = new ArrayList<Integer>();

Here’s the new version of the recursive Fibonacci method, rFib.

/**
 * recursive Fibonacci
 * @param n The position of the number we wish
 * @return the nth Fibonacci number
 */
public static int rFib(int n) {
    int result;
    if (fibData.isEmpty()) {
        // Contains no values so load a few
        fibData.add(0, new Integer(0));
        fibData.add(1, new Integer(1));
    }
    try {
        // is the answer in the arraylist?
        Integer x = fibData.get(n);
        return x.intValue();
    }
    catch (Exception e) {
        // if not, calculate it
        result = rFib(n-1) + rFib(n-2);
        // and remember it for next time
        fibData.add(n, new Integer(result));
        return result;
    }
}

We have loaded the ArrayList with the smallest set of values, the values specified in the definition of the function. Any time we calculate another value, we will add it to the ArrayList.

To see that the method actually works, follow these slightly-strange instructions.

Right-click the Algorithms class in the class diagram, and select new Algorithms(); you have now placed an instance of the class on the Object Bench.
Then right-click the Algorithms class in the class diagram, not on the Object Bench, and select rFib to calculate some Fibonacci number.

Finally, right-click the object on the Object Bench, and inspect it, particularly its static fields. What is the size of the ArrayList and what are its contents? The answer depends on which Fibonacci number you calculated and which numbers you have calculated in the past.

The constructor I used above creates an ArrayList of ten elements, according to the ArrayList documentation. When it needs more elements, it grows, but the details of how it grows are not described in the documentation. As a result of the growth, you will sometimes find a number of null elements at the end of the ArrayList.

But what if we wish to save the data so that we can use it again next week or next month?

**Saving the data – Fibonacci**

We have seen how to save data before, when we serialized various classes in our college model; previous chapters contained several ways to serialize data, in fact. This section uses XML serialization.

When we create the file, we will store the index and the value. When we restore the file, we will restore the value to the correct place.

While this may seem overly-complicated for a couple of integers (and it is), it will work well as we deal with functions of more than one variable.

We need to have three import statements, for the List, the ArrayList, and for the XMLEncoder.

```java
import java.util.List;
import java.util.ArrayList;
import java.bean.XMLEncoder;
```

We declare the table of data values.

```java
// private instance variables
private static List<Integer> fibData = new ArrayList<Integer>();
```

Recall that we are using a static method. As such, it must read and write its own files. If the method were associated with an object, we could use the finalize method which is invoked by the garbage collector. But finalize does not work with static methods.

Thus we need to add the following statements to our static method.

```java
// fibdata exists, so remove any extra elements and then try to serialize it
fibData.trimToSize();
try {
    XMLEncoder e = new XMLEncoder(
```
new BufferedOutputStream(
    new FileOutputStream("fibonacci.xml"));
for (int i = 0; i < fibData.size(); i++) {
    e.writeObject(i);
    e.writeObject(fibData.get(i));
}
e.close();
} catch (IOException e) {
    // can't write. maybe in an applet
} // end catch
}

This will create the file, fibonacci.xml.

Run a unit test and see what is stored in the file.

**Restoring the data – Fibonacci**

To retrieve the data stored in the file, we need the following statements, at the beginning of the method, of courses.

if (fibData.isEmpty()) {
    try {
        // does the file exist and is it readable?
        XMLDecoder d = new XMLDecoder(
            new BufferedInputStream(
                new FileInputStream("fibonacci.xml"));

        // read as much data from the file as possible
        boolean more = true;
        Integer value;
        Integer i;
        while (more) {
            try {
                i = (Integer) d.readObject();
                value = (Integer) d.readObject();
                fibData.add(i, value);
            } catch (Exception e) {
                more = false;
            }
        }
    } catch (FileNotFoundException fnfe) {
        // cannot find file, so start with minimal set of values
        fibData.add(0, new Integer(0));
        fibData.add(1, new Integer(1));
    }
} // end try
catch (IOException e) {
    } // end catch

When we use the method, it checks if there already is data in the table. If not, it checks if the file
the method wrote earlier exists; if so, we load the data from it into the table. If not, we create the
table ourselves by adding the two values we need.

**Saving the data – Ackermann function**

The Fibonacci method implements a function of one variable, so we can use that one variable as
the index for an ArrayList, and the value of the function is the element for the ArrayList.

But the Ackermann method implements a function of two variables, so we can’t use an ArrayList.
Instead, we need a HashMap.

Recall that a HashMap provides a key (the two parameters to the function) and a value to which
the key maps (the value of the function.) The datatype Point, although designed for graphics, will
serve as our key. (Note that we could have used a HashMap with the Fibonacci function; the key
would be one integer and the value would be the other.)

import java.util.Map;
import java.util.HashMap;
import java.awt.Point;

We declare the variable

private static Map<Point, Integer> ackData = new HashMap<Point, Integer>();

Whenever we calculate, a result, we add it to the table of remembered data.

if (m == 0) {
    result = n + 1;
    ackData.put(new Point(m, n), new Integer(result));
} else {
    if (n == 0) {
        result = A(m-1, 1);
        ackData.put(new Point(m, n), new Integer(result));
    } else {
        result = A(m-1, A(m, n-1));
        ackData.put(new Point(m, n), new Integer(result));
    }
}
Restoring the data – Ackermann function

We can use the deserialization technique we used in the Fibonacci function to serialize the Ackermann function.

Summary

We have seen how to write several mathematical functions, using iteration and recursion, and using (for the Fibonacci and Ackermann functions) dynamic programming techniques. The other methods here don’t need, but could use, dynamic programming.

The main problem some of these methods have is that the results become large very quickly. We can address that with the BigInteger class, should we wish. Of course, it is always possible to create a number so large that the memory of the computer is not large enough to hold it. That’s the fun of working with numbers; there is always one bigger.
Exercises

1. Implement a class that contains some of the physical or chemical constants.

2. Write a class that contains some of the chemical elements and their properties.

3. Modify your Fibonacci function to use at least longs if not BigIntegers.

   Use your Fibonacci functions to explore the ratio $F(n + 1) / F(n)$ as $n$ increases.

4. Explore Pascal’s triangle and the functions associated with its elements.

   One such function is the choose function, in either its iterative or recursive form.

   Use the formula to allow you to calculate the expansion of $(x + y)$ raised to the power $n$.

5. If you wish to divide $m$ presents among $n$ children, the number of ways to do it is calculated as $\binom{m+n-1}{n-1}$. Write a method to implement the function.

   How does this formula arise?

   Imagine $m + n - 1$ items lined up. Remove $n - 1$ of them. The first child gets all the items (if any) up to the first item removed. The second child gets all the items (if any) up to the second item removed. And so on, until the last child gets all the items (if any) after the last item removed.

6. Recursion appears in many graphics situations.

   Explore the Sierpinski triangle. Create a method which will draw the triangle.

   Explore the Koch snowflake. Create a method which will draw the snowflake.
Chapter 23 – Loose ends

Learning objectives

By the end of the chapter you will be able to:

- Explain public static void main(String[] args)

Introduction

We have looked at many aspects of creating solutions using Java, and all the code we have seen works, works well and is well-written Java. However, there is one topic found in all introductory programming courses which we have omitted.

**public static void main(String[] args)**

This is the statement with which most Java books begin. It is the method with which traditionally-written programs begin.

We have used BlueJ to allow access to the methods of our classes and objects. But not everyone uses BlueJ, and BlueJ has its limitations, too. How do you run programs when you don’t have BlueJ to help?

You create a main method in your class. That is, you create a method whose name is main and whose signature is the heading of this section.

Suppose you are testing. Depending on the testing you wish to do, there may be a main method for each class you wish to test. Recall that we created unit tests for Student and Professor. In each, we tested the constructors, the toString method, and the getters and setters. If we didn’t have BlueJ we would have needed a main method. In it we would have constructed one or more objects and then invoked the methods we wished to test.

Suppose you wish to create a GUI for an application. You will create a class which generates and displays the GUI. That class will contain a main method.

Without an integrated development environment like BlueJ, you may be running from the command line.

You’ll need to use the command javac SomeClass.java to compile the class.
To execute the main method of SomeClass, you’ll need to use the command `java SomeClass` and the Java interpreter will look for a method within that class named `main`.

Let’s examine the parts of the header for the `main` method.

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

- `public` tells us that the method is available outside the class. It has to be so that the Java interpreter can invoke it.
- `static` tells us that there is only one copy of the method in this class.
- `void` tells us that when the method completes its processing, it just completes; it doesn’t return any special value to the operating system.
- `main` tells us the name of the method. The Java interpreter expects to find a `main` method. Yes, BlueJ created the method for us in the background.
- `String[] args` tells us that we can provide arguments to the `main` method, similar to the way we provide arguments to applets.

If you are using the command line, the arguments are placed on the command line after `java SomeClass`.

The arguments are provided as elements of the array of `String` called `args`, which is accessible within the `main` method; the first argument is placed in `args[0]`, the second in `args[1]`.

**Summary**

This brief chapter completes our discussion of the Java language. Now you have a good basis for going forward and using Java, and for learning more about Java, since your textbook is only an introduction to Java.
Exercises

1. Modify the CollegeMenu so that it uses a main method.

Once you have done that, create a jar file of your project. Do not include the source code. Transfer the jar file to another computer, one which has the Java Runtime Environment (installed). Open the jar file and execution begins with the main method in CollegeMenu.
In Conclusion

Now we have seen how the basic (and some not-so-basic) techniques of Java allow us to implement our ideas. Go forth and expand your knowledge.

Here are some possible areas you may wish to explore to build upon the knowledge you have gained here.

- **Professor class** – office number, phone number, degrees (where, subject, when). Include subjects taught and when as a separate class.
- **Databases.** How to normalize them and how to use them with Java programs.
- **A related example of inheritance.** Sports players and their teams. Contact me if you wish to see the labs I used while teaching a course using this textbook for the first time; they modelled a lacrosse league.
- **Customization** – How do you allow people to change the colours of your interface? How do you allow them to change the way courses and rooms are numbered? How did BlueJ implement its Preferences dialogu?
- **Program standard OO examples.** The most common one is different types of bank accounts. This environment will use inheritance and the Singleton pattern to provide interest rates.
References

Printed Reference materials


Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995


Hu, Chenglie. Just Say “A Class Defines a Data Type”, Communications of the ACM, volume 51, number 3, 19-21


Marrero, Will, and Amber Settle, *Testing First: Emphasizing Testing in Early Programming Classes*, ACM SIGCSE Bulletin, volume 37, number 3, 4-8

Metsker, Steven John, *Design Patterns Java Workbook*, Addison-Wesley, 2002


Online references


Horstmann, Cay. [http://www.horstmann.com/violet](http://www.horstmann.com/violet)
Violet is intended for students, teachers, and authors who need to produce simple UML diagrams quickly.

Java online documentation. [http://download.oracle.com/javase/6/docs/api/index](http://download.oracle.com/javase/6/docs/api/index).

Java tutorial. [http://download.oracle.com/javase/tutorial/index](http://download.oracle.com/javase/tutorial/index)

JUnit.org. [http://www.junit.org](http://www.junit.org)


O’Hanley, John, [http://www.javapractices.com](http://www.javapractices.com)


Unicode.org [http://www.unicode.org](http://www.unicode.org)

Lab Assignments

I wrote this textbook to be used in conjunction with lab assignments.

Some years I have used the college model developed in the textbook. Having some of the code available made life easier for the students and taught them something about their college.

Some years I have used a sports league. I used lacrosse, Canada's national summer sport, but you could use any sport. The problem with using a sport is that many in your class might not be interested in sports.

But my favourite assignments involve birding and tracking the different species you have seen. If you would like to see those assignments, please contact me at rdgee at shaw dot ca.
Index

@param, 28, 29, 36, 40, 46, 72, 109, 123, 127, 142, 152, 153, 169, 173, 177, 191, 204, 205, 206, 207, 221, 236, 262, 263, 265, 289, 292, 296, 299, 302, 303, 305, 319, 362, 365, 366, 416, 437, 455, 464, 465, 466, 467, 469
@return, 29, 36, 39, 46, 47, 72, 99, 109, 124, 129, 143, 146, 169, 171, 172, 175, 206, 217, 242, 253, 265, 266, 296, 303, 305, 310, 311, 312, 313, 314, 319, 362, 366, 416, 417, 420, 421, 454, 455, 464, 465, 466, 467, 469
@throws, 109, 263, 302, 303, 305

above, 354
abs, 450, 451, 452, 453
abstract, 70, 93, 95, 96, 97, 139, 146, 189, 198, 277, 307
Abstract Window Toolkit
  AWT, 332, 337, 347, 397, 415, 417
AbstractButton, 387
abstraction, 101, 187
accessibility, 397, 400, 405
Ackermann, 467, 468, 472, 473
ActionCommand, 387, 403
ActionEvent, 355, 356, 386, 387, 388, 403
ActionListener, 355, 356, 358, 363, 367, 386, 387, 388, 402
ad hoc, 56, 147
Adapter, 447
addActionListener, 356, 388, 389, 402
addAll, 201, 204, 205
AddressException, 135
Afghanistan, 433
African Union, 433
age, 137, 463
aggregation, 213, 235, 236
Albania, 433
algebra, 77, 78, 79, 88
ALPHABETIC_ORDER, 197, 198, 199, 201, 204, 205, 207
analysis, 93, 160, 192, 428
Apple, 15, 177
aplet
  stop, 22, 56, 117, 304, 379, 416, 417, 418, 419, 420
Applet, 413, 416, 417, 419, 422, 426, 427
destroy, 271, 416, 417, 420
clone, 128, 129, 130, 131, 132, 135, 175, 183, 191, 205, 220, 221, 260, 268, 299, 328, 334, 482
Cloneable, 129, 130, 163, 274, 442
CloneNotSupportedException, 106, 129, 130, 260, 261, 268
CodePad, 126, 418
coffee, 21, 439
cohesion, 234
cohesive, 234, 235, 443
College classes
AddressException, 135
CourseException, 389
Department, 160, 179, 180, 182, 183, 185, 286
HighestMark, 255, 256, 277
Mark, 236, 237, 239, 241, 243, 244, 246, 254, 255, 256, 277, 280, 286, 367, 368, 378
MyDate, 140, 142, 143, 144, 146, 147, 148, 149, 150, 152, 153, 155, 156, 157, 182, 199, 273, 277, 286, 310, 311, 313, 315, 317, 318, 319, 320, 336, 338, 340, 341, 349, 356, 373
MyDateException, 152, 153, 154, 156, 317
Name, 13, 28, 122, 134, 142, 179, 310, 314, 315, 316, 317, 319, 320, 352
NameException, 120, 316, 317, 319, 320
NonnumericIdentifierException, 108, 116, 117, 118
CourseException, 389
CPU, 14, 49
Create Test Class, 50, 58, 59
createVerticalBox, 374
Creator, 442, 447
CSV, 301, 310, 311, 312, 313, 314, 315, 317, 318, 319, 320, 329
currentThread, 289, 292, 293, 296

Data structure
array

Data Structure
ArrayList, 163, 191, 201, 216, 222, 255, 469, 470, 472
Circular List, 191
Generic, 140, 166, 279, 332, 405, 406, 408
queue, 347, 415
Set, 99, 163, 165, 166, 167, 172, 173, 179, 185, 190, 200, 201, 204, 205, 206, 207, 216, 220, 280, 295, 328, 339, 366, 380, 422, 441, 482
TreeSet, 163, 187, 200, 201, 204, 207, 208, 216, 217, 220, 222, 243, 283, 294
Vector, 163, 378
database, 16, 162, 262, 287, 305, 327, 375, 385, 446

DataFormatException, 106, 108, 261, 262, 263, 265, 266, 267
datatype, 23, 32, 36, 37, 39, 40, 56, 60, 74, 77, 97, 117, 125, 143, 144, 188, 245, 276, 283, 284, 290, 291, 360, 458, 460, 469, 472
Date, 138, 139, 140
debt, 449, 451
debug, 202, 203
debugger, 43
decimal, 24, 143, 144, 145, 188, 189, 242, 245
DecimalFormat, 189, 194, 211, 223, 230, 232, 245
Decorator, 447

Definitions
gigahertz, 14
hardware, 13, 16, 57
i8ln, 405, 408
MB, 14
Megahertz, 14
overload, 206
override, 100, 278, 416, 417, 420, 421
RAM, 14, 15, 272
ROM, 14
software, 13, 16, 26, 57, 68, 357, 404, 439, 440
wildcard, 275, 304
degree, 15, 103, 170, 222
denominator, 242, 452, 453, 454, 455, 456
Department, 160, 179, 180, 182, 183, 185, 286
deprecated, 138, 139
derived class, 94, 96, 97, 98, 99, 100, 106, 127, 146, 149
deriveFont, 423
deserialize, 273, 276, 278, 297, 281
design, 13, 25, 49, 71, 74, 93, 97, 160, 161, 187, 273, 277, 331, 334, 376, 390, 408, 409, 428, 439, 440, 442, 443, 444, 446, 456, 467
destroy, 271, 416, 417, 420
dialog, 55, 59, 357
dictionary, 162, 163
Direction, 122, 314, 349, 353, 355, 377
Direction.ABOVE, 353
Direction.BELOW, 349, 377
Direction.LEFT, 353, 355
Direction.RIGHT, 377
byte, 23, 141, 449
float, 24, 71, 188, 307
for loop, 71, 172, 227, 228, 233, 307, 383, 432
for-each, 159, 172, 173, 176, 202, 208, 227, 228, 239
java.awt.event.MouseListener, 385
java.awt.event.WindowAdapter, 394
java.awt.event.WindowEvent, 394
java.awt.EventQueue, 340, 348, 373, 399
java.awt.FlowLayout, 334, 355, 374, 388, 391
java.awt.Font
g.getFont, 423
java.awt.Frame, 340, 349, 352, 372
    MAXIMIZED_BOTH, 352
    setState, 352
java.awt.Graphics
drawOval, 430, 431, 432
drawPolygon, 424, 430, 436, 438
drawRect, 430
drawString, 416, 419, 423, 451
fillOval, 430
fillPolygon, 430, 438
fillRect, 416, 419, 423, 430
getClipBounds, 430
setColor, 416, 419, 423, 432, 451
java.awt.GridBagLayout, 335, 368, 393
java.awt.GridLayout, 334
java.awt.Point, 472
java.awt.Window, 44, 48, 50, 54, 56, 59, 65,
    68, 81, 84, 126, 128, 134, 174, 183, 185,
    205, 303, 304, 306, 328, 334, 335, 341,
    395, 413, 419, 420, 421, 426, 428
java.bean, 288, 291, 296, 297, 298, 470
java.beans.Encoder
    setPersistenceDelegate, 296
java.beans.PersistenceDelegate, 297
java.io, 273, 274, 275, 278, 279, 301, 304,
    306
    flush, 275, 281, 296
java.io.BufferedOutputStream, 289, 296,
    471
java.io.BufferedReader, 302, 304, 315, 320
java.io.BufferedReader, 306, 314
java.io.FileInputStream, 273, 276, 282, 292,
    471
java.io.FileOutputStream, 273, 274, 275,
    280, 289, 296, 471
java.io.FileReader
    FileReader, 301, 302, 303, 305, 315, 320
java.io.FileWriter
    FileWriter, 306, 314
java.io.InputStream, 302, 303, 304, 305
java.io.InputStreamReader, 302, 303, 304
java.io.IOException, 106, 259, 260, 261,
    275, 276, 278, 279, 281, 282, 302, 303,
java.io.ObjectInputStream, 273, 276, 278,
    279, 282, 300
    readObject, 276, 277, 278, 279, 280, 282,
    283, 284, 288, 292, 293, 299, 300, 471
java.io.ObjectOutputStream, 273, 274, 275,
    276, 277, 278, 279, 280, 281, 284, 288,
    289, 296, 297, 300, 471
java.io.Serializable, 273, 274, 275, 277, 442
java.lang.Character, 91, 116, 117, 197, 252
java.lang.Class
    getClass, 289, 292, 293, 296, 303, 305
    getClassLoader, 289, 292, 293, 296, 303,
    305
java.lang.Double, 122, 426, 437
java.lang.Long, 141, 142, 143, 144, 145,
    431, 432
java.lang.Math, 89, 90, 450, 452, 456
    abs, 450, 451, 452, 453
    BigDecimal, 451, 452
    BigInteger, 449, 450, 451, 452, 458, 473
java.lang.String, 23, 24, 28, 29, 31, 32, 34,
    35, 36, 37, 38, 39, 40, 41, 46, 47, 55, 62,
    64, 71, 72, 73, 74, 80, 81, 82, 83, 84, 85,
    90, 91, 96, 97, 98, 99, 100, 106, 109, 110,
    112, 113, 114, 115, 117, 118, 121,
    122, 123, 124, 125, 126, 127, 128, 130,
    132, 142, 143, 145, 152, 173, 177, 178,
    179, 180, 181, 182, 187, 189, 192, 194,
    197, 199, 200, 201, 202, 204, 205, 206,
    207, 208, 213, 214, 217, 222, 223, 225,
    230, 232, 233, 234, 235, 236, 239, 241,
    242, 243, 246, 247, 248, 249, 250, 251,
    252, 253, 254, 255, 256, 262, 263, 265,
    266, 274, 279, 281, 282, 292, 295, 299,
    302, 303, 304, 305, 307, 310, 311, 312,
    313, 314, 315, 316, 318, 319, 320, 359,
    360, 361, 375, 379, 382, 384, 385, 386,
    389, 407, 416, 417, 420, 421, 422, 429,
    451, 454, 456, 461, 475, 476
charAt, 85, 86, 90, 91, 99, 116, 117, 230, 233, 248, 251, 252, 382, 389
equalsIgnoreCase, 252, 403
indexOf, 111, 112, 248, 250, 253, 254, 256
isLetter, 91, 252
lastIndexOf, 250
length, 80, 82, 83, 84, 97, 98, 108, 110, 114, 115, 124, 234, 251, 252, 359, 361, 384, 385
toLowerCase, 31, 44, 141, 143, 151, 168, 188, 197, 199, 236, 352
java.lang.StringBuffer, 111, 213, 224, 225, 226, 227, 228, 229, 232, 234, 239
setCharAt, 230, 231, 233, 234
java.lang.StringBuilder, 225, 239
err, 126, 128, 275, 276, 279, 281, 282, 289, 292, 315, 317, 321
exit, 119, 269, 403
getProperties, 418, 438
out, 126, 174, 183, 204, 249, 280, 292, 293, 302, 304, 306, 316, 317, 320, 418
java.lang.Thread
setContextClassLoader, 289, 292, 293, 296
java.security, 278, 286
java.swing.JList
setSelected, 386, 390
java.text, 189, 230
java.text.DecimalFormat, 189, 194, 211, 223, 230, 232, 245
java.text.NumberFormat, 189
java.util.Calendar, 139, 154, 193, 213, 223, 231
java.util.EventObject
ggetSource, 387, 403
java.util.GregorianCalendar, 88, 139, 154, 157, 193, 362
java.util.HashMap
Key, 70, 71, 307
java.util.Locale, 407
java.util.Map
Key, 70, 71, 307
java.util.Map.Entry, 300

java.util.regex.Pattern, 248, 249, 347
regular expression, 248, 249, 250, 318
java.util.ResourceBundle, 239, 407, 408
java.util.Scanner, 315, 317, 318, 319, 320
readLine, 302, 304, 315, 317, 320
java.util.Set, 99, 163, 165, 166, 167, 172, 173, 179, 185, 190, 200, 201, 204, 205, 206, 207, 216, 220, 280, 295, 328, 339, 365, 366, 441, 482
java.util.StringTokenizer, 318
java.util.TreeSet, 163, 187, 200, 201, 204, 205, 207, 208, 216, 217, 220, 222, 243, 283, 294
javac, 35, 475
javadoc, 29, 61, 64, 70, 71, 81, 152, 165, 198, 265, 307
postcondition, 198
precondition, 198
JavaHelp, 404
javax.accessibility
setAccessibleDescription, 397, 400
javax.swing, 332, 337, 351, 376, 397, 415, 432
javax.swing.AbstractButton, 387
javax.swing.ActionCommand, 387, 403
javax.swing.ActionEvent, 355, 356, 386, 387, 388, 403
javax.swing.addActionListener, 356, 388, 389, 402
javax.swing.Border, 376
javax.swing.BorderFactory, 376, 444
javax.swing.BoxLayout, 334, 373, 374
javax.swing.Box, 374, 375
javax.swing.BoxLayout, 374
javax.swing.ButtonGroup, 380, 382, 387
javax.swing.InputVerifier, 360, 362, 385
javax.swing.JApplet, 332, 415, 416, 417, 418, 419, 420, 421
javax.swing.JButton, 332, 342, 353, 354, 356, 388
javax.swing.JFormattedTextField, 360, 365, 392
javax.swing.JCheckBox, 332, 371, 382
javax.swing.JFrame, 332, 359, 361, 384, 385
javax.swing.JEditorPane, 346
javax.swing.JTextField, 332, 334, 338, 341, 373, 376, 377, 388
javax.swing.JScrollPane, 332, 342, 353, 354, 356, 388
javax.swing.setLayout, 360, 365, 392
javax.swing.JComponent, 356, 388, 399
javax.swing.ImageIcon, 394
javax.swing.JEditorPane, 346
javax.swing.JTextField, 332, 334, 338, 341, 373, 376, 377, 388
javax.swing.JComboBox, 352
javax.swing.JTextField, 340, 348, 350, 373
javax.swing.JScrollBar, 350
javax.swing.JLabel, 352
javax.swing.JTextField, 364, 367, 390
javax.swing.JLabel, 352
javax.swing.JTextField, 340, 349, 358, 364, 373, 388, 399
javax.swing.JList, 371, 374, 375, 387, 390
javax.swing.JMenu, 15, 44, 48, 50, 54, 58, 65, 74, 81, 84, 95, 126, 152, 154, 166, 174, 202, 332, 333, 392, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 407, 408, 409, 414, 444
javax.swing.JMenuItem, 332, 398, 403, 407
javax.swing.JOptionPane, 356, 359, 361, 362, 364, 384, 385, 388, 403
javax.swing.JOptionPane, 356, 357, 358, 359, 361, 362, 364, 384, 385, 388, 403
javax.swing.JPasswordField, 346, 369
javax.swing.JRadioButton, 332, 371, 381, 382, 386
javax.swing.JScrollPane, 346, 369, 371, 375
javax.swing.JSeparator, 401
javax.swing.JTextArea, 346, 369
javax.swing.JTextField, 332, 342, 353, 354, 356, 388
javax.swing.JOptionPane, 356, 357, 358, 359, 361, 362, 364, 384, 385, 388, 403
javax.swing.JPasswordField, 346, 369
javax.swing.JRadioButton, 332, 371, 381, 382, 386
javax.swing.JScrollPane, 346, 369, 371, 375
javax.swing.JSeparator, 401
javax.swing.JTextArea, 346, 369
javax.swing.JTextField, 332, 342, 353, 354, 356, 388
javax.swing.JOptionPane, 356, 357, 358, 359, 361, 362, 364, 384, 385, 388, 403
javax.swing.JPasswordField, 346, 369
javax.swing.JRadioButton, 332, 371, 381, 382, 386
javax.swing.JScrollPane, 346, 369, 371, 375
javax.swing.JSeparator, 401
javax.swing.JTextArea, 346, 369
javax.swing.JTextField, 332, 342, 353, 354, 356, 388
javax.swing.JOptionPane, 356, 357, 358, 359, 361, 362, 364, 384, 385, 388, 403
javax.swing.JPasswordField, 346, 369
javax.swing.JRadioButton, 332, 371, 381, 382, 386
javax.swing.JScrollPane, 346, 369, 371, 375
javax.swing.JSeparator, 401
javax.swing.JTextArea, 346, 369
ListSelectionModel, 375
Listener List, 163, 185
List, 163, 185
less than, 163, 185
Layout manager, 331, 332, 334, 335, 336, 337, 338, 341, 343, 344, 345, 346, 355, 368, 371, 374, 375, 388, 390, 393
Layout manager
BorderLayout, 334, 373, 374
BoxLayout, 374
FlowLayout, 334, 355, 374, 388, 391
GridBagLayout, 335, 368, 393
GridLayout, 334
RelativeLayout, 4, 335, 336, 337, 338, 341, 355, 374, 376, 377, 388, 390, 391, 427, 482
leap year, 87, 88, 152, 153, 154, 362
leftAlignedWith, 339, 340, 344, 382, 384
less than, 85, 89, 116, 194, 196, 197, 199, 218, 219, 245, 246, 351, 465, 468
Libya, 433
List, 163, 185, 190, 191, 195, 201, 216, 378, 470
ArrayList, 163, 191, 201, 216, 222, 255, 469, 470, 472
Vector, 163, 378
Listener
MouseListener, 385
ListSelectionModel, 375
Locale, 407
logic, 25, 79, 80, 107, 154, 197, 228, 251, 280, 304, 362, 437
Long, 141, 142, 143, 144, 145, 431, 432
look and feel, 332, 368
lowercase, 31, 44, 141, 143, 151, 168, 188, 197, 199, 236, 352
makeCollections, 294
Map, 163, 300, 328, 441, 472
HashMap, 163, 171, 472
Map.Entry, 300
Mark, 236, 237, 239, 241, 243, 244, 246, 254, 255, 256, 277, 280, 286, 367, 368, 378
Marker, 442
Math, 89, 431, 432, 435, 436, 437, 450, 451, 452, 453, 468
Math.abs, 453
Math.pow, 468
mathematics, 5, 35, 77, 242, 434, 436, 450, 456, 465
Mathematics
Ackermann, 467, 468, 472, 473
Algebra, 77, 78, 79, 88
choose, 466, 474
Complex Number, 449, 452, 456, 458
cosine, 431, 432, 434, 435
Factorial, 465, 466
Fibonacci, 462, 463, 464, 468, 469, 470, 471, 472, 473, 474
greatest common divisor, 450, 451, 454, 460, 461
hypotenuse, 431, 432, 434
Modulus, 86, 91, 215, 239, 461
Rational number, 449, 452, 453, 454, 455, 456
lacrosse, 267, 479
lastIndexOf, 331, 332, 334, 335, 336, 337, 338, 341, 343, 344, 345, 346, 355, 368, 371, 374, 375, 388, 390, 393
 Lacrosse
nullary constructor, 130, 290, 292, 293, 294
overload, 206
override, 100, 278, 416, 417, 420, 421
setter, 17, 27, 36, 37, 48, 61, 62, 63, 68, 82, 90, 100, 126, 131, 141, 148, 172, 290, 452
toString, 17, 28, 37, 38, 39, 41, 42, 47, 48, 50, 53, 68, 73, 74, 75, 77, 80, 82, 83, 84, 96, 97, 98, 99, 100, 122, 123, 124, 125, 126, 131, 132, 140, 141, 142, 143, 144, 145, 147, 150, 152, 157, 173, 177, 179, 188, 189, 194, 201, 202, 204, 223, 237, 243, 244, 246, 280, 293, 367, 386, 387, 420, 451, 452, 453, 454, 456, 475
Object Bench, 134, 202, 271, 469, 470
ObjectInputStream, 273, 276, 278, 279, 282, 283, 300
ObjectOutputStream, 273, 274, 275, 278, 279, 280, 281, 300
Observer, 443, 447
octal, 145
Okanagan College, 5, 6, 9, 18, 24, 85, 102, 103, 137, 148, 222, 241, 242, 255, 300
Olympic Movement, 433
open, 36, 42, 48, 70, 72, 84, 126, 140, 276, 303, 305, 336, 396, 397, 404, 415, 418
Oracle, 102, 105, 259, 335, 404
Organization
  African Union, 433
  Olympic Movement, 433
overload, 206
override, 100, 278, 416, 417, 420, 421
pack, 340, 349, 373
Paint, 416
palindrome, 251, 252
panel, 126, 372, 374, 375, 376, 377, 380, 383, 387, 388, 390, 391
paradigm, 16, 19, 69
parameter, 32, 36, 37, 50, 62, 64, 74, 90, 91, 109, 110, 112, 117, 118, 127, 131, 167, 169, 170, 174, 206, 208, 209, 220, 250,
265, 289, 290, 297, 300, 349, 356, 359, 407, 417, 419, 421, 422, 429, 430, 431, 433, 437, 467
parent class, 94, 96, 98, 107, 149, 255, 261, 277
parentheses, 38, 40, 43, 47, 77, 80, 111, 150, 151, 152, 157, 215
parenthesis, 38, 47, 151
Pattern, 447
Adapter, 447
Bridge, 447
Builder, 447
Capturing Light, 439, 440
Chain of Responsibility, 447
Cohesion, 234
Command, 447
Composite, 447
Controller, 447
Coupling, 312, 443, 446, 447
Creator, 442, 447
Decorator, 447
Factory, 444, 447, 452
Flyweight, 447
Information Expert, 447
Loose Coupling, 456
Marker, 442
Mediator, 443, 447
Memento, 447
MVC, 445, 446
Observer, 443, 447
Prototype, 447
Proxy, 447
Singleton, 180, 181, 182, 194, 204, 207, 227, 232, 236, 342, 381, 403, 409, 440, 445, 447, 479
Visitor, 447
PDF, 143, 197, 288
persistence, 5, 238, 256, 271, 272, 287, 288, 300, 301, 327
PersistenceDelegate, 297
pixel, 343
Planck, 459
Point, 472
polygon, 424, 425, 434, 436
polyline, 434
polymorphism, 206
postcondition, 198
precondition, 198
preference, 48, 60, 174, 390
primitive, 17, 23, 71, 117, 141, 142, 143, 152, 175, 188, 191, 217, 219, 244, 273, 283, 300, 307, 318, 360, 379, 450, 460, 469
principle, 49, 442, 443, 444, 456
Print, 396, 400
Professor, 11, 20, 21, 24, 26, 42, 69, 74, 93, 94, 95, 96, 97, 98, 100, 101, 107, 121, 127, 131, 133, 134, 146, 147, 149, 159, 160, 161, 166, 167, 168, 172, 173, 175, 177, 178, 179, 180, 181, 182, 194, 196, 198, 201, 202, 204, 210, 237, 238, 239, 261, 277, 284, 367, 368, 442, 445, 475, 479
programming style, 19, 30, 32, 124
protected, 97, 127, 294
Prototype, 447
Proxy, 447
public, 28, 29, 30, 34, 36, 37, 38, 39, 40, 41, 46, 47, 49, 50, 60, 61, 62, 70, 72, 73, 80, 82, 83, 84, 85, 86, 87, 90, 95, 97, 98, 99, 109, 113, 114, 115, 117, 123, 124, 128, 129, 130, 131, 142, 143, 145, 146, 148,

question mark, 48, 219
queue, 347, 415

RAM, 14, 15, 272
rational number, 449, 452, 453, 454, 455, 456
reading, 55, 150, 189, 202, 253, 272, 303, 319, 445
readLine, 302, 304, 315, 317, 320
readObject, 276, 277, 278, 279, 280, 282, 283, 284, 288, 292, 293, 299, 300, 471
recipe, 439
Rectangle, 430
recursion, 251, 459, 473
recursive, 251, 451, 459, 460, 461, 462, 464, 465, 466, 467, 469, 474
refactoring, 192
reference, 50, 55, 69, 97, 107, 127, 129, 176, 252, 253, 261, 305, 380
reflection, 312, 313, 435
regression, 56, 57, 256
regular expression, 248, 249, 250, 318
RelativeLayout, 4, 335, 336, 337, 338, 341, 355, 374, 376, 377, 388, 390, 391, 427, 482
Above, 354
addBinding, 338, 339, 340, 344, 345, 354, 355
addBindings, 345, 363, 377, 382, 383
Below, 338, 339, 344, 350, 382, 383
BindingFactory, 338, 341, 342, 343, 344, 377, 444
Constraint, 343, 349, 377
Direction, 122, 314, 349, 353, 355, 377
ABOVE, 353
BELOW, 349, 377
LEFT, 353, 355
RIGHT, 377
Edge, 349, 353, 355, 377
leftAlignedWith, 339, 340, 344, 382, 384
rightAlignedWith, 344
setVerticalSpacing, 344
topAlign, 339, 345, 355, 363, 377
requestFocusInWindow, 394
ResourceBundle, 239, 407, 408
retirement, 137, 148, 149, 150, 172, 278
RGB, 71, 423
rightAlignedWith, 344
risks, 203
ROM, 14
RSA, 449
rugby, 267
Runnable, 340, 347, 348, 372, 399
Scalable Vector Graphics, 288, 328
Scanner, 315, 317, 318, 319, 320
scope, 70, 173, 235
protected, 97, 127, 294
Scrabble, 162
SECTION_NUMBER_ORDER, 236
SectionException, 384, 389
security, 130, 415
serializable, 273, 274, 275, 277, 442
serialize, 272, 273, 274, 275, 276, 277, 278, 281, 284, 286, 427, 470, 473
Set, 99, 163, 165, 166, 167, 172, 173, 179, 185, 190, 200, 201, 204, 205, 206, 207, 216, 220, 280, 295, 328, 339, 365, 366, 411, 482
HashSet, 163, 165, 166, 167, 168, 171, 172, 176, 177, 179, 180, 183, 200, 201, 205, 207, 208, 216, 280, 282, 294
TreeSet, 163, 187, 200, 201, 204, 207, 208, 216, 217, 220, 222, 243, 283, 294
setAccessibleDescription, 397, 400
setBorder, 376, 387
setCharAt, 230, 231, 233, 234
setColor, 416, 419, 423, 432, 451
setContextClassLoader, 289, 292, 293, 296
setDisplayMnemonic, 352
setDisplayMnemonicIndex, 352
setExtendedMnemonic, 352
setFont, 423
setIcon, 353
setInputVerifier, 360, 362, 385
setLabelFor, 352
setLayout, 334, 338, 341, 373, 376, 377, 388
setMinimumSize, 399
setMnemonic, 354, 388, 397
setPersistenceDelegate, 296
setPreferredSize, 340, 348, 350, 373
setSelected, 386, 390
setSize, 350
setter, 17, 27, 36, 37, 48, 61, 62, 63, 68, 82, 90, 100, 126, 131, 141, 148, 172, 290, 452
setInputVerifier, 360, 362, 385
setter
setAccessibleDescription, 397, 400
setBorders, 376, 387
setCharAt, 230, 231, 233, 234
setColor, 416, 419, 423, 432, 451
setContextClassLoader, 289, 292, 293, 296
setDisplayedMnemonic, 352
setDisplayedMnemonicIndex, 352
setExtendedState, 352
setFont, 423
setIcon, 353
setLabelFor, 352
setLayout, 334, 338, 341, 373, 376, 377, 388
setMinimumSize, 399
setMnemonic, 354, 388, 397
setPersistenceDelegate, 296
setPreferredSize, 340, 348, 350, 373
setSelected, 386, 390
setSize, 350
setText, 364, 387, 390
setVerifyInputWhenFocusTarget, 360, 365, 392
setVerticalSpacing, 344
setVisible, 340, 349, 358, 364, 373, 388, 399
showMessageDialog, 356, 357, 358, 359, 361, 362, 364, 384, 385, 388, 403
setText, 364, 387, 390
setVerifyInputWhenFocusTarget, 360, 365, 392
setVerticalSpacing, 344
setVisible, 340, 349, 358, 364, 373, 388, 399
showMessageDialog, 356, 357, 358, 359, 361, 362, 364, 384, 385, 388, 403
sighting, 157, 369, 393
signature, 128, 169, 175, 434, 436, 437, 475
sine, 431, 432, 434, 435
Singleton, 180, 181, 194, 260, 342, 403, 409, 440, 445, 447, 479
getInstance, 181, 182, 204, 207, 227, 232, 381
skeleton, 30, 44, 46, 60, 72, 140, 165, 186, 237, 243, 279
Smalltalk, 19, 441, 445
smoke testing, 68
software, 5, 13, 16, 26, 57, 68, 357, 404, 439, 440
split, 248, 249, 347
Sport
  baseball, 267
  basketball, 267
  football, 267
  Lacrosse, 267, 479
  rugby, 267

Water polo, 267
state, 17, 19, 27, 41, 60, 105, 122, 201, 207, 214, 259, 272, 278, 285, 443
stop, 22, 56, 117, 304, 379, 416, 417, 418, 419, 420
Strategy, 447
stream, 272, 273, 275, 276, 278, 302, 303, 304, 305, 306
Google, 4, 163, 164, 186, 335, 423
Guava, 163
HTML, 342, 343, 351, 353, 369, 425, 426, 427
integrated development environment, 43, 47, 481
Integrated development environment, 43, 44, 475
Internet Explorer, 291, 298, 415
Interpreter, 166, 347, 476
javac, 35, 475
javadoc, 29, 61, 64, 70, 71, 81, 152, 165, 198, 265, 307
JavaHelp, 404
jig, 439
moe.defs, 70, 71
native2ascii, 406
refactoring, 192
Risks, 203
Smalltalk, 19, 441, 445
stub, 385
Unicode, 23, 75, 141, 196, 197, 406, 482
Wikipedia, 16, 56, 100, 125, 153, 202, 245, 251, 252, 287, 332, 333, 347, 440, 461, 462, 464, 482
XML, 287, 288, 289, 291, 292, 293, 294, 296, 297, 299, 300, 327, 328, 396, 427, 445, 446, 470
XMLSpy, 291, 300
tooltip, 351
ToolTipManager, 351
toAlign, 339, 345, 355, 363, 377
toRadians, 431, 432
toString, 17, 28, 37, 38, 39, 41, 42, 47, 48, 50, 53, 68, 73, 74, 75, 77, 80, 82, 83, 84, 96, 97, 98, 99, 100, 122, 123, 124, 125, 126, 131, 132, 140, 141, 142, 143, 144, 145, 147, 150, 152, 157, 173, 177, 179, 188, 189, 194, 201, 202, 204, 223, 237, 243, 244, 246, 280, 293, 367, 386, 387, 420, 451, 452, 453, 454, 456, 475
transaction, 74, 368, 393
transcript, 25, 121, 237, 241, 242, 243, 247, 252, 253, 254, 255, 256, 257
transient, 277, 278, 280
TreeSet, 163, 187, 200, 201, 204, 207, 208, 216, 217, 220, 222, 243, 283, 294
trigonometry, 431, 434
Try
Twitter, 443
typeface, 19, 74, 343, 345
Ukraine, 433
UML, 21, 22, 23, 24, 26, 43, 59, 481, 482
understanding, 43, 187, 202, 218, 238, 443, 460
Unicode, 23, 75, 141, 196, 197, 406, 482
unit test, 1, 4, 43, 56, 57, 58, 59, 60, 63, 64, 68, 69, 73, 74, 76, 86, 99, 100, 101, 112, 117, 118, 126, 131, 132, 133, 134, 141, 142, 144, 146, 147, 149, 150, 154, 170, 173, 174, 175, 182, 183, 188, 190, 193, 194, 198, 199, 202, 203, 204, 205, 209, 235, 237, 256, 263, 266, 267, 274, 275, 276, 277, 280, 281, 282, 289, 290, 292, 296, 300, 331, 454, 471, 475
Unit testing
assertion, 61, 62, 64, 86, 113, 114, 115, 117, 131, 150, 172, 174, 199, 267, 275, 276, 281, 282, 293
fail, 64, 67, 68, 113, 114, 115, 117, 118, 267, 276
JUnit, 1, 2, 4, 17, 57, 62, 64, 68, 174, 367, 482
United Arab Emirates, 433
United Kingdom, 75, 135, 433
United States, 23, 433, 437
Uppercase, 197
URL, 39, 40, 42, 163, 301, 303, 304, 305, 404
user experience, 331
UTF-8, 291, 297, 298, 406
valueOf, 91, 316, 318, 444, 452
Vector, 163, 378
verify, 134, 135, 358, 359, 360, 361, 362, 384, 385, 391
view, 196, 422, 445, 446
virtual machine, 49, 119, 269
visible, 29, 69, 81, 154, 333, 340, 344, 346, 349, 351, 358, 363, 364, 391, 393, 404
Visitor, 447
water polo, 267
while, 111, 112, 116, 117, 159, 176, 177, 178, 179, 253, 302, 304, 315, 320, 454, 460, 471
whitespace, 222, 249, 318, 333
Wikipedia, 16, 56, 100, 125, 153, 202, 245, 251, 252, 287, 332, 333, 347, 440, 461, 462, 464, 482
wildcard, 275, 304
window, 44, 48, 50, 54, 56, 59, 65, 68, 81, 84, 126, 128, 134, 174, 183, 185, 205, 303, 304, 306, 328, 334, 335, 341, 395, 413, 419, 420, 421, 426, 428
WindowAdapter, 394
WindowEvent, 394
WindowFocusListener, 394
writeObject, 275, 276, 277, 278, 279, 280, 281, 284, 288, 289, 296, 297, 300, 471
writing, 29, 30, 63, 80, 93, 124, 166, 222, 269, 272, 273, 279, 284, 301, 327, 329, 335, 362, 406, 422, 451
XML, 287, 288, 289, 291, 292, 293, 294, 296, 297, 299, 300, 327, 328, 396, 427, 445, 446, 470
SVG, 288, 328
XMLDecoder, 288, 290, 291, 292, 293, 297, 299, 300, 471
XMLEncoder, 288, 289, 290, 291, 293, 296, 298, 444, 470
XMLDecoder, 288, 290, 291, 292, 293, 297, 299, 300, 471
XMLSpy, 291, 300