Introduction to Computer Science with Java Programming

Seth Bergmann

DOI: 10.31986/
issn.2689-0690_rdw.
oer.1000

5-1-2017

Introduction to Computer Science with Java Programming

Seth D. Bergmann
Rowan University

DOI: 10.31986/issn.2689-0690_rdw.oer.1000

Introduction to Computer Science with Java Programming

Seth D. Bergmann

May 1, 2019

Preface

This book is intended to be used for a first course in computer programming. No prior experience with programming should be necessary in order to use this book. But this book is intended to be used with a course that teaches more than computer programming; it is intended to be used with a course that teaches Computer Science. The distinction is subtle, but important.

The author(s) believe that a breadth-first approach is the best way to introduce the concepts of Computer Science to students. Rather than isolate topics in courses (bits and bytes in a computer organization course; formal grammars and languages in a theory course; lists, sets, and maps in a data structurs course; etc) we believe that topics should be introduced in a brief and simple manner at the starting level. Elaboration on these topics should occur in subsequent courses. This breadth-first approach allows the student to build on existing knowledge and retain a greater proportion of the material.

Our colleagues in the physical sciences have done this for over a century: Physics I, Physics II, Physics III; Chemistry I, Chemistry II, Chemistry III.

Some examples of this breadth-first approach to Computer Science:

- We teach the rudiments of binary numbers. Is this necessary to learn to program the solution to a simple problem? Probably not, but it will become necessary at some later time, when studying hardware, or the limitations of software.

- Every introductory programming book teaches the concept of an arithmetic expression. We give a formal definition, and show the structure of an expression by placing boxes around sub-expressions. The student thinks we are teaching how to write a correct expression; we are actually teaching recursion, formal grammars, and derivation trees.

- We advise the student, "Program to an interface whenever possible". The student thinks we are teaching the correct usage of Java interfaces, but we are actually teaching object-oriented design and software engineering.

The student thinks we are teaching programming, but we are actually teaching Computer Science. Knowledge is not separated into compartments, and our curriculum should not attempt to do so.

Several topics were recently addes to this book, in order that it be used as the primary textbook for the Educational Testing Service's Advanced Placement course in Computer Science (CSA):

- More extensive discussion of arrays
- sorting
- binary search

The AP course includes a *Marine Biology* simulation case study which is not included in this version of the book.[1] The reader will notice an empty section for this case study in several chapters of this book. The author invites current instructors to participate in this project by providing these sections; those who do so would be listed as a *secondary author* or *contributor* on the title page and/or the preface, depending on the level of contribution. Those who are interested in contributing should contact the primary author at the email address shown below.

This book is an open source book. That means that not only is the pdf version available (to potential students and teachers) for free download, but that the original (LaTeX) source files are also available (to potential authors and contributors). Based on the model of open source software, open source for textbooks is a relatively new paradigm in which many authors and contributors can cooperate to produce a high quality product, for no compensation. For details on the rationale of this new paradigm, and citations for other open source textbooks, see the journal *Publishing Research Quarterly*, Vol. 30, No. 1, March 2014. The source materials and pdf files of this book are licensed with the Creative Commons NonCommercial license, which means that they may be freely used, copied, or modified, but not for financial gain.

This book is available in pdf at `rdw.rowan.edu /oer`.

The source files are available at `cs.rowan.edu/~bergmann/books`.

The author may be reached at `bergmann@rowan.edu`

Secondary Authors

Contributors

[1] The Marine Biology case study is no longer included on the AP exam. Thus, students using this book are not penalized on the exam for not having seen the case study.

Contents

Preface **i**

0 Computers and Computer Programs **1**
0.1 The CPU and machine language 1
0.2 High level languages and compilers 2
0.3 Data representation: bits and bytes 3
0.3.1 Whole numbers 3
0.3.2 Other numbers 3
0.3.3 Characters 3
0.3.4 Images 4
0.3.5 Sound 4
0.3.6 Exercises 4

1 Java classes, objects, object diagrams and methods **7**
1.1 Classes and objects 7
1.1.1 Exercises 8
1.2 Variables and references 8
1.2.1 Exercises 9
1.3 Defining a Java class 9
1.3.1 Exercises 10
1.4 Object diagrams 11
1.5 Methods 11
1.5.1 Exercises 13
1.6 Constructors and object creation 14
1.6.1 Exercises 15
1.7 Getting started: IDE or command line 16
1.7.1 Using an IDE 16
1.7.2 The BlueJ IDE 17
1.7.3 Exercises 20
1.8 Constructors and objects in the GridWorld case study 20
1.9 Projects 20

2 Program Elements and Methods (revisited) 22
2.1 Data types . . . 22
2.1.1 Whole numbers: int . . . 23
2.1.2 Other numbers: float and double . . . 23
2.1.3 Logical values: boolean . . . 24
2.1.4 Character values: char . . . 25
2.1.5 Strings of characters: String . . . 25
2.1.6 Other reference types . . . 26
2.1.7 Exercises . . . 27
2.2 Operations and expressions . . . 28
2.2.1 Arithmetic operations . . . 28
2.2.2 String operations . . . 28
2.2.3 Arithmetic Expressions . . . 30
2.2.4 Exercises . . . 33
2.3 Declaration and initialization of variables . . . 34
2.3.1 Declaration of variables . . . 35
2.3.2 Iniitialization of variables . . . 35
2.3.3 Exercises . . . 35
2.4 Assignment of values to variables . . . 35
2.4.1 Type conversion in assignments . . . 37
2.4.2 Type conversions and initializations . . . 37
2.4.3 Assignment of references . . . 38
2.4.4 Exercises . . . 39
2.5 Method definitions, signatures, and invocation . . . 40
2.5.1 Method definition . . . 40
2.5.2 Method signature and body . . . 41
2.5.3 Method invocation . . . 42
2.5.4 Methods From the Java Class Library . . . 44
2.5.5 Exercises . . . 44
2.6 Recursive methods . . . 46
2.6.1 Exercises . . . 46
2.7 Printing the output . . . 46
2.7.1 Exercises . . . 47
2.8 Constants and class variables . . . 47
2.8.1 Constants . . . 47
2.8.2 Class variables . . . 48
2.8.3 Class constants . . . 48
2.8.4 Exercises . . . 49
2.9 Comments and readability . . . 49
2.9.1 Formatting a program . . . 49
2.9.2 Comments . . . 50
2.9.3 Exercises . . . 50
2.10 Program elements and methods in the GridWorld case study . . . 51

3 Selection Structures **52**
3.1 Comparison operators . . . 52
3.1.1 Exercises . . . 53
3.2 Boolean operators . . . 53
3.2.1 AND, OR, NOT . . . 53
3.2.2 Short circuit evaluation . . . 56
3.2.3 De Morgan's Laws . . . 56
3.2.4 Exercises . . . 57
3.3 One-way selections . . . 58
3.3.1 Exercises . . . 59
3.4 Two-way selections . . . 60
3.4.1 Exercises . . . 64
3.5 Compound statements and scope . . . 66
3.5.1 Compound statements . . . 66
3.5.2 Scope of variables . . . 67
3.5.3 Java statements - revisiting a formal definition . . . 68
3.5.4 Exercises . . . 68
3.6 Recursive methods revisited . . . 70
3.6.1 Exercises . . . 71
3.7 Comparing Strings and other reference types . . . 73
3.7.1 Comparison for equality or inequality . . . 73
3.7.2 Ordered comparisons . . . 73
3.7.3 Exercises . . . 75
3.8 Selection structures in the GridWorld case study . . . 76
3.9 Projects . . . 76

4 Iteration Structures **80**
4.1 Looping with `while`– pre-test loops . . . 80
4.1.1 Infinite loops . . . 83
4.1.2 Exercises . . . 83
4.2 Looping with `for` – counter-controlled loops . . . 85
4.2.1 Autoincrement and autodecrement . . . 85
4.2.2 The for loop . . . 86
4.2.3 Exercises . . . 88
4.3 Equivalence of `while` and `for` loops . . . 89
4.3.1 Exercises . . . 90
4.4 Nested loops . . . 91
4.4.1 Exercises . . . 91
4.5 Definition of Statement - updated . . . 93
4.5.1 Exercises . . . 94
4.6 Iterations in the GridWorld case study . . . 95
4.7 Projects . . . 95

5 Collections, and Iteration Revisited 97
5.1 Lists . . . 97
5.1.1 Java packages and java.util . . . 98
5.1.2 ArrayList . . . 98
5.1.3 Exercises . . . 105
5.2 Iteration revisited, with lists . . . 107
5.2.1 Exercises . . . 108
5.3 Sets . . . 109
5.3.1 Exercises . . . 111
5.4 Iteration through a collection with for-each, and extrema problems 113
5.4.1 Iteration through a collection with for-each . . . 113
5.4.2 Extrema problems . . . 114
5.4.3 Exercises . . . 116
5.5 Iterators and selective removal from a collection . . . 117
5.5.1 Iterators . . . 117
5.5.2 Selective removal . . . 118
5.5.3 Exercises . . . 119
5.6 Arrays . . . 121
5.6.1 Initialization of arrays . . . 123
5.6.2 Passing arrays as parameters . . . 124
5.6.3 Vector product of numeric arrays . . . 124
5.6.4 Exercises . . . 125
5.7 Matrices: Two Dimensional Arrays . . . 126
5.7.1 Examples of Matrix Arithmetic . . . 127
5.7.2 Exercises . . . 130
5.8 Collections in the GridWorld case study . . . 133
5.9 Projects . . . 133

6 Abstraction, Inheritance, and Polymorphism 137
6.1 Software engineering . . . 137
6.2 Abstraction . . . 138
6.2.1 Duplicated code . . . 138
6.2.2 Method abstraction . . . 140
6.2.3 Object abstraction and encapsulation . . . 143
6.2.4 Exercises . . . 144
6.3 Inheritance . . . 145
6.3.1 Is-a versus Has-a . . . 149
6.3.2 Factoring duplicated code and defining subclasses . . . 150
6.3.3 Making use of inheritance . . . 155
6.3.4 Exercises . . . 158
6.4 Polymorphism and dynamic method look-up . . . 160
6.4.1 Dynamic method look-up . . . 160
6.4.2 Polymorphism . . . 161
6.4.3 Exercises . . . 162
6.5 Overriding methods from the Object class . . . 164
6.5.1 Overriding the `toString()` method . . . 164

6.5.2 Exercises . . . 166
6.6 Abstract methods and classes . . . 167
6.6.1 Abstract methods . . . 167
6.6.2 Abstract classes . . . 168
6.6.3 Exercises . . . 168
6.7 Java Interfaces . . . 170
6.7.1 The need for Java interfaces – multiple inheritance . . . 171
6.7.2 Interfaces which we've already been using . . . 173
6.7.3 Exercises . . . 174
6.8 Inheritance and Polymorphism in the GridWorld case study . . . 176
6.9 Projects . . . 176

7 Maps, Collections Revisited 180
7.1 Fast look-up . . . 180
7.1.1 Exercises . . . 180
7.2 Sequential search . . . 181
7.2.1 Exercises . . . 182
7.3 Java maps . . . 182
7.3.1 Exercises . . . 186
7.4 Examples of methods which use maps . . . 187
7.4.1 Exercises . . . 188
7.5 Instantiating maps . . . 189
7.5.1 HashMap . . . 189
7.5.2 Exercises . . . 193
7.6 TreeMap and Collections revisited: TreeSet and LinkedList . . . 195
7.6.1 TreeSets . . . 195
7.6.2 TreeMaps . . . 197
7.6.3 LinkedList . . . 198
7.6.4 Exercises . . . 200
7.7 Projects . . . 201

8 Exceptions - Handling Errors 205
8.1 Client/Server terminology . . . 206
8.1.1 Exercises . . . 207
8.2 Assertions . . . 207
8.2.1 Exercises . . . 210
8.3 Exceptions . . . 210
8.3.1 Run-time errors resulting in an Exception . . . 210
8.3.2 Throwing exceptions in a server method . . . 212
8.3.3 What to do when an Exception is thrown . . . 213
8.3.4 Handling exceptions with try/catch in a client method . . . 214
8.3.5 Defining your own Exception classes . . . 218
8.3.6 Exercises . . . 221
8.4 Debuggers . . . 223
8.4.1 Exercises . . . 224
8.5 Debugging with print statements . . . 225

8.6 Projects 225

9 Console Applications – Input and Output 226
9.1 Standard io files 226
9.1.1 Exercises 227
9.2 Output to stdout or stderr 228
9.2.1 Output to stdout 228
9.2.2 Output to stderr 228
9.2.3 Exercises 228
9.3 Input from stdin 229
9.3.1 Exercises 230
9.4 Data Files 230
9.4.1 Opening a data file 231
9.4.2 Input from Data Files 231
9.4.3 Output to data files 232
9.4.4 Exercises 234
9.5 Running an Application from the Command Line 234
9.5.1 Compile and Test from the Command Line 235
9.5.2 public static void main (String [ ] args) 235
9.5.3 Exercises 236
9.6 Projects 237

10 Graphical User Interfaces 240
10.1 Packages java.awt and javax.swing 241
10.1.1 Exercises 241
10.2 Starting out: Frame and ContentPane 242
10.2.1 Exercises 244
10.3 Adding components to a container 244
10.3.1 Designing the GUI 244
10.3.2 Adding components 245
10.3.3 Exercises 246
10.4 Layout managers 246
10.4.1 Flow Layout 247
10.4.2 Grid Layout 247
10.4.3 Border Layout 249
10.4.4 Nested containers and summary of layout managers . . . 250
10.4.5 University Information System - version 1 251
10.4.6 Exercises 252
10.5 Actions and Listeners 255
10.5.1 University Information System - version 2 257
10.5.2 Exercises 258
10.6 Menus 259
10.6.1 Adding menus to the frame 259
10.6.2 Listening for menu selection 261
10.6.3 Menus for the University Information System - version 3 . 261
10.6.4 Exercises 262

10.7 Projects 263

11 Abstract Data Types **265**
11.1 The Rational ADT 265
11.1.1 Some problems with float and double 265
11.1.2 Defining the Rational ADT 266
11.1.3 Exercises 270
11.2 MyFloat 272
11.2.1 Constructor for MyFloat 273
11.2.2 Arithmetic operations for MyFloat 274
11.2.3 Exercises 277
11.3 BigNumber 280
11.3.1 Constructing BigNumbers 280
11.3.2 Adding BigNumbers 281
11.3.3 Subtracting BigNumbers 282
11.3.4 Exercises 287

12 Algorithms: Sorting and Searching **291**
12.1 Searching: Binary Search 292
12.1.1 Exercises 296
12.2 Sorting a list 297
12.2.1 Rationale for Sorting 297
12.2.2 Selection Sort Algorithm 298
12.2.3 Insertion Sort Algorithm 298
12.2.4 Merge Sort Algorithm 302
12.2.5 Exercises 309

Glossary **314**

Chapter 0

Computers and Computer Programs

The first digital computers were developed in the early 1950's. Since then the capabilities of computers (and computer science) have increased at an amazing rate. It has been said that if the automotive industry had seen similar progress, that a Rolls Royce would get 3,000 miles per gallon, would have a top speed approaching the speed of light, and would cost less than one dollar. Various monikers for this amazing development include "The Digital Revolution" and the "Information Age". When we use the word *digital* we are talking about information which is composed of discrete atomic (i.e. having no sub-cmponents) values, generally described as 0 or 1. Anything which is said to be digital, at its innermost level, is nothing but lots of 0's and 1's. This includes not only computers but telephones, cameras, music boxes, televisions, and the list goes on. Today there are digital components in automobiles, household appliances, roads, bridges, medical devices, medicines, and even people.

This book is an introduction to some of the things we have learned about developing computer software. Software is the driving force in a computer system; without software, computer hardware is not capable of carrying out the simplest of tasks. In the process of learning to program a computer in a popular language known as Java, we hope to expose many of the concepts and principles which apply to the development of software in any language.

Computer Science has been defined as "the design, analysis, and implementation of algorithms", and in this book we introduce the notion of *algorithm* in the hope that this will spark the student's interest for further study of Computer Science.

0.1 The CPU and machine language

A computer system is made up of hardware and software. The hardware consists of the physical components:

- Semiconductor chips (CPU, memory, communications, etc)
- Wires and other conductors connecting the components
- Storage devices such as flash memory and disks
- Peripheral devices such as keyboards, mice, displays

The most fundamental hardware component is the *Central Processing Unit* or CPU. This is the component which is capable of performing calculations and making decisions. The CPU is capable of executing only instructions which are coded in a binary format (consisting of only 0's and 1's). These binary instructions, when stored in the computer's memory, constitue a *program*. The language of these binary instructions is called *machine language*

Consequently, if we write a 'program' in Java, it is not a program in the strict sense of the word, because it is not written in machine language.

0.2 High level languages and compilers

If we were to write our programs in machine language, the rate of software development would be slow; the binary coded instructions of machine language make it exceedingly difficult for use by humans. For this reason we have developed *high-level languages*, also known as *programming languages* which are much easier to use for programming than machine language. Some examples of high-level languages are:

- Java
- C++
- C
- Visual Basic
- Python
- Ruby

However, the CPU is not capable of executing the statements of a high-level language directly, so it must first be translated into machine language. This is done by a program known as a *compiler*. The compiler will examine the statements in a program, check for syntax errors, and produce output consisting of binary machine language instructions which the CPU is capable of executing.

In later chapters we speak of 'compile time' versus 'run time'. An error which is detected by the compiler is a *compile-time* error, whereas an error which occurs when the machine language program is executing is a *run-time* error.

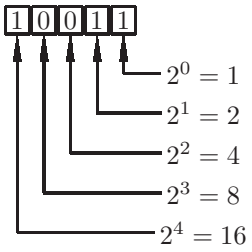

Figure 1: The binary representation of 19 (19 = 16 + 2 + 1)

0.3 Data representation: bits and bytes

As noted above all digital devices, including computers, store information in binary (0's and 1's). Each such binary digit is called a *bit*. This means that in order for us to represent information, everything must be encoded in binary; this includes not only numbers, but characters from the keyboard, sound, images, colors, video clips, ... everything. In this section we offer some insight as to how this is done. Further details on these data representation schemes are given in later chapters.

0.3.1 Whole numbers

To represent a whole number we use base two. Whereas in a base ten (decimal) number each digit represents a power of 10, in base two each digit represents a power of 2, as shown in Fig 1 which shows the base two representation of 19.

Whole numbers may also be negative; to represent a negative whole number we use *two's complement representation*. In this scheme numbers, positive or negative, can be easily added, always producing the correct result. We describe two's complement in more detail in chapter 2

0.3.2 Other numbers

Numbers which are not whole numbers (or are too big to be stored as simple binary values) are stored in floating point format. This means that associated with each number is an exponent to magnify (or diminish) the value. This kind of number is similar to scientific notation: $6.02x10^{23}$

0.3.3 Characters

Any character from the keyboard (and others) can be represented with a binary code. This includes the letters (a..z, A..Z) the numbers (0..9) and other characters such as `$!@#%^&`. A binary code is assigned to each character. A 16-bit code with characters from international alphabets is called Unicode. An older code, using an 8-bit (byte) code called ASCII is a subcode of Unicode.

0.3.4 Images

A digital image is like a newspaper photograph, which consists of a matrix of discrete dots. Each such dot is called a *pixel*, or picture element. For color images each pixel is simply a whole number representing the red, green, and blue components of the desired color.

0.3.5 Sound

Sound is made up of air pressure waves; when these waves enter our ears, our brain receives a signal from the auditory nerves. To represent sound, all we need do is store a digital version of the pressure waves. This is diagrammed in Fig 2, in which the horizontal axis is time, and the vertical axis is the amplitude of the pressure waves. The wave at top left of Fig 2 represents just one cycle of a sound wave. The wave at top right represents a louder sound at the same pitch because the amplitude is greater, but the frequency of the cycles is the same. The wave at bottom left represents a higher pitch, because there are twice as many cycles in the same time period; the loudness is the same as the sound at top left. The quality of the sound is determined by the number of values sampled per unit time. To represent sound with very high quality (high fidelity) requires many numbers. Consequently a sound clip is merely a sequence of whole numbers representing varying air pressure. Most sound clips are actually a compressed format of these numbers. Some examples of compression formats are .wav and .mp3.

0.3.6 Exercises

1. Show the following decimal numbers in base two (binary):
   (a) 7
   (b) 23
   (c) 123
   (d) 127
   (e) 255
   (f) 256
2. Read parts (a) and (b) below aloud so that they make sense.
   (a) There are 10 kinds of people in the world: those who know binary and those who do not know binary.
   (b) There are 10 kinds of people in the world: those who know base three, those who do not know base three, and those who have no idea what I am talking about.
   (c) Make up a similar statement for some number base greater than three but less than 10.

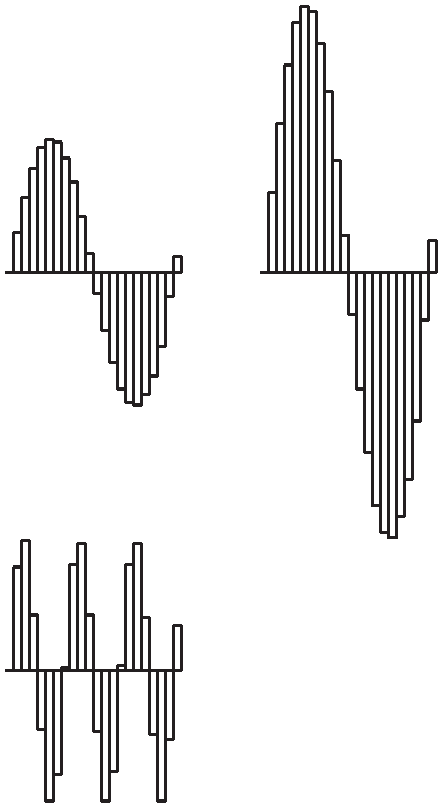

Figure 2: Representing sound (i.e. air pressure waves) as discrete numbers. Top right: a louder sound. Bottom left: a higher pitch.

3. Show your friends how to count from 0 to 31 using only the fingers on one hand (tell them not to be offended when you get to 4).

4. Do an internet search to find the ASCII code for each of the following characters:

   (a) 'a'

   (b) 'A'

   (c) '8'

   (d) '('

5. How many pixels are there in a 8x8 square inch display which has 256 pixels per inch? Hint: Use powers of 2. $256 = 2^8$.

6. Do an internet search to find sound data compression formats other than .wav and .mp3.

Chapter 1

Java classes, objects, object diagrams and methods

1.1 Classes and objects

Computer programs usually deal with concepts and problems selected from our common environment and/or experiences. In order to represent these concepts in software, the Java language introduces the concept of *class*. A class is like a template, or an archtect's blueprint; it allows one to specify the attributes and behavior of objects. A class is the plan from which objects can be created. Hence, an *object* is merely an instance of a class.

For example, we could have a class named `Student` which specifies all the attributes of a student. These attributes could be things such as the student's name, social security number, and gpa. A diagram for a particular Student object is shown in Figure 1.1.

The values of the object's attributes are collectively known as the *state* of the object.

Classes can also specify behavior of the objects. For example:

- We may ask a Student object to provide us with his/her name, gpa, or social security number.

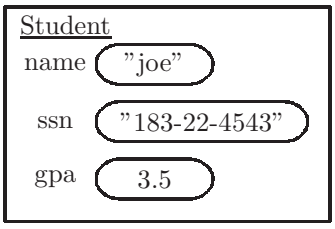

Figure 1.1: An object diagram showing an instance of the class `Student`

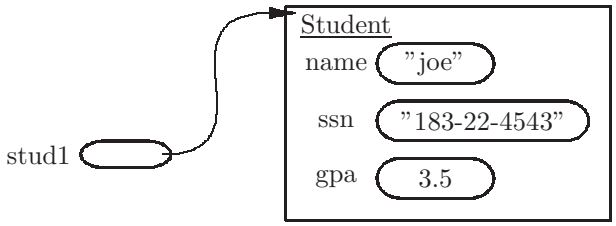

Figure 1.2: An object diagram showing the value of the variable `stud1` as a reference to an object

- We may wish to change a Student's gpa.
- We may allow a Student object to register for courses (which could entail including a list of those courses as another attribute for Students), and do any number of things that students in the real world normally do.

This behavior is specified with *methods* (more on this later). The compiler will permit a class name to begin with a lowercase letter, however we will follow the convention that all class names must begin with an *uppercase* letter.

1.1.1 Exercises

1. Show an object diagram for a Student whose name is "mary", and whose ssn is "999-99-9999" and whose gpa is 3.8.

2. Assume there is a class named `University` which stores a name and a size (number of students enrolled). Draw an object diagram showing an instance of this class; make up values for the state of this object.

1.2 Variables and references

A Java program may have many variables, and as we'll see later, there are various kinds of variables. For now, we'll define a *variable* as a symbol which can store a value. An example would be the symbol `gpa` in the Student class. This kind of variable is called an *instance variable*, or *field*, because it is part of an object which is an instance of a class. Figure 1.1 shows that the value of the variable `gpa` is 3.5.

Variables can also store references to objects. A *reference* is merely an indication of where an object can be found in the computer's memory. Figure 1.2 shows a variable named `stud1` which stores a reference to a Student object. The compiler will permit a variable name to begin with an uppercase letter; however we will follow the convention that all variable names begin with a *lowercase* letter.

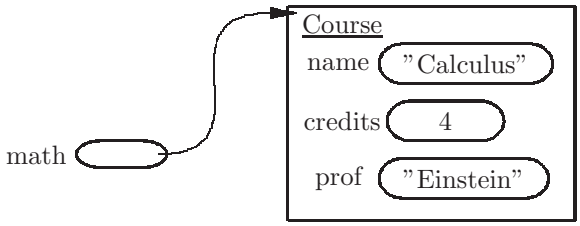

Figure 1.3: An object diagram

1.2.1 Exercises

1. Refer to Figure 1.3.
   (a) What is the name of the object's *class*?
   (b) What are the names of the *fields* in the object?
   (c) A *reference* to the object is stored in which variable?

2. Show an object diagram for a variable named `univ` which stores a reference to a University object whose name is "Southern State" and whose size is 12000 students.

1.3 Defining a Java class

Figure 1.2 is an example of an *object diagram*. It shows that a variable named `stud1` stores a reference to an object which is an instance of the class `Student`. It also shows all the fields (i.e. instance variables) and their values. Note that the value of a variable is always shown in a rounded rectangle, whereas objects are shown in ordinary rectangles.

The value of the variable `stud1` in Figure 1.2 is a reference to a Student object. This value is depicted with an arrow in the diagram. More accurately, the reference is a memory address, or location of the object in memory.

In a Java program, we can specify the name of a class, and its fields, as shown in Figure 1.4. This class definition begins with the key words `public class`, and contains the field definitions within a set of curly braces. We note that:

- The word `public` is used to indicate that the class being defined can be accessed from any other class. The word `class` is used to indicate that what follows is the definition of a class. The words `public` and `class` are *key words*, which means that they can be used only for these purposes (i.e. they cannot be used as variable names).

- After the name of the class there is an open brace.

- This is followed by a definition of each field in the class.

```
public class Student
{  // fields
   private String name;
   private String ssn;
   private double gpa;
}
```

Figure 1.4: Java code defining a class named `Student` which has three fields

- Each field has a type (such as `String` or `double` - more on this later).
- The class is *public*, but the fields are *private*. This is called *visibility* (more on this later).
- The class definition ends with a closed brace.

We also note that:

- Java is *free format*. This means that we are free to include spaces and new lines anywhere in a java program. We will generally try to include spaces so as to make the program easy to read. Conceivably, Figure 1.4 could have been written entirely on one or two lines:

  ```
  public class Student {private String name;private
  String ssn;private double gpa;}
  ```

  Though the Java compiler would allow this, it is not considered good style, and we will avoid it.
- Java is *case sensitive*. This means that you must pay attention to upper case versus lower case letters. The word `class` is different from the word `Class`.

1.3.1 Exercises

1. Show the Java code to define a class named `University`; it should have two fields: `name`, which is a String, and `size`, which is an int (i.e. integer).
2. Refer to Fig 1.4.
   (a) How many fields are in the class defined as Student?
   (b) What is the *type* of the field `ssn`?
   (c) What is the *type* of the field `gpa`?
3. Each class definition shown below contains, at most, one error; find the error and correct it if there is one.

(a)
```
public Class Car
{  private double cost;
   private String make;
   private String model;
}
```

(b)
```
public class Vehicle
{  private double
   cost; private
   String make; private String model; }
```

(c)
```
public class Vehicle
   private double cost;
   private String make;
   private String model;
   private int wheels;
```

4. Show the Java code to define a class named `Ticket` with three fields:

   - A section (type is String)
   - A row number (type is int)
   - A seat number (type is int)

1.4 Object diagrams

This book will make extensive use of object diagrams, similar to the one shown in Figure 1.2. The concept of an object diagram is one that is vital to a good understanding of Java programs and more advanced concepts in computer science, such as data structures.

In an object diagram a reference (i.e. an arrow) will *always* refer to an object, and *never* to another variable. As we will see later, the value of an object's field may be a reference to another object.

1.5 Methods

The fields of a class specify the attributes, or collectively, the state of an object of that class. Objects can also have behavior; the behavior of an object is specified in the class with *methods*, or more properly *instance methods*[1].

A method consists of a signature and a body. The *signature* defines how the method is to be invoked, and the body defines exactly what it does. An example of a method signature in the Student class could be

[1] C++ programmers would call these *member functions*.

```
public String getName()
```

- The visibility of this method is `public`. Methods may also be `private` - more on this later.
- This method will *return* a String.
- The name of this method is `getName`. The compiler will permit a method name to begin with an uppercase letter; however, we will follow the convetion that every method name begins with a *lowercase* letter.
- This method has no *parameters*, though the parentheses are always required in the signature.

The method *body* consists of one or more Java statements enclosed in curly braces. In the `getName` method, the body could be:

```
{  return name;  }
```

The *return* statement defines what the method produces as a result when it is invoked. It also terminates the method.

One more important feature (but not required by the compiler) is a group of *comments* which help to describe the purpose and correct usage of the method. These comments are ignored by the compiler, but can be very useful to us humans as we attempt to define and use methods. These comments begin with /** and end with */ . Comments of this form are used by a utility program, `javadoc` to generate nicely formatted documentation for a class and its methods. We call this documentation an Application Program Interface, or API. The complete method definition, with comments, is shown below:

```
/** This method returns the name of this Student. */
public String getName()
{
   return name;
}
```

Methods do not have to return any value (these are called *void* methods. Also a method may have one or more *parameters*, specified inside the parentheses in the signature. These are used to pass information into the method when it is invoked.

```
/** This method changes the name of this Student. */
public void setName(String newName)
{
   name = newName;
}
```

The `setName` method has one parameter, whose value is the new name for a Student. It is a `void` method because it produces no explicit result.

We can now show the class definition, with fields and methods, in Figure 1.5

```
public class Student
{  // fields
   private String name;
   private String ssn;
   private double gpa;

   /** This method returns the name of this Student. */
   public String getName()
   {  return name;  }

   /** This method changes the name of this Student. */
   public void setName (String newName)
   {  name = newName;  }

}
```

Figure 1.5: Java code defining a class named `Student` which has three fields and two methods

1.5.1 Exercises

1. Refer to the following class defining a `University`

   ```
   public class Unviersity
   {  private String name;
      private int size;
   }
   ```

   (a) Include a method definition in this class to return the University's name.

   (b) Include a method definition in this class to change the size of the University to a given nuumber of students.

2. Refer to Figure 1.4. Each of the following method definitions contains, at most, one error. Find and correct the error if there is one.

   (a)
   ```
   public getGPA()
   {  return gpa;  }
   ```

   (b)
   ```
   public void clearGPA()
   {  gpa = 0.0;  }
   ```

   (c)
   ```
   public void setGPA(newGPA)
   {  gpa = newGPA;  }
   ```

1.6 Constructors and object creation

In this section we discuss how objects can be created. Java has an operator called *new* whose sole job is to create new objects. When the `new` operator is invoked, in the process of creating the object, a *constructor* is called to initialize the fields in the object. The constructor is a peculiar kind of method with the following properties:

- The name of the constructor is the same as the name of its class.
- The constructor has no return type *not even void*.

The student object shown in Figure 1.1 can be created as shown below:

```
new Student ("joe", "183-22-4543");
```

The values "joe" and "183-22-4543" are *actual parameters* for the constructor. They are the initial values of two of the fields in the Student object being created.

The constructor, defined as one of the methods in the Student class, is shown below:

```
/** Constructor.
    Initialize this Student's name, ssn, and gpa
 */
public Student (String initialName, String initialSSN)
{  name = initialName;
   ssn = initialSSN;
   gpa = 0.0;
}
```

Note that when creating a new Student, no initial gpa is provided. Instead the constructor initializes gpa to 0.0 for all new students.

The parameters in the constructor could have been the same as the fields which they are initializing. In this case we need to distinguish between the field and the parameter using the key word *this* as shown below:

```
/** Constructor.
    Initialize this Student's name, ssn, and gpa
 */
public Student (String name, String ssn)
{  this.name = name;     // assign parameter value to field
   this.ssn =  ssn;      // assign parameter value to field
   gpa = 0.0;
}
```

We now have three kinds of entities in a class definition:

1. private field(s)

```
public class Student
{  // fields
   private String name;
   private String ssn;
   private double gpa;

   /** Constructor.
       Initialize this Student's name, ssn, and gpa
    */
   public Student (String initialName, String initialSSN)
   {  name = initialName;
      ssn  = initialSSN;
      gpa  = 0.0;
   }

   /** This method returns the name of this Student. */
   public String getName()
   {  return name;  }

   /** This method changes the name of this Student. */
   public void setName (String newName)
   {  name = newName;  }

}
```

Figure 1.6: Java code defining a class named `Student` which has three fields, one constructor, and two methods

2. public constructor(s)
3. public and/or private method(s)

These entities can be placed in the class definition in any order, but we usually conform to the order shown above. Our updated class definition is now shown in Figure 1.6.

1.6.1 Exercises

1. Show a *constructor* for the `University` class which will initialize the `name` and `size` fields from constructor parameters (see Exercises above).
2. Each of the following constructor definitions for the Student class contains, at most, one error. Correct each error, if there is one.

   (a)
   ```
   public void Student ()
   {  name = "joe";
   ```

```
      ssn = "222";
   }
```

(b)
```
public StudentConstructor ()
{  name = "joe";
   ssn = "222";
}
```

(c)
```
public Student (String name, String ssn)
{  name = name ;
   ssn = ssn  ;
}
```

3. Show a constructor for the Student class with three parameters. The parameter names should be `name`, `ssn`, and `gpa`. Each parameter should be used to initialize the corresponding field.

1.7 Getting started: IDE or command line

Having seen some of the rudiments of defining a class, we can now see how to work with these concepts on the computer. There are two basic ways to create and test Java programs:

- Use an *Integrated Development Environment* or IDE.
- Run the java compiler and its runtime environment from the command line prompt. This is covered in chapter 9.

Most beginning users will prefer to use a simple IDE, such as BlueJ, but other IDEs are available for free download on the internet.

1.7.1 Using an IDE

Some examples of IDEs which can be used to develop Java software are:

- BlueJ - A fairly simple, yet powerful, IDE which can be used to edit and execute Java source files, and inspect objects. It also allows the user to test code snippets to find out what effect they have. BlueJ is available for free download at `www.BlueJ.org`.
- NetBeans - This IDE began as a student project in Czechoslavakia and was later acquired by Sun Microsystems (now Oracle). Like BlueJ it is designed specifically for Java development. It differs from BlueJ in that it has tools specifically for the automatic creation of graphical user interfaces.
- Eclipse - This IDE has so many features that it is rather difficult to learn. The main advantage of Eclipse is that it can be used for many different programming languages, not just Java. Once you learn Eclipse, you can use it on many different projects. Another interesting feature of Eclipse is that the debugger can *step backwards*.

All three of these IDEs are free for download on the Internet; in addition, they are all open source and have many optional plug-in features.

1.7.2 The BlueJ IDE

Since BlueJ is the easiest IDE for novices to learn, it is the one we choose to look at here. After learning BlueJ, many users will opt for a more powerful IDE at a later time.

1.7.2.1 Getting started with BlueJ

The basic development unit in BlueJ is the *project*. A project may consist of several related classes (a project is really a folder, or directory, containing a source file and related files for each class in the project). When starting up BlueJ, use the menu to open a new project, or to open an existing project. Be sure to save the project to a disk or storage device for which you have write access. As you make changes to your source files, BlueJ will automatically save the project.

To create a new class for your project, click the button `New Class...`. This will bring up a dialog box allowing you to enter the name of the class (the name should begin with an uppercase letter) such as *Student*. There will now be a file with suffix `.java` in your project folder, such as `Student.java`.

An example of a BlueJ project window is shown in Figure 1.7 in which the project has three classes, and one Student object has been created.

To edit a class, double click on the (tan) icon for the class. This brings up the BlueJ editor window, showing the lines of Java code in the class. BlueJ will provide a template of sorts for you to get started; you will want to modify or delete most of what BlueJ has provided. Note that Java source files are plain text documents (like Notepad documents), and the BlueJ editor is nothing more than a text editor (like Notepad for Windows or TextEdit for MacOS).

To test your work two steps are needed:

1. Compile - Translate your source code to 'byte code', a language that can be understood by the Java runtime environment. If there are syntactic errors in your class, the compiler will advise you of these. You must correct these errors before going further. BlueJ will draw hashmarks on classes which need to be compiled. If there are no syntactic errors, the compiler will create the byte code file, with a `.class` suffix, such as `Student.class`. Do not try to examine a `.class` file; it will be unintelligible and is strictly for Java's use.

2. Test - Execute the byte code by invoking a method in one of your classes. Right-click on a class:

   - If the class has a static method, select it to run it directly from the selected class. If that method has parameter(s), a dialog box will be opened allowing you to enter the parameter value(s).

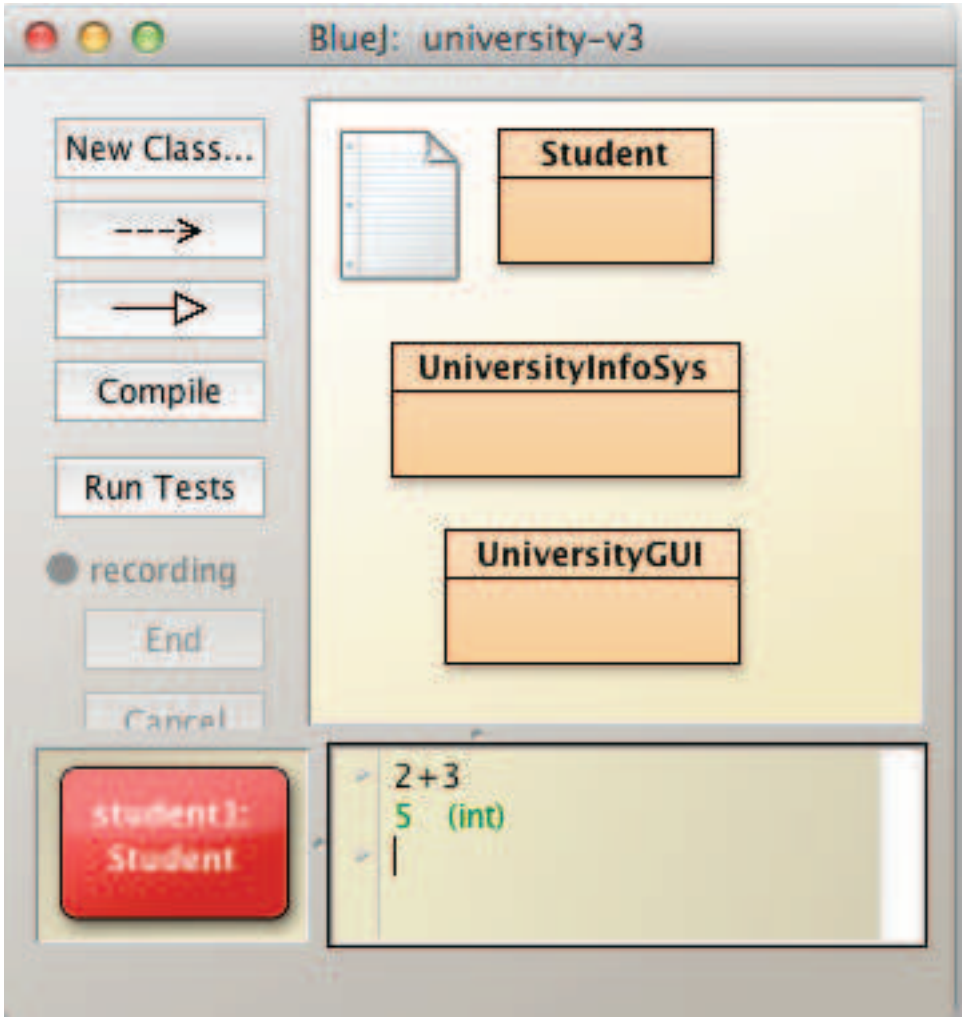

Figure 1.7: An example of a BlueJ project with three classes. One object of class Student has been instantiated. The CodePad shows that the value of the expression `2+3` is `5`

- Otherwise, instantiate a class (i.e. create an object of the class) by selecting `new ...`. If the class' constructor has parameters, a dialog box will allow you to enter the parameter value(s) at this point (caution: remember the double quote-marks if the parameter is a String). The object should appear as a red box in the object window at the lower left. Right-click on the red object to inspect it (look at the values of its fields) or to invoke an instance method. If the method has parameters, a dialog box will allow you to enter values for the parameters (caution: remember the double quote-marks if the parameter is a String).

1.7.2.2 The BlueJ Terminal Window

If an executing method produces output, it will be sent to the BlueJ Terminal Window. Some options available for the terminal window include:

- Unlimited Buffering. The window acquires a scroll bar on the right, and will retain an unlimited amount of output. Use the scroll bar to view any output.
- Clear. Clear the output window so that you do not confuse the output of multiple executions.
- Clear Screen at Method Call. Clear the screen automatically each time BlueJ starts up a method. This is often the preferred option.
- Record Method Calls. Show all methods which have been invoked from BlueJ.
- Save to file... Open a dialog box to allow all output to be saved in a text file.

1.7.2.3 The BlueJ Debugger

A *debugger* is a tool designed to help the programmer locate the source of a logic error in the program. A debugger will not tell you where to find a bug, nor will it correct the mistake for you; this is the programmer's job. With a debugger, the programmer can step throught the statements of a program, one statement at a time, while watching the values of variable change. This kind of tool is often essential in diagnosing an error. Debuggers are used at runtime, not at compile time.

To start up the debugger, open a BlueJ Editor window, and click the mouse in the left margin (on a line number). You should see a red stop sign appear; this is known as a break point. When execution reaches this point, it will pause and wait for you to direct the debugger to continue execution, either single-step, or at full speed.

More details on the use of the BlueJ debugger are provided in chapter 8.

1.7.2.4 The BlueJ Codepad and inspections

The BlueJ Codepad is in the lower right corner of the project window. The Codepad can be used to execute small code snippets to see how they work. BlueJ will invoke the compiler and runtime environment to evaluate expressions as you enter them in the Codepad. Figure 1.7 shows the expression `2 + 3` in the Codepad, along with its result, `5`. It is possible to instantiate classes in the Codepad and inspect the objects which are created.

Any object on the object bench (lower left corner) can be inspected by right-clicking on the object and selecting `inspect`. You will see the values of all fields; if any are references to objects, you can select the reference and inspect the object to which it refers.

1.7.3 Exercises

1. What is the full name of the source file for the University class?
2. What will be the name of the output file when compiling the University class?
3. True or False: If the compiler produces no error messages, your program must be correct.
4. Which of the IDEs mentioned in this section have a debugger?

1.8 Constructors and objects in the GridWorld case study

1.9 Projects

1. Define a class named Course which is to store information on a university course. Each Course object should store:

   - A course title (a String)
   - The number of credits for the course (an int)
   - The name of the the prof teaching the course (a String)
   - Course level – `"Grad"` or `"Undergrad"`

   Your class should have public methods which provide access to each of the fields and public methods which allow changes to each of the fields. Your class should have two constructors, one with three parameters and one with four parameters; the constructor with three parameters should assume the course is an undergrad course. Each constructor should initialize all four fields.

   Test your solution:

(a) Compile the class; it should compile without error messages

(b) Instantiate the Student class (i.e. create a Student object) using the constructor with three parameters.

(c) Instantiate the Student class (i.e. create a Student object) using the constructor with four parameters.

(d) Inspect both objects (if your IDE will permit this) and make sure the values of the fields are correct.

Chapter 2

Program Elements and Methods (revisited)

2.1 Data types

Java permits several kinds of data for use in a program. As described in chapter 1, each data value can be represented by a sequence of bits (0's and 1's). In this section we will explore some of the primitive data types available. In each case we will take a quick peek at its binary representation.

All data types can be classified as either *primitive* or *reference* types (reference types are also known as *object* types). An easy way to distinguish these is that all pirmitive types begin with a lower-case letter, whereas reference types begin with an upper case letter (because they are class names). Examples of data types are shown in Figure 2.1 In this section we explore some of the primitive types, and a few reference types.

Primitive types	Reference types
int	String
float	Student
double	System
char	
boolean	
byte	
short	
long	

Figure 2.1: Examples of data types. All primitive types begin with a lower-case letter. All reference types begin with an upper-case letter.

2.1.1 Whole numbers: int

One of the simplest primitive data types, *int*, is used to represent (positive or negative) whole numbers. Possible values that can be represented by an int are 124, 0, -9833, and 248888. Note that 2.43, 0.05, and even 3.0 are *not* ints. int values do not contain a decimal point.

Figure 2.2 shows how the values -8 through +7 can be represented using only 4 binary digits. (These are the only values that can be represented with 4 bits, because there are $2^4 = 16$ different patterns of four bits.) This is called *twos complement* representation. Note that:

- There are more negative numbers than there are positive numbers.
- All negative values begin with a 1.
- If a value begins with 0, it may or may not be positive.
- Odd numbers end with a 1, and even numbers end with a 0.
- -1 is represented by all ones. This makes sense; consider an automobile odometer of a new car (circa 1970), initially at 000000. If you back up the car one mile, the odometer will read 999999.
- You can add and subtract values, and throw away the bit carried out of the high order digit.

```
   0101 = +5
  +1110 = -2
------------
   0011 = +3
```

In Java, the int data type is actually 32 bits in length, which places an upper and lower bound on the magnitude of an int.

2.1.2 Other numbers: float and double

In programs where we need to work with numbers other than whole numbers, or with numbers that are very large or very close to zero, there are two data types available: *float* and *double*. Examples of floats (actually doubles) are: 2.43, 0.001, -34334.1, and 3.0

With these data types we must include the decimal point. It is also possible to specify very large numbers and numbers that are very close to zero, using a notation similar to the scientific notation used in your science classes. A value can include an exponent of 10, written after an e or E:

2.04e5 = $2.04x10^5$ = 204000

6.02e23 = $6.02x10^{23}$ (a very large number)

int value	binary
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
+1	0001
+2	0010
+3	0011
+4	0100
+5	0101
+6	0110
+7	0111

Figure 2.2: Representation of int values using a 4-bit word.

-1.0e-53 = $-1.0x10^{-53}$ (very close to zero)
24.03E-2 = $24.03x10^{-2} = 0.2403$
Note that:

- Each value consists of two parts, which we call the *mantissa* and *exponent*.
- These two parts are separated by an e or E.
- The mantissa and exponent can each be positive or negative.
- The exponent implies an exponent of 10.
- If the exponent is positive, slide the decimal point to the right the given number of places.
- If the exponent is negative, slide the decimal point to the left the given number of places.

The data type *double* is the same as float, but it permits more precision. We will generally use double rather than float, simply because the cost (in memory or processing time) is not excessive. Constant values, such as 1.3e-2, are actually stored as doubles.

2.1.3 Logical values: boolean

Perhaps the simplest primitive data type is *boolean*. There are only two boolean values: `true` and `false`.

This data type will be used extensively in chapters 3 and 4.

char value	binary	decimal
'A'	01000001	65
'B'	01000010	66
'C'	01000011	67
'Z'	01011010	90
'a'	01100001	97
'b'	01100010	98
'z'	01111010	122
'0'	00110000	48
'1'	00110001	49
'2'	00110010	50
'9'	00111001	57
'\$'	00100100	36
'\#'	00100011	35

Figure 2.3: ASCII codes for some common characters.

2.1.4 Character values: char

We will need to work with data other than numbers in our programs, specifically data made up of alphabetic characters and other characters (such as $, %, +, ...). Any character which can be typed on the keyboard (and other characters) can be represented by the data type *char*. Examples of chars are 'a', 'T', '8', '*', '!'. Note that each value is enclosed in *single* quote marks. Figure 2.3 shows how these data values are represented in binary using an 8-bit code known as ASCII (American Standard Code for Information Interchange). When it became necessary to accommodate other alphabets, a 16-bit code, known as Unicode was introduced. ASCII is a sub-code of Unicode.

Note in Figure 2.3 that:

- The codes for numbers are smaller than the codes for upper-case letters, which in turn are smaller than the codes for lower-case letters.
- The codes for upper-case and for lower-case letters are contiguous, i.e. 'a' is 'b'-1, and 'b' is 'c'-1, etc.
- If you subtract the code for a '0' from the code of any numeric digit, you get that digit's int value: '7' - '0' = 55 - 48 = 7

2.1.5 Strings of characters: String

Normally when working with character data, we wish to group several characters into a single entity, known as a *String*. Examples of String data are "joe", "dataSet314", and "3a09d&#9aw". Note that the characters of a String are enclosed in *double* quote marks. Each String may contain any number of characters from the keyboard.

```
String          length
"joe j"         5
"jim"           3
"jo"            2
"j"             1
""              0

'j'             This is a char,  not a String
'jo'            This is a mistake, a char must consist of
                exactly one character
```

Figure 2.4: Examples of Strings and chars, showing the length of each String

Character to be included	Escape sequence	Example
double-quote	\"	"Title: \"Pygmalion\", Shaw"
newline	\n	"line break \n here"
tab	\t	"\tcol1 \tcol2 \tcol3"
backslash	\\	"\\root\\directory\\filename"

Figure 2.5: Escape sequences are used to include special characters in a String

The `String` data type is a reference type, not a primitive type (it begins with an upper-case letter). As shown in Figure 2.4:

- each String has a length
- a space character counts as one of the characters in a String
- the length of a String can be 0

How can we include the double-quote character in a String constant? We use what is called an *escape* sequence, using the backslash character:
`"She said \"hi\" to me"`

The escape chracter can also be used to quote a newline character (`'\n'`) or a tab character (`'\t'`) as shown in Fig 2.5

Finally, we should be able to include the backslash in a String.

2.1.6 Other reference types

Because we have defined a class named `Student`, this may also be used as a data type (it is a reference type). Thus there can be any number of data types, corresponding to classes we have defined. Moreover, we have access to thousands of classes that others have defined and can use them as data types. These classes are available in what is known as the *Java class library*, and `String` is just one example of the classes available there.

2.1.7 Exercises

1. Having defined `Student` as a class, can `Student` be used as a type? If so, would it be a primitive type or a reference type?

2. Refer to Figure 2.2.

   (a) Show a similar table for 5-bit words. (Hint: the value of zero is 00000).

   (b) How many different numbers can be represented with 5 bits?

   (c) What is the largest positive number that can be represented with 5 bits?

   (d) What is the smallest negative number that can be represented with 5 bits?

3. Complete the following table:

word size (bits)	number of different values	largest positive value	smallest negative value
4	16	7	-8
5	?	?	?
8	?	?	?
16	?	?	?
32	?	?	?
64	?	?	?
n	?	?	?

   Hint: Sometimes it is easier to show a number as a power of 2.

4. What are the *primitive types* representing whole numbers which correspond to the word sizes 8, 16, 32, 64, respectively? Hint: There are 8 bits in a byte.

5. Write Avogadro's number as a Java constant.

6. Arrange the following constants in order from smallest to largest: 99.0, 1405.3e-12, 1e2, -999.3e45

7. What is the length of each of the following Strings?

   (a) `"elephant"`

   (b) `"my small cat"`

   (c) `"$*&#@!("`

   (d) `"x"`

   (e) `""`

8. What is the value of each of the following?

Operator	Meaning	Example	Result
+	Addition	$7 + 4$	11
−	Subtraction	$7 - 4$	3
∗	Multiplication	$7 * 4$	28
/	Division (quotient)	$7/4$	1
%	Mod (remainder)	7%4	3

Figure 2.6: Common arithmetic operations

(a) '8' - '3'

(b) '8' - '0'

(c) '7' - '0'

(d) '6' - '0'

(e) 'f' - 'b'

(f) 'a' - 'z'

2.2 Operations and expressions

Java statements make extensive use of operations (such as add and subtract) and expressions (such as $(a + b) - 3$). But there are many more operations available. This section will introduce some of the more common operations.

2.2.1 Arithmetic operations

The most common arithmetic operations are listed in Figure 2.6.

As an example of an operation, $3 + 4$ would compute the value 7. If either, or both, operands of an operation are floating point (i.e. `float` or `double`), then a floating point operation is performed, producing a floating point result, not an int result. In many cases this will appear to have no effect. For example, $3 + 4$ produces the `int` result 7, but $3.0 + 4$ produces the `double` result 7.0.

This behavior is most evident with division. Division of ints must produce an int result, with no decimal places. Therefore, $7/4$ produces the `int` result 1, not 2, nor 1.75. Decimal places are truncated.

The *modulus* (or *mod*) operation, designated, perhaps surprisingly, by the % operator, has nothing to do with percentages. It should be applied to ints only, and the operation $a\%b$ produces as a result the `int` remainder when a is divided by b. Mathematically, integer division has two results: a quotient (obtained by the / operator, and a remainder (obtained by the % operator).

Examples of division and mod operations are shown in Figure 2.7.

2.2.2 String operations

Strings of characters can also be manipulated with operations. The simplest String operation is called *concatenation* and is designated with a + operator.

Example	Result	type of result	remarks
15/8	1	int	decimal places are truncated
15/8.0	1.875	double	at least one operand is double
15.0/8	1.875	double	at least one operand is double
15%8	7	int	remainder after division
0%5	0	int	
1%5	1	int	
2%5	2	int	
3%5	3	int	
4%5	4	int	
5%5	0	int	
6%5	1	int	
7%5	2	int	
10%5	0	int	mod serves as a circular counter

Figure 2.7: Some examples of division and mod operations

The + operator in this case does not mean addition; it means concatenation. When an operator can take on different meanings, depending on its operands, we say that the operator is *overloaded*. The arithmetic operators were already overloaded because the operation could be either an int operation or a floating point operation, depending on the types of the operands.

In the case of strings, concatenation allows us to form a new String from two smaller strings:

- "john" + "son" produces the result "johnson"
- "no" + "thing" produces the result "nothing"
- "no" + " thing" produces the result "no thing"
- "john" + " " produces the result "john "
- "john" + "" produces the result "john"
- "%f*!3" + "321" produces the result "%f*!3321"

Recall that a space character counts as a character in a String, just like any other character that you find on the keyboard. Also the String "" represents a String whose length is 0 (sometimes called a *null* String, though we will not do so to avoid confusion with the null reference)

When one of the operands of the + operator is a String, but the other operand is not a String, the non-String operand is automatically converted to a new String, which is then used as the operand for the concatenation:

- 23 + "skiddoo" produces the result "23skiddoo"
- "skiddoo" + 23 produces the result "skiddoo23"

- "17" + 23 produces the result "1723"
- 17 + "23" produces the result "1723"
- 17 + 23 produces the result 40

The String class is part of the Java Class Library, and it provides us with a multitude of features with which we can manipulate strings. To see the complete documentation for the String class, point your web browser to a current version of the API for the Java class library (`docs.oracle.com/javase/7/docs/api`). We now give a few of the more useful operations on Strings. These are all methods in the String class, and can be called by attaching them to any String (which we call *this* String).

- `int length()` - returns the length of this String as an int.
- `char charAt(int index)` - returns the char at the given position of this String. The first character is at position 0.
- `int indexOf(String str)` - returns the position of the first occurrence of `str` in this String, or -1 if not found.
- `String substring(int begin, int end)` - returns a part of this String, beginning with the character at position `begin`, and ending with the character at position `end` - 1.
- `String substring(int begin)` - returns a part of this String, beginning with the character at position `begin`, and and continuing to the end.
- `String toUpperCase()` - returns a new String in which all lower-case letters of this String have been converted to upper-case.
- `int compareTo(String otherString)` - returns a positive int if this String follows `otherString` alphabetically, returns a negative int if this String precedes `otherString` alphabetically, and returns 0 if the this String is equal to `otherString`.

Examples of these methods are shown in Figure 2.8.

2.2.3 Arithmetic Expressions

Several operations may be combined into a single *expression* using parentheses as needed to indicate the order of operations. Some examples of expressions are shown in Figure 2.9.

The concept of *expression* is so fundamental to java programming, that we give a more formal and precise definition for expressions involving ints here, in which *expression* is abbreviated as *expr*:

An expr may be a:

1. number

Method signature	Example	Result
`int length()`	`"and how".length()`	`7`
`char charAt(int index)`	`" and how".charAt(5)`	`'h'`
`int indexOf(String str)`	`"jonson".indexOf("on")`	`1`
	`"jonson".indexOf("on.")`	`-1`
`String substring(int begin, int end)`	`"and how".substring(4,7)`	`"how"`
`String substring(int begin)`	`"and how".substring(2)`	`"d how"`
`String toUpperCase()`	`"AnD HoW23!!".toUpperCase()`	`"AND HOW23!!"`
`int compareTo`		
`(String otherString)`	`"john".compareTo("sam")`	a positive int
	`"john".compareTo("johnson")`	a negative int
	`"john".compareTo("john")`	`0`
	`"John".compareTo("john")`	a negative int
	`"99".compareTo("100")`	a positive int

Figure 2.8: Examples of some String operations

Expresssion	Value
$4 * 7 + 3$	31
$(4 * 7) + 3$	31
$4 * (7 + 3)$	40
$((7 - 2) * (7 + 3))/4$	12
$((7 - 2.0) * (7 + 3))/4$	12.5

Figure 2.9: Examples of arithmetic expressions

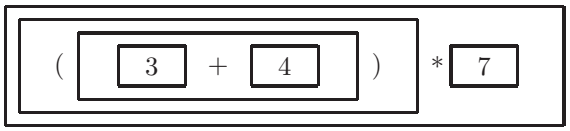

Figure 2.10: Applying the definition of expression to $(3 + 4) * 7$

2. expr + expr
3. expr - expr
4. expr * expr
5. expr / expr
6. expr % expr
7. (expr)

We now suggest an exercise in which we apply the definition given above, to determine whether a piece of code constitues a valid expression. We apply the rules of the definition, one at a time, by putting a box around each expression (or subexpression) noting that nothing is an expression until there is a box around it. Figure 2.10 shows how the definition can be applied to the expression $(3 + 4) * 7$. Note that each box represents a subexpression, and shows which rule of the definition is applied.

Note that our definition of *expr* involves the word *expr*. We call this a *recursive* definition, and this is valid and legitimate as long as:

- At least one rule does not involve the usage of *expr*. This is called the *base case*.
- Rules that use the word *expr* contain other text as well. I.e. a rule should not define an *expr* to be merely an *expr*.

It is interesting to note that any precise definition of expression *must be recursive*. This results from the fact that an expression is inherently recursive. If you continue to study Computer Science, you will learn the importance of recursion.

A better version of Figure 2.10 is shown in Figure 2.11. Here we show the rule number from the definition of expression each time a rule is applied with a box. For example, rule 2 of our definition is:

2. expr + expr

Thus, in Figure 2.11 there is a small number 2 in the upper right corner of the subexpression, $3 + 4$.

Another example is shown in Figure 2.12, in this case for the expression $3+4*7$. Here we note that there is an alternate solution, shown in Figure 2.13.

This is a serious problem. There appear to be two different ways of applying the rules of our definition. The solution shown in Figure 2.12 suggests that $3+4$

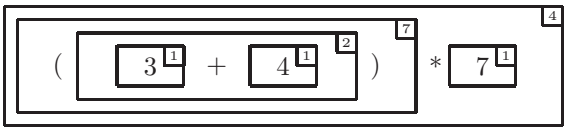

Figure 2.11: Include rule numbers from the definition of expression when applying the definition to $(3 + 4) * 7$

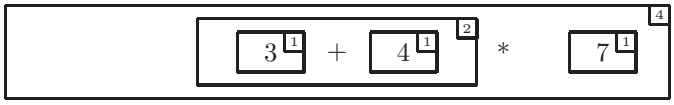

Figure 2.12: Applying the definition of expression to $3 + 4 * 7$

is a subexpression, and that therefore the addition is done before the multiplication. On the other hand, Figure 2.13 suggests that $4 * 7$ is a subexpression, and that therefore the multiplication is done before the addition.

These two solutions produce different results for the same expression, 49 in the first case and 31 in the second case. This is not good; an expression must have a single consistent value. The reason for this problem is that our original definition is *ambiguous*. This means that it allows for different interpretations of the same expression.

One way to resolve this ambiguity is with *precedence rules*:

- Multiplication, division, and mod always take precedence (are performed first) over addition and subtraction. For example $3 + 4 * 7 = 3 + (4 * 7)$, as shown in Figure 2.12. If one wishes to do the addition first, one must use parentheses: $(3 + 4) * 7$. We now see that Figure 2.12 does not suggest a true interpretation of the given expression, but that Figure 2.13 does suggest a true interpretation.

- If there is more than one operation at the same level of precedence, they are executed left-most first. For example, $9 - 4 - 2 = (9 - 4) - 2$ and $12/2/3/2 = ((12/2)/3)/2$

2.2.4 Exercises

1. Show the *value* and *type* of each of the following expressions.

   (a) `2 * 8.0`

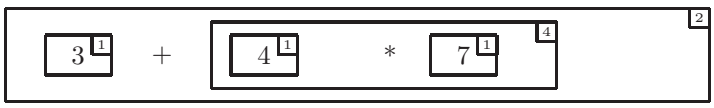

Figure 2.13: An alternate solution when applying the definition of expression to $3 + 4 * 7$

(b) `17 / 5`

(c) `17 % 5`

(d) `17 / 5.0`

(e) `234884173 % 10`

(f) `"some" + "body"`

(g) `("some" + "body").length()`

(h) `"somebody".charAt(2)`

(i) `"SomeBody".toUpperCase()`

(j) `"somebody".indexOf ("me")`

(k) `"somebody".indexOf ("Me")`

2. Find the *value* of each arithmetic expression shown below. They should all be ints.

(a) `4 + 2 * 3`

(b) `(4 + 2) * 3`

(c) `8 / 2 / 3`

(d) `(9999 / 10000) * (192 - 383)`

(e) `94528 % 5 + 9 / 10`

3. Draw boxes around each of the *expressions* in the previous problem, as shown in Figure 2.11. Show the rule number applied in each box, and be sure to obey the Java precedence rules.

2.3 Declaration and initialization of variables

In Java a *variable* may be used to store, or remember, a value. A variable may be a single letter or it may consist of many letters, underscore characters, and/or numeric digits, but it must begin with a letter. Some examples of variables are x, total, birthDay2014, sum_total. Note that:

- These are NOT strings (they are not in double quote marks).
- As usual, these are case sensitive, so the variable $sumOfProducts$ is different from the variable $sumofproducts$.
- The compiler will permit a variable to begin with an upper-case letter, but we will not do this (only class names should begin with an upper-case letter).

2.3.1 Declaration of variables

Before a variable can be used, its type must be declared:
type variable − list;
For example, the declarations:

```
int x, sum, result;
String name;
```

mean that the three variables `x`, `sum`, and `result` all will be used to store int values, and the variable `name` will be used to store a (reference to a) String.

2.3.2 Iniitialization of variables

Before a variable can be used in an expression it must have a value. We can give a value to a variable in the same statement which declares the variable. This is known as *initialization*, and the format is:
type variable = expression;
Examples of variable initializations are shown below:

```
int x=0;
double sum=0.0, tolerance=0.0001;
boolean done=false, ok;
String name = "joe";
```

The above gives initial values to the variables `x`, `sum`, `tolerance`, `done`, and `name` but it leaves the variable `ok` uninitialized.

2.3.3 Exercises

1. Which of the following are *not valid names* for variables?
   (a) flummox33foo22
   (b) 3x
   (c) var(ok)
2. Show how the variables `salary`, `tax`, and `fica` can all be declared to be of type `double` and initialized to the values 99,000, 345.53, and 150, respectively, all in one statement.

2.4 Assignment of values to variables

Variables are used to store, or remember, values. A value can be assigned to a variable using the *assignment* operator, =, as shown below:
variable = expression
For example, $result = 3 + 4 * 7$ means that the value 31 will be stored in the variable *result*. The assignment operator can be summarized:

1. The expression on the right side of the operator is evaluated.

2. The value of the expression is stored in the variable.

3. The type of the expression should match the type of the variable. If they are of different types, in some cases the expression can be converted to the appropriate type (as described in the next section).

The = operator does *NOT* mean *equals* and should not be read that way. We suggest reading it as 'is assigned the value of' or 'gets the value of'. Here are a few examples to clarify the requirement that types match:

```
int x = 8;
String city = "Boise";
x = x + city.length();     // x is now 13
city = "New York";
x = city;                  // ERROR - types do not agree
city = 18;                 // ERROR - types do not agree
18 = x;                    // ERROR - left operand must
                           //         be a variable
```

We can now add another rule to our definition of *expression*:

An *expr* may be: 8. *variable*.

This means that we can include $a + b * 28$ as a valid expression. When it is evaluated, the *current* values of a and b are used. (Values should have been assigned to these variables before attempting to use this expression)

We are now able to form an executable Java statement. A Java statement may be:

variable = *expression*;

Java statements are written sequentially, and are executed in the order in which they are written. We will have a more extensive definition of *statement* in chapter 3. Figure 2.14 shows an example of a series of statements, along with the values stored in variables as the statements are executed. Note that:

- The value of a variable can change as the statements are executed.

- When an expression involving variables is evaluated, the *current* value of each variable is used.

- The statement $x = y$; does **NOT** imply a comparison of the value of x with the value of y. It means find the value of y and store that value into the variable x.

Students are often confused by the sequential nature of expressions and statements, but careful attention to these examples should help alleviate the confusion.

Program code	Value of a	Value of b	Value of c
int a,b,c;			
a = 3;	3		
b = 2;	3	2	
c = a+b*2	3	2	7
a = b;	2	2	7
c = (a+b)*5	2	2	20
b = 8;	2	8	20
b = a;	2	2	20
b = b+1;	2	3	20

Figure 2.14: Examples of assignment statements, and their effect on variables

2.4.1 Type conversion in assignments

An int may be assigned to a float or double variable:

```
int total, count;
double average;

total = 10;
count = 3;
average = total / count;
```

In this example, the division produces an `int` result, 3, which is then converted to `double` 3.0 when assigned to the variable average.

The compiler will not allow a float or double to be assigned to an int:

total = 3.0; The compiler will produce an error message: "Possible loss of precision" even though the value being assigned is a whole number.

The programmer can assure the compiler that this possible loss of precision is acceptable by using a *cast*, in which case the decimal places are truncated:
total = (*int*)3.9;
The variable *total* is assigned the `int` value 3

A `char` may also be casted to yield its numeric ASCII code:

```
char ch = 'x';
int code = (char) ch;      // code is 120
```

2.4.2 Type conversions and initializations

At this point we caution the student to be careful when initializing variables; consider the following example:

```
double x = 5 / 3;
```

The intent here is to initialize the variable `x` with the value 1.6666666667, the result of the division. However this is not what happens. The division will be an `int` division because both of its operands are ints, producing an `int`

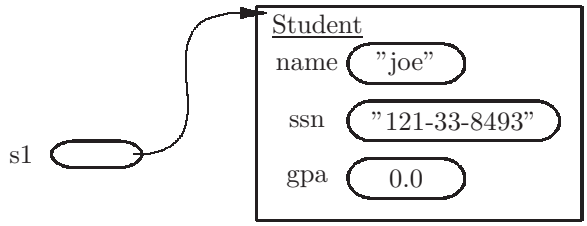

Figure 2.15: An object diagram showing a variable storing a reference to a Student object

result: 1 which is then converted to 1.0 when assigned to the variable `x`. There is an important lesson to be learned here: the components of a statement are executed separately and sequentially; they are not executed 'all at once'.

2.4.3 Assignment of references

The assignment operator will always evaluate the expression on the right side, and assign that value to the variable on the left side. This is straightforward for variables of primitive type. However, for variables of reference type we need to take a careful look at the behavior of the assignment operator.

Recall that a variable of reference type does not store the actual data; rather it stores a *reference* to the data. For example, the declaration

```
int gpa;
```

means that `gpa` stores an int; however the declaration

```
Student s1;
```

does *not* mean that `s1` stores a Student. Rather, it means that `s1` stores a *reference* to a Student (initially a null reference). We can change that reference as shown below: `s1 = new Student ("joe", "121-33-8493");`
Figure 2.15 shows the object diagram which would result. The reference stored in the variable `s1` is actually a memory location for the Student object to which it refers, and this is depicted in the diagram with an arrow.

Suppose we declare another Student variable:

```
Student s2;
```

Now, however, instead of creating a new Student object, we use the assignment operator thus:

```
s2 = s1;
```

This simply means to copy the reference which is in `s1` into `s2`. Consequently the variables `s1` and `s2` will be storing the same reference, and will refer to the same object, as shown in Figure 2.16 .

This becomes interesting if we were to make a change to the data:

```
s1.setName("jim");
```

changing the name of the Student. The result is shown in Figure 2.17 . We then access the name of Student s2:

```
System.out.println ("The name of s2 is " + s2.getName());
```

which will print "jim". Since `s1` and `s2` refer to the same object, a change to

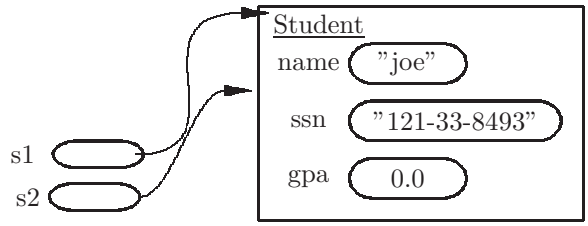

Figure 2.16: An object diagram showing the effect of assignment of a reference to a variable: `s1 = s2;`

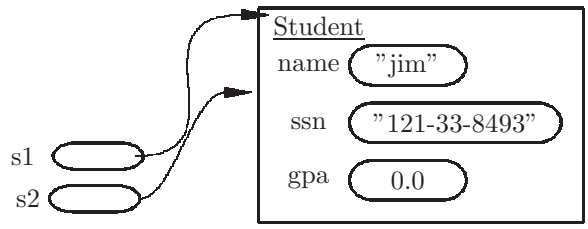

Figure 2.17: An object diagram showing the effect of a change to the variable `s1`. The name of `s2` is now `"jim"`.

the object referred to by `s1` will also change the object referred to by `s2` even though we never made an explicit change to the variable `s2`.

2.4.4 Exercises

1. Show the final *values* of the variables `x,y,z` after the code shown below has executed.

```
int x = 3,y,z;
y = x + 2;
z = y;
x = 14;
y = y + 1;
```

2. Show the final *values* of the variables `x` and `i` after the code shown below has executed.

```
int i;
double x;
i = 4;
x = i;
i = i + 1;
```

3. Which of the statements shown below will cause a *syntax error* from the compiler?

```
int i = 7;
double x = 2.5;
double y = 2.0;
i = x;
i = y;
x = i;
```

4. Show the final *values* of the variables `i,j` and `y` after the code shown below has executed.

```
int i = 7;
double x = 2.99, y = 11 / 12 * 2;
i = (int) x;
j = ((int) ((3.5 / 4) * 4)) % 7
```

5. Refer to the University class introduced in the exercises from chapter 1.

```
University u1, u2, u3;
u1 = new University ("Slippery Rock",1000);
u2 = u1;
u2.setSize(900)
System.out.println ("Size of u1 is " + u1.getSize());
```

   (a) What will be *printed* by the code shown here?

   (b) Draw an *object diagram* showing the values of u1, u2, and u3 after the code shown here is executed.

2.5 Method definitions, signatures, and invocation

We introduced the notion of a Java *method* in chapter 1. Methods, more properly known as *instance methods*, are used to define the behavior of objects in a class.

2.5.1 Method definition

A method definition consists of an *API*, a *signature* and a *body*.

The API (Application Program Interface) is ignored by the compiler, but is very useful to the programmer. It defines the purpose of the method, what it expects from the calling method (preconditions), and what it produces as a result, or what changes it makes to objects (postconditions). Hence the API provides all information needed for someone to write a call to the method. The API begins with $/**$ and ends with $*/$. This API is processed by a utility program known as *javadoc* which produces html for a nice looking description of the method's purpose, preconditions, and postconditions in the form of a web page.

```
/** Calculate this student's gpa.
 *  Precondition: Both parameters are positive.
 *  Postcondition: This student's gpa is set
 *                 as the result of dividing
 *                 the grade points by the credits
 */
public void calculateGPA(int gradePoints, int credits)
{  double creditsAsDouble;
   creditsAsDouble = credits;
   gpa = gradePoints / creditsAsDouble;
}
```

Figure 2.18: Method which calculates, and sets, this Student's gpa, given total number of grade points and total number of credits

2.5.2 Method signature and body

The method signature consists of:

1. Visibility, i.e. `public` or `private`: Private methods can be invoked only from another method in the same class, whereas public methods may be invoked from a method in any class.

2. Return type: This is the type of the explicit result of the method. In this case we think of the method as being similar to a mathematical function which produces a single result. If the method produces no result, the return type is `void`.

3. Method name: The name of the method, as with variables, may consist of 1 or more alphabetic and/or numeric characters, but it must begin with an alphabetic character. Note that the method name is case sensitive.

4. Parameter list: 0 or more variables, with types specified, separated by commas. The list must be enclosed in parentheses, even if there are no parameters. We call these parameters *formal parameters* to distinguish from *actual* parameters, described below. Parameters are used to pass data into a method.

The method body consists of Java statements, such as assignment statements, all enclosed in one set of curly braces.

Figure 2.18 shows an example of a method which could be included in the `Student` class. The purpose of this method is to calculate, and change, the student's gpa, given the total number of grade points, and the total number of credits.

Note that we do not wish to do a floating point division in this method, so we assign the `int` value of `credits` to a `double` variable, `creditsAsDouble`. Then the division is a floating point division, producing a floating point result.

2.5.3 Method invocation

How and when are methods executed? Generally, they are called, or invoked, from another method. In the method call, the name of the method being called is attached to an object on which the method is being invoked. Also, actual values of parameters are included in the parameter list. These *actual parameters* may be constants, variables, or more complex expressions, but their types *must* correspond to the types of the corresponding formal parameters in the method signature.

As an example, we could invoke the calculateGPA method in the Student class from a method in some other class as shown below:

```
Student s1;
int calculus, comp, stats;
calculus = 3;
comp = 4;
stats = 2;

s1 = new Student ("jim", "240-33-4321");

s1.calculateGPA (calculus*4 + comp*3 + stats*2,
                 4 + 3 + 3);
```

When a method is invoked:

1. The actual parameters, which are expressions, are evaluated, and these *values* are copied to the corresponding formal parameters in the method definition.
2. The statements in the method being invoked are executed in sequential order.
3. When the final statement has been executed (or a `return` statement is executed), control returns to the calling method.

Be sure that actual parameters in a method call correspond in number and type with formal parameters in the definition of the method being called. Figure 2.19 illustrates some valid and non-valid method calls.

Also note that the following code is not correct:

```
Student s2;
s2.calculateGPA(20,6);
```

When the variable `s2` is declared, its value is `null`, a reference which refers to nothing. A Student has not been instantiated, and no valid value has been assigned to the variable `s2`. This will produce a run-time error called a `nullPointerException`, [1] meaning that your program will come to a crashing

[1] It is unfortunate that this exception is named *nullPointerException* in the java class library, rather than *nullReferenceException* Other programming languages use pointers which are similar to references, but it is possible to do arithmetic with pointers.

Method signature	Method invocation	Remarks
public void		
meth(int a, char b)	s1.meth(3);	Incorrect: there are two formal parameters and only two actual parameters
	s1.meth(3, 4);	Incorrect: The type of the second actual parameter must be char, to agree with the second formal parameter
	s1.meth(3,'b');	ok
	x = s1.meth(3,'b');	Incorrect: The method is a void method and has no explicit result for assignment to x.
public int		
evenOdd (int a)	int result;	
	result = evenOdd(17);	ok
	evenOdd(17);	The compiler will accept this, but it is probably not what is desired; the explicit result is discarded.

Figure 2.19: Examples of correct and incorrect method calls. Actual and formal parameters must have a one-to-one correspondence. Void methods have no explicit result.

halt. This is to be avoided. In general, when your program halts unexpectedly with a `nullPointerException`, check the variable *to the left of the dot*. It should not be null; make sure you have assigned a value to it.

Figure 2.20 shows a method named `getRoundedGPA` which will return the student's gpa, rounded to two decimal places. It may be invoked from another class:

```
double roundedGPA;
Student s1;
s1 = new Student ("jim", "123-33-3222");
...   Statements establishing a GPA for s1, not shown here
roundedGPA = s1.getRoundedGPA();
```

Note that:

- The method `getRoundedGPA` returns an explicit result, which is then stored in the variable `roundedGPA`.
- The method `getRoundedGPA` has no parameters but the parentheses are still needed, both in the method definition and in the method call.
- When the method's `return` statement is executed, the method terminates execution (even if there are more statements after the `return` statement), and the explicit result of the method is available to the calling method.

```
/**  Postcondition: return this Student's gpa,
  *                 rounded to the nearest one
  *                 hundredth.
  */
public double getRoundedGPA()
{  int gpaAsInt;
   gpaAsInt = (int) (gpa * 100 + 0.5);
   return gpaAsInt / 100.0;
}
```

Figure 2.20: Method which returns a Student's gpa, rounded to the nearest hundredth

2.5.4 Methods From the Java Class Library

[2]

There are many methods which are predefined and ready for use, in the Java class library. Here we discuss a few methods in the `Math` class. These methods all happen to be static methods, otherwise known as class methods. To invoke a class method it must be preceded by the name of its class.

To find the absolute value of a number, use the `abs` method. Its parameter may be an `int` or a `double`, in which case it returns an `int` or a `double`, respectively.[3] For example, Math.abs(-3) would return 3, and `Math.abs(12.05)` would return `12.05`.

There is a method in the Math class which raise a number to a given power (i.e. exponent). It's name is `pow` and it is defined for double precision floating point only.[4] For example, `Math.pow(4.0,3)` would return $4.0^3 = 64.0$.

Another method is used to find the (positive) square root of a number. It is abbreviated `sqrt`. For example, `Math.sqrt(16)` would return `4.0` and `Math.sqrt(0.01)` would return `0.1`.

The Math class also has a method which will return a random floating point value. It has no parameters and always returns a random `double` in the range `0.0 .. 1.0`. For example, `Math.random()` might return `0.32803492`. [5]

These methods are summarized in Fig 2.21.

2.5.5 Exercises

1. Describe the *error* in each of the following method signatures:

[2]This section may be omitted without loss of continuity, but note that the methods described here are in the AP subset of the Java language.

[3]Also available are float and long.

[4] If provided with an int or float parameter it will automatically convert the parameter to type `double` and will always return a result of type `double`.

[5]There are other useful random number methods in the `java.util` package (see chapter 5).

Method signature	Example	Result
`int abs(int x)`	`Math.abs(99)`	`99`
`double abs(double x)`	`Math.abs(-9.09)`	`9.09`
`double pow(double base, double exponent)`	`Math.pow(-2.0, 3.0)`	`-8.0`
`double sqrt(double x)`	`Math.sqrt(100.0)`	`10.0`
`double random()`	`Math.random()`	`0.771107`

Figure 2.21: Examples of some class methods from the Math class

(a) `public myMethod ()`

(b) `public int void myMethod()`

(c) `public void myMethod (x, y, z)`

(d) `int myMethod(double x)`

2. Define a *method* for the Student class which will return the student's gpa as a percentage of 100. For example if the student's gpa is 2.5, the result should be `"62.5 %"`. The name of the method should be gpaAsPct, and it should have no parameters (don't forget to include the API).

3. Define a method which could be included in any class to return the *average gpa* of three students. The name of the method should be average3, and there should be three parameters (all students).

4. Consider the following method definitions, which could be included in any class:

```
/** Change the name of the given Student to "jim"
 */
public void meth1(int x, Student s)
{   x = x + 1;
    s.setName ("jim");
}

/** This method is used to expose passing of
    parameters.
 */
public void meth2()
{  int x = 7;
   Student s1 = new Student ("joe", "999-99-9999");
   meth1 (x,s1);
   System.out.println ("x is " + x);
   System.out.println ("name of s1 is " + s1.getName());
}
```

Show the what would be *printed* by a call to `meth2()`

Hints:

- The value of an actual parameter is copied into the corresponding formal parameter. They are separate and distinct variables, even if they have the same name.
- When a parameter is a reference type, the reference, not the object to which it refers, is copied to the actual parameter.

2.6 Recursive methods

Methods can call themselves. These are called *recursive* methods, and the concept is similar to our recursive definition of *expression*. We will take a more careful look at recursive methods in chapter 3.

2.6.1 Exercises

1. In the definition of *expr* given earlier in this chapter, which of the rules do not make use of the word being defined?
2. In the definition of *expr*, which of the rules do make use of the word being defined?

2.7 Printing the output

Up until now we have considered computations that occur when a program executes. At some point, however, we may wish our program to display information for the user. The way this is done can vary considerably. In most modern applications a *graphical user interface* or *GUI* is used to display results in a multitude of different forms. We will introduce GUIs in chapter 10, but for now we will simply display plain text strings for the user to view. Depending on the development environment you are using this could appear in a few different ways:

- If using an IDE such as BlueJ, NetBeans, or Eclipse, the IDE will open a window or pane in which the text is displayed. BlueJ calls this the *terminal window*.
- If running the program from a unix or Windows command line, the text will be displayed in that terminal window.

In any case, the output is produced by a `print` or `println` method in the `System` class. Each of these methods has one parameter - a String. When calling these methods, we must provide a String value. This can be a String constant, a String variable, or any more complex expression which evaluates to a String. Figure 2.22 shows some examples of calls to the print methods.

```
String name = "joe";
int total = 99;
```

Method call	output
`System.out.println ("hello");`	`hello`
`System.out.println (name);`	`joe`
`System.out.println ("hello " + name);`	`hello joe`
`System.out.print ("hello ");`	
`System.out.println (joe);`	`hello joe`
`System.out.println ("The total is " + total);`	`The total is 99`

Figure 2.22: Examples of calls to print methods

Note that the *println* method prints its output on a separate line, whereas the *print* method does not. Students who have programmed in other languages are cautioned that the print methods have just one parameter. Other languages may utilize a sequence of expressions separated by commas, but not Java.

2.7.1 Exercises

1. Show what would be *printed* by the following code segment. Be careful in regard to newlines.

```
int x = 3;
String str = "foo";
System.out.print (str + "bar");
System.out.println ("foobar");
System.out.println (x + 2 + " is the result.");
System.out.println ("The result is " + x + 2);
```

2.8 Constants and class variables

2.8.1 Constants

There are many cases where we need to use a value in a program repeatedly. For example, in a mathematical application we may wish to use the value of Pi, 3.14159... in many parts of a class or method. It is best to use a named constant in cases like this, as opposed to the actual value. This can be done with the keyword `final`. A variable which is declared to be *final* can be initialized, but can never be changed after its initialization. An example would be:

```
final int PI = 3.14159;
```

whereupon we would then use the variable PI instead of the number 3.14159 throught the scope of that variable. This has a few advantages:

- The program is easier to read and understand; presumably the name of the variable provides a clue as to the meaning or intent of the value being used.

- If the value is incorrect, it needs to be corrected in one place only, the initialization of the constant.

For historic reasons constants are usually written in all upper-case letters, and we will conform to this practice despite the fact that it contradicts our convention that only class names begin with upper-case letters.

2.8.2 Class variables

We've seen that classes can have fields (also known as instance variables). Each object of the class has its own copy of the values for those fields; these make up the *state* of the object. In situations where all objects are to share the same value for a field, we can make use of *class variables*. A class variable can be declared with the `static` keyword. This means that there is only one copy of the variable, shared by all objects of that class. Our Student class could have a class variable to designate the maximum number of credits which can be taken by any Student:

```
public static int maxCredits = 18;
```

This declaration would generally be placed with the other fields. A static field is a class variable, and a non-static field is an instance variable. Since the field maxCredits is public and not final, some other class (such as University) could change its value, in which case all student objects would see the new value of maxCredits.

A public class variable can be accessed from any other class, but the name of its class needs to be specified:

```
System.out.println ("Max credits is " + Student.maxCredits);
```

2.8.3 Class constants

Constants and class variables are most often used together, as shown below:

```
public static final int MAX_CREDITS = 18;
```

The variable `MAX_CREDITS` is both a class variable (static) and a constant (final). This is often referred to as a *class constant*. In this case all Student objects would share it value, which can never be changed during execution of the program.

Many students confuse `static` with `final`, and understandably so, because of the normal usage of 'static' in the English language – "Having no motion; at rest". The Java keyword `static` does NOT mean that the variable cannot be assigned a new value.

There are many class constants in the Java class library, most notably the constant PI in the class Math. We can use it as shown below:

```
double area = Math.PI * radius * radius;
```

2.8.4 Exercises

1. You are given the following class definition of `MyClass` (this class has no constuctors, so the compiler supplies a default constructor which does not initialize any of the fields):

```
public class MyClass
{  private int x = 3;
   public static int var = 7;
   public final  int VAR = 8;
   public static final int MAX = 99;
}
```

   Which of the following statements in a method of some other class would cause *syntax errors*?

```
MyClass mc = new MyClass();
mc.x = 4;
mc.var = 18;
MyClass.var = 9;
mc.VAR = 0;
MyClass.VAR = 9;
System.out.println (mc.MAX);
System.out.println (MyClass.MAX);
```

2.9 Comments and readability

It will soon become evident that programs can be arbitrarily complex, difficult to read, and difficult to understand completely. Even if a program produces correct output, its value is limited if people, including the original programmer, find it difficult to read, understand, and make modifications. For this reason it is extremely important that the programmer make efforts to explain and clarify all aspects of the program.

2.9.1 Formatting a program

One way of clarifying a program is by formatting it in a readable way. Notice in Figure 2.20 that the statements of the method are placed on separate lines. As far as the compiler is concerned, these statements could be written on 1 line, or any number of lines; Java is free format. However, it is in our best interest to format the program in what we consider to be a readable style. We will discuss this further in chapter 3.

Also notice that the statements in a method are indented. Again, this is not required by the compiler, but we do it anyway to make the program easier to read. In the next few chapters indentation becomes increasingly important.

2.9.2 Comments

Another way of improving the clarity of a program is to provide *comments*. Comments are ignored by the compiler, but are helpful to programmers (including the original programmer) trying to understand the program. Java allows for two kinds of comments: *single-line* comments and *mult-line* comments.

Single-line comments begin with // and end at the end of the line. Some examples of single-line comments are shown below:

```
// Calculate the gpa
gpa = gradePoints / credits;  // This should be a floating point division
```

Note that a single-line comment can stand alone by itself on a line. It can also be tacked on to a statement (or even part of a statement). It ends at the end of the line.

Multi-line comments allow for a single comment to span across several lines. A multi-line comment begins with /* and ends with */ as shown below:

```
/* Calculate the gpa by dividing
   grade points by the total number
   of credits.  Be sure that this
   is a floating point division.
*/
gpa = gradePoints / credits;  /* Do NOT divide by zero! */
```

Be sure to include the */ which terminates the multi-line comment. If this is omitted, the remainder of the program will be viewed as one long comment! It should now be clear that the specifications of pre-conditions and post-conditions (the API) in Figure 2.18 is really a special kind of multi-line comment; it begins with /** instead of just /*.

A good programmer will include a liberal dose of comments throughout a program. Rarely is a software project declared to be finished, complete, and fixed. Rather, there will always be corrections, enhancements, extensions, etc. Whether these modifications are to be made by the original programmer or by a maintenance programmer, the comments will guide the way through the existing code.

2.9.3 Exercises

1. Rewrite the following code segment to be more readable without altering the meaning:

```
int sum = 0; double average; sum
=

first +
```

```
second + third; average =
((
double) sum
)/ 3; System
.   out.println (
average)
;
```

2. Show what would be printed by the following code segment:

```
// int x = 3;
double x = 1, y = 2;
/*
y = 4;   //  x = y+3;
System.out.println (x + y);
*/
y = x / y;      // x = 17;
System.out.println (x + y);
```

2.10 Program elements and methods in the GridWorld case study

Chapter 3

Selection Structures

We have seen that the statements in a method are executed sequentially, in the order in which they appear in the method definition. However, it is often that we might wish to alter this *flow of control*. For example, we may wish to execute a statement only if certain conditions are satisfied; or we may wish to skip over a statement if certain other conditions are satisfied. For this purpose, Java provides *selection structures*. We have *one-way* selection structures and *two-way* selection structures, which permit us to execute statement(s) *conditionally*. But first we need to discuss comparison operators and boolean operations.

3.1 Comparison operators

Java numbers may be compared with operators similar to those you've seen in math courses. These operators operate on two numbers, and always produce a boolean result: `true` or `false`. The six comparison operators are described in Figure 3.1.

It is important to note that comparison for equality is a *double* equal sign, ==. Do NOT confuse this operator with the assignment operator which is a *single* equal sign, =. $a = b + c$ can *change* the value of a, but $a == b + c$ can *not* change the value of a.

operation	returns
$x == y$	true only if x is equal to y
$x < y$	true only if x is less than y
$x > y$	true only if x is greater than y
$x <= y$	true only if x is less than or equal to y
$x >= y$	true only if x is greater than or equal to y
$x! = y$	true only if x is not equal to y

Figure 3.1: Definitions of the comparison operators

Also note that Java provides slightly different versions of the following comparison operators that you may have seen in your math courses: $\leq, \geq, \neq$.

These comparison operators have *lower precedence* than the arithmetic operators. Hence, the expression `3 == 4 + 5` is the same as `3 == (4 + 5)`. The addition is done before the comparison.

Comparison for equality (or inequality) may be applied to reference types and to booleans:

```
myStudent == null         // true only if myStudent stores a null reference
myStudent != null         // compare for inequality
(x < 3) == false          // same as x >= 3
```

3.1.1 Exercises

1. What is the *value* of each of the following (assume `x` has been declared as an int variable)?

   (a) `3 != 4`

   (b) `-99 <= 3`

   (c) `7 - 7 == 2 / 3`

   (d) `x = 3`

   (e) `(2 < 5) == (3 > 5)`

2. *Rewrite* the following expression more succinctly without changing its meaning:

   `(x > 3) == (x < 3)`

3.2 Boolean operators

In chapter 1 we exposed the `boolean` type. This type consists of only two values: `true` and `false`. Do not think of these as strings of characters, nor as variables; they are constants, in the same way that 23 and -302 are constant values of type `int`.

3.2.1 AND, OR, NOT

Just as there are operations on numbers, there are operations on booleans. The basic operations are *or*, *and*, and *not*. These operations are represented by the operators ||, `&&`, and !, respectively, and are defined in Figure 3.2. Note that:

- The *or* operation produces a `false` result only when both operands are `false`.
- The *and* operation produces a `true` result only when both operands are `true`.

- The *not* operation is a *unary* operation; it has only one operand, to its right. It produces as a result the logical complement of its operand.

Figure 3.3 shows some examples of logical statements in English to further describe the meanings of these logical operations. These logical operations are fundamental to computer science and will be used extensively in writing Java programs.

Just as we had a formal definition of arithmetic expressions, we can provide a similar definition for boolean expressions:

A boolExpr may be:

1. `false`
2. `true`
3. boolean variable
4. expr comparison expr
5. boolExpr || boolExpr
6. boolExpr && boolExpr
7. ! boolExpr
8. (boolExpr)

in which `comparison` represents any of the six comparison operators shown in Figure 3.1, and `expr` is an arithmetic expression as defined in chapter 2. Again we have a definition in which we use the word we are defining, boolExpr in this case. As with arithmetic expressions a boolean expression is inherently recursive, and consequently there is no other way to define a boolean expression. An example of a boolean expression is:

```
!a || b && c
```

in which `a`, `b` and `c` are assumed to be declared as `boolean`.

Figure 3.4 and Figure 3.5 show how this boolean expression can be diagrammed as we did with arithmetic expressions in chapter 2. Once again we have an ambiguous definition: there are at least two different ways of diagramming the same expression.

This ambiguity is resolved as follows:

- ! *not* is executed before `&&` *and*
- && *and* is executed before || *or*

As usual, parentheses may be used to effect the desired order of operations. When you cannot remember these precedence rules, parentheses can be used even if not needed.

The logical *or* operation is designated by the Java operator `||`

x	y	x `||` y
false	false	false
false	true	true
true	false	true
true	true	true

The logical *and* operation is designated by the Java operator

x	y	x `&&` y
false	false	false
false	true	false
true	false	false
true	true	true

The logical *not* operation is designated by the Java operator ! and is a unary operation

x	! x
false	true
true	false

Figure 3.2: Definitions of the logical operations *or*, *and*, and *not*.

Statement	`true` or `false`
This book is written in French *or* elephants have 6 legs	`false`
This book is written in French *or* elephants have 4 legs	`true`
This book is written in French *and* elephants have 4 legs	`false`
This book is written in French *and* elephants have 4 legs	`false`
This book is written in English *and* elephants have 4 legs	`true`
This book is not written in French	`true`

Figure 3.3: Examples of logical statements in English

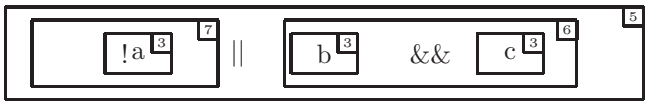

Figure 3.4: Applying the definition of boolean expression to: $!a||b\&\&c$

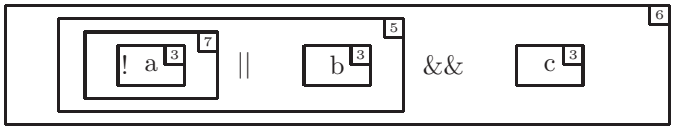

Figure 3.5: An alternative application of the definition of boolean expression to: $!a||b\&\&c$

We now see that Figure 3.4 represents the desired interpretation for the given boolean expression. In other words,

```
!a || b && c
```

is the same as

```
((!a) || (b && c)
```

All three of these boolean operators have lower precedence than the comparison operators:

```
 x < 3 || y == 0
```

is the same as

```
 (x < 3) || (y == 0)
```

3.2.2 Short circuit evaluation

If `b` represents any boolean expression, then we have the following identities:

- `true || b` is always true
- `false && b` is always false

To see this look at Figure 3.2 and substitute `true` (or `false`) for the operand `x`.

Java can make use of these identities to *optimize* the evaluation of boolean expressions (and provide a convenience for the programmer). When evaluating a boolean expression, the *left* operand of a logical operator is *always* evaluated first. If the operator is an OR (`||`) and the left operand is true, the result *must* be true; hence, there is no need to evaluate the right operand, and Java will not attempt to evaluate the right operand. Likewise, if the operator is an AND (&&) and the left operand is false, the result *must* be false; hence, there is no need to evaluate the right operand, and Java will not attempt to evaluate the right operand. This is called *short circuit* evaluation.

Note that this allows for an easy way to check, and avoid, possible run-time errors:

```
student != null && student.getGPA() === 4.0
```

The call to `student.getGPA()` will cause a null pointer exception if student is a null reference. But we avoid that error by checking for a null reference first, and the right operand of the && is not evaluated.

The following would *not* work correctly:

```
student.getGPA() == 4.0 && student != null
```

because the left operand of the && is always evaluated first; in this case it would cause a null pointer exception.

3.2.3 De Morgan's Laws

There are other logical identities which, in some cases, can simplify boolean expressions. Here we examine the two identities known as De Morgan's Laws. They can be expressed concisely, for boolean expressions `x` and `y`:

x	y	!(x && y)	(!x || !y)	!(x || y)	!x && !y
false	false	true	true	true	true
false	true	true	true	false	false
true	false	true	true	false	false
true	true	false	false	false	false

Figure 3.6: Proof of De Morgan's Laws

- `!(x && y) = !x || !y`
- `!(x || y) = !x && !y`

For example, the following two boolean expressions shown below are perfectly equivalent:

```
! (salary < 100000 && status==FULL_TIME)
salary >= 100000 || status!=FULL_TIME
```
[1]

The second version is perhaps a little simpler, and therefore preferable, though both versions would behave the same.

A proof of De Morgan's Laws is shown as a *truth table* in Fig 3.6. A truth table shows the evaluation of a boolean expression for all possible values of the variables. In this case there are two variables and therefore four rows in the truth table. The first of De Morgan's Laws is proved by noting that columns 3 and 4 are the same. The second of De Morgan's Laws is proved by noting that columns 5 and 6 are the same.

3.2.4 Exercises

1. Find the *value* of each of the boolean expressions shown below:
   (a) `4 > 3 && 4 < 2`
   (b) `(3 < 2 || 2 - 2 == 0) && 5 > 3`
   (c) `! true || 3 >= 2`
   (d) `! ( x < 2 || 3 > 1) || (x == 0 || true)`
2. Draw boxes around each boolean expression from the above problem, as shown in Figure 3.4. Be sure to show the rule numbers, and adhere to the precedence rules for boolean expressions.
3. Simplify the following boolean expressions (assume x and y have been declared as ints):
   (a) `(x == 0 || x != 0) && y < 0`
   (b) `y < 0 || (x == 0 || x != 0)`
   (c) `(x > 0 && x < 0) || y == 3`

[1] Note that the logical complement of < is >=

(d) `y == 3 && (x > 0 && x < 0)`

4. Assume x and y have been declared as ints. Which, if any, of the following expressions will cause a run-time error when the value of x is 0 (division by 0)?

   (a) `y/x > 3 || x == 0`

   (b) `x == 0 || y/x > 3`

   (c) `y/x > 3 && x != 0`

   (d) `x != 0 && y/x > 3`

5. Which of the following boolean expressions is *equivalent* to:
   `!(name.length()>12 && gpa < 1.0)` ?

   (a) `name.length()>12 && gpa < 1.0`

   (b) `name.length()<12 || gpa > 1.0`

   (c) `name.length()<=12 && gpa >= 1.0`

   (d) `name.length()<=12 || gpa >= 1.0`

6. For each of the following boolean expressions use one of De Morgan's Laws to show an *equivalent* boolean expression.

   (a) `!(name.equals("joe") && gpa <= 3.5)`

   (b) `name.equals("susie") && gpa == 3.5`

   (c) `name.equals("susie") || gpa == 3.5`

   (d) `!(name.equals("sue") || gpa > 3.5)`

3.3 One-way selections

One-way selections are used when we want to execute a statement, but only if a certain condition holds. The Java keyword `if` is used for this purpose. The general form is:

`if (boolean expression) statement`

The statement is executed only if the boolean expression is true. For example:

```
if (credits > 0)
    gpa = gradePoints / credits;
```

In this example, the value of the variable `gpa` will be changed to the result of the division, but only if the value of the variable `credits` is greater than 0.

A flow diagram for the one-way selection is shown in Figure 3.7 in which *Condition* represents the boolean expression in parentheses.

A few remarks on one-way selections:

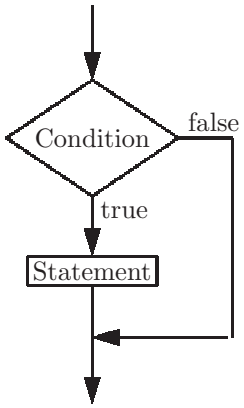

Figure 3.7: Flow diagram for a one-way selection structure

- The parentheses are *always* needed.
- The consequence of the `if` must be a *single* statement. If it is desired that several statements be executed when the condition is true, we will use a compound statement, defined later in this chapter.
- The condition in the parentheses must produce a `boolean` result, i.e. it must evaluate to either `true` or `false`.

The student should be cautious when using boolean expressions; a boolean expression which 'sounds ok' in English is not necessarily correct. For example, suppose we wish to calculate a student's gpa, but only if the number of credits is positive and less than 200. We may be tempted to write it as:

```
if (credits > 0 && < 200) gpa = ...
```

The compiler would reject this as incorrect. To see why, try to box the boolean expression using the rules of our definition; it cannot be done. This statement should be written as:

```
if (credits > 0 && credits < 200) gpa = ...
```

3.3.1 Exercises

1. Which of the following `if` statements contain syntax errors (assume `x` and `y` have been declared as int variables and `b` has been declared as a boolean variable)?

   (a)
   ```
   if x==0
      y = 3;
   ```

   (b)
   ```
   if (b)
      y = 3;
   ```

(c)
```
if (x > 0 && < 100)
   y = 3;
```

(d)
```
if (b = true)
   y = 3;
```

2. Show what would be printed by each of the following `if` statements (assume `x` and `y` are declared as int variables, and `b` is declared as a boolean variable):

(a)
```
x = 3;
y = 4;
if (x == y)
   System.out.println(x);
   System.out.println(y);
System.out.println ("done");
```

(b)
```
x = 3;
y = 5
b = x + y < 8;
if (b == true)
   System.out.println(x);
System.out.println ("done");
```

(c)
```
x = 3;
y = 5
b = x + y < 8;
if (b = true)
   System.out.println(x);
System.out.println ("done");
```

3. Show a better way of writing this `if` statement (which would prevent the slip-up exposed in part (d) of the above exercise):

```
if (b == true) ...
```

3.4 Two-way selections

Two-way selections are similar to one-way selections. The main difference is that two statements are provided in a two-way selection, exactly one of which must be executed. The general format is :

```
if (boolean expression)
   Statement1
```

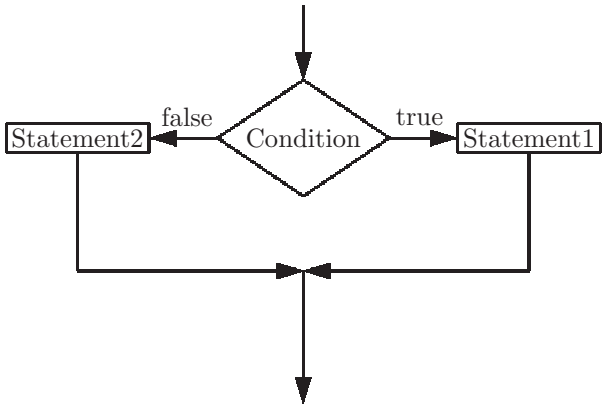

Figure 3.8: Flow diagram for a two-way selection structure

```
else
   Statement2
```

As with one-way selections, the boolean expression is evaluated. If it is true, Statement1 is executed. If it is false, Statement2 is executed.

A diagram of two-way selections is shown in Figure 3.8.

Note that (many of these have been noted previously with respect to one-way selections):

- The parentheses are *always* needed.
- The `true` consequence of the `if` must be a *single* statement. If it is desired that several statements be executed when the condition is true, we will use a compound statement, defined later in this chapter.
- The `false` consequence of the `if` must also be a *single* statement. This is the statement which comes after `else`. If it is desired that several statements be executed when the condition is false, we will use a compound statement, defined later in this chapter.
- The condition in the parentheses must produce a `boolean` result, i.e. it must evaluate to either `true` or `false`.
- Exactly one of the two statements must be executed because the boolean expression must evaluate to either `true` or `false`.

An example of a two-way selection is shown below:

```
if (credits >  0)
    gpa = gradePoints / credits;
else
    gpa = 0.0;
```

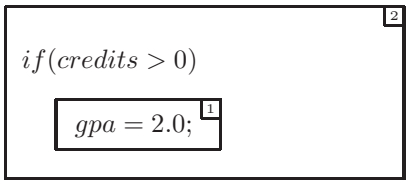

Figure 3.9: Applying the definition of Statement to:
$if(credits > 200)gpa = 2.0;$

In this example the variable `gpa` is set to the result of the division only if the variable `credits` is positive, and the variable `gpa` is set to 0.0 only if the variable `credits` is not positive. Notice that we have carefully *indented* both the true and false consequences of the `if`. This is not required by the compiler, but is done to make the program easier to read and maintain. The indentation of the two assigment statements is supposed to clarify the fact that they are part of the `if` statement.

We can now provide a preliminary definition of a Java statement, abbreviated stmt:

A Java stmt may be:

1. variable = expression ;
2. if (boolean expression) stmt
3. if (boolean expression) stmt else stmt

Once again, we have an inherently recursive construct; there is no way to define *statement* without using the word *statement* in the definition. Fortunately, rule 1 does not use the word *statement*, providing a base case.

Figure 3.9 shows how the definition can be applied to the one-way selection:

```
if (credits > 0)
   gpa = gradePoints / credits;
```

Figure 3.10 shows how the definition can be applied to the two-way selection:

```
if (credits > 0)
   gpa = gradePoints / credits;
else
   gpa = 0.0;
```

The definition of Stmt shown above indicates that both the true and false consequences of an `if` can be any Stmt, which would include `if` statements. In other words `if` statements may contain `if` statements, which in turn may contain other `if` statements, ... to any number of levels in depth. In other words `if` statements may be *nested* as deeply as you may wish. We now look at a more interesting example:

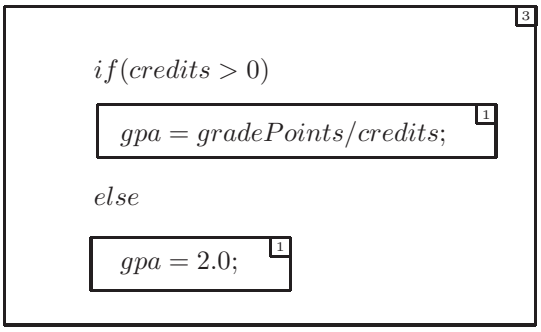

Figure 3.10: Applying the definition of Statement to: if credits > 0) gpa = gradePoints / credits; else gpa = 0.0;

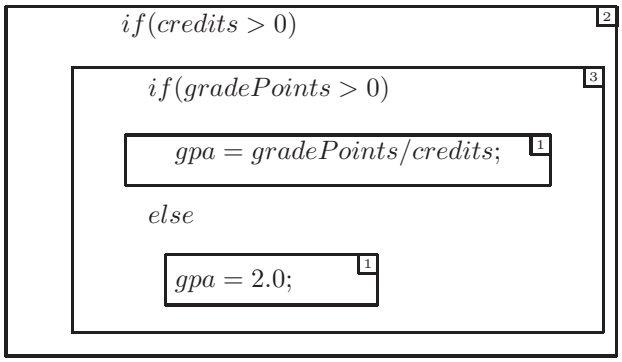

Figure 3.11: Applying the definition of Statement to an `if` statement which contains another `if` statement

```
if (credits > 0)
if (gradePoints > 0)
   gpa = gradePoints / credits;
else
   gpa = 2.0;
```

The indentation in this example is not good, but we will improve it after some discussion. Figure 3.11 shows how we can box all the statements using our definition of statement. However, Figure 3.12 shows a different way to box the statements. In Figure 3.11 we have a two-way selection inside a one-way selection; i.e. the `else` goes with the *second* `if`. This means that gpa will be set to 2.0 only if either credits or gradePoints is not positive.

In Figure 3.12 we have a one-way selection inside a two-way selection; i.e. the `else` goes with the *first* `if`. This means that gpa will be set to 2.0 only if credits *only* is not positive.

The fundamental question here is 'which `if` is matched with the `else`?' Could it be that once again we have an ambiguous definition? Yes, that is the

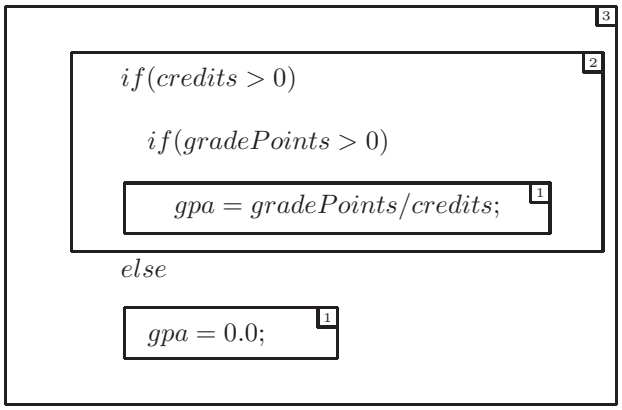

Figure 3.12: An alternate application of the definition of Statement to the same `if` statement containing an `if` statement

case, because there are two different interpretations for the same statement. This is a classic ambiguity problem in computer science, known as the *dangling else*. To resolve the ambiguity we state the following rule:

- Each `else` is matched with the nearest preceding unmatched `if`.

This means that the `else` should be matched with the second `if`, and we have a two-way selection inside a one-way selection. The correct interpretation is shown in Figure 3.11.

3.4.1 Exercises

1. Which of the following `if` statements contain syntax errors (assume `x` and `y` have been declared as int variables and `b` has been declared as a boolean variable)?

   (a)
   ```
   if (x >= 17)
         y = 2;
   else
         y = 3;
   ```

   (b)
   ```
   if (x >= 17)
         y = 2;
   else
         y = 3;
   else
         y = 0;
   ```

(c)
```
if (x >= 17 && < 25)
      y = 2;
else
      y = 3;
```

(d)
```
if (b)
      b = false;
else
      b = true;
```

(e)
```
if (x > 0)
    x = 2;
    y = 3;
else
      b = true;
```

2. Show what would be printed by each of the following `if` statements (assume `x` and `y` have been declared as int variables and `b` has been declared as a boolean variable):

(a)
```
x = 7;
y = 3;
if (x >= y+4)
   System.out.println (x);
else
   System.out.println (y);
   System.out.println ("false case");
```

(b)
```
x = 7;
y = 3;
if (x < 0)
   System.out.println (x);
else
   System.out.println (y);
   System.out.println ("false case");
```

(c)
```
x = 7;
y = 3;
b = x < y;
if (b)
   System.out.println (x);
else
   System.out.println (y);
```

(d)
```
x = 7;
y = 3;
if (x > y)
if (x <= y)
System.out.println (x);
else
System.out.println (y);
else
System.out.println (x + y);
```

(e)
```
x = 7;
y = 3;
if (x < y)
   if (x <= y)
      System.out.println (x);
else
   System.out.println (x + y);
```

3. Draw a box around each *statement* in the previous problem, as shown in Figure 3.11. Show the rule number applied, and be sure to resolve the dangling `else` correctly.

4. You are given a variable declared as:

   `Student stu;`
   Show a single `if` statement which will print stu's gpa if stu is not null, and will print `"null"` if stu is null.

3.5 Compound statements and scope

3.5.1 Compound statements

We noted in the previous section that the consequence(s) of an `if` in a one-way selection or in a two-way selection must be a *single* statement. However, it is often the case that we wish to execute (or not execute) more than one statement, i.e. a whole group of statements. The solution here is to use a *compound statement*. A compound statement is 0 or more statements enclosed in curly braces. This compound statement is treated as one big statement in a selection structure. For example:

```
double creditsAsDouble;
boolean active;          // true only if this Student is active
int tuition;             // current tuition in whole dollars
active = true;

if (credits > 0)
```

Program code	value of a	value of b	value of c
int a,b;			
a = 3;	3		
b = 7;	3	7	
{ // compound stmt	3		
int b;	3		
int c;	3		
b = 5;	3	5	
a = 9;	9	5	
c = 11;	9	5	11
}	9	7	

Figure 3.13: Scope of local variables

```
   {  creditsAsDouble = credits;
      gpa = gradePoints / creditsAsDouble;
   }
else
   {  tuition = 0;
      active = false;
   }
```

In this example, a group of two statements is executed if the condition is true and another group of two statements is executed if the condition is false.

Note that it is possible to have just one statement in a compound statement. Moreover, many instructors recommend this practice, because students often omit the curly braces when they are actually needed.

3.5.2 Scope of variables

Variables declared inside methods are called *local* variables, as opposed to *instance* variables (described in chapter 1). When a variable is declared inside a compound statement, the *scope* of that variable is limited to that compound statement; it is not known outside the compound statement. We say the variable is *local* to that compound statement. The scope of an instance variable is the entire class. It is possible to redefine a local variable in a separate scope as shown in Figure 3.13. When a method terminates, all local variables and parameters declared in that method are disposed from memory; they no longer exist.

Care must be taken to distinguish between local variables (or parameters) named the same as fields. Field names may be prefixed with `this.` to distinguish them from a local variable or parameter with the same name. A very common (and nasty) error is shown below:

```
// This method is in the Student class which has a field
// named gpa.
```

```
public void someMethod ()
{   int gpa = 3.4;
    System.out.println ("This student's gpa has been changed to 3.4");
    ...
}
```

The student's gpa has not been changed to 3.4 because the variable gpa is a *local* variable; it is declared in the body of the method. It is not the *field* gpa. When this method executes there are two, different, variables named gpa; one is a field and the other is a local variable. To refer to the field gpa, use `this.gpa`. To correct the problem, do not declare gpa as an `int`; do not declare it at all, and you will have just one occurrence of the variable gpa. The error described here is rather nasty because the compiler will not produce an error message; instead there will likely be a runtime error, a crash, or incorrect output. It may take several hours of work with the debugger to track down this problem.

3.5.3 Java statements - revisiting a formal definition

We can now expand our definition of statement to include compound statments.

A Java stmt may be:

1. variable = expression ;
2. if (boolean expression) stmt
3. if (boolean expression) stmt else stmt
4. method call, such as `System.out.println()`l
5. { stmt stmt stmt ... }

Figure 3.14 shows a diagram for the statement given above using this definition.

3.5.4 Exercises

1. Which of the following statements contain syntax errors (assume `x` has been declared as an int variable)?

   (a)
   ```
   if (x > 0)
      {  System.out.println ("positive");
   else
   System.out.println ("negative");
      }
   ```

   (b)
   ```
   if (x > 0)
      {  System.out.println ("positive"); }
   else
      {   }
   ```

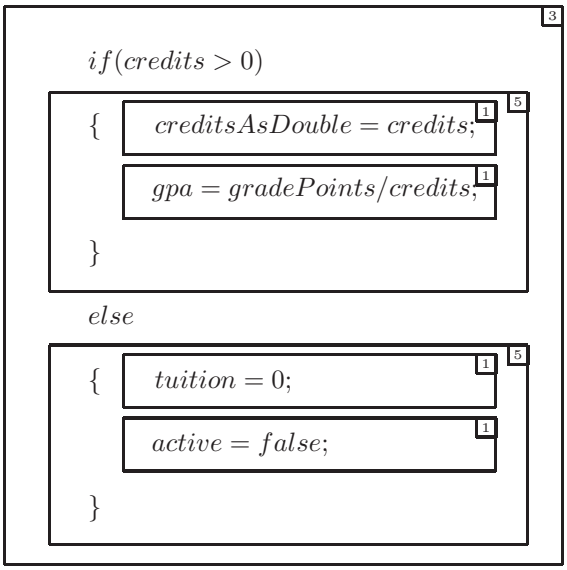

Figure 3.14: Applying the definition of Statement to an `if` statement containing compound statements

(c)
```
if (x > 0)
     System.out.println ("positive"); }
x = 0;
```

(d)
```
{ int y = 0;
  x = x + 1;
  y = x + 2;
}
```

2. Show what would be printed in each of the following:

(a)
```
{  int a = 7, b = 8;
   {  char b = '$';
      System.out.println ("a is " + a);
      System.out.println ("b is " + b);
   }
   System.out.println ("a is " + a);
   System.out.println ("b is " + b);
}
```

(b)
```
{  int a = 7, b = 8;
   {  char b = '$';
      a = 3;
```

```
            System.out.println ("a is " + a);
            System.out.println ("b is " + b);
         }
         System.out.println ("a is " + a);
         System.out.println ("b is " + b);
      }
```

(c)
```
      int x = 7;
      if (x >0)
         {
            x = x + 1;
            System.out.println (x);
         }
      else
            x = x - 1;
            System.out.printn (x);
```

(d)
```
      x = 12;
      if (x > 20)
         x = x + 1;
         System.out.println (x);
      System.out.println ("done");
```

3. Draw a box around each statement in the previouis problem, as shown in Figure 3.11. Show the rule number for each statement. For purposes of this problem, you may ignore variable declarations such as `int a = 7;`

3.6 Recursive methods revisited

As we mentioned earlier, methods can call themselves. Such a method is a *recursive* method. In order for this to work correctly, there are two criteria:

- There must be a path through the method which does not involve a call to itself. This is called the *base case*.
- The input(s) to the method, i.e. the parameter(s) must be reduced, in some way, when the method calls itself.

The mathematical function *factorial(n)* is defined to be the product of all whole numbers from 1 through n:

$factorial(n) = 1 \cdot 2 \cdot 3 \cdot ... \cdot n$

Sometimes this function is written with an exclamation point: n! For example, $factorial(4) = 4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$

Here is another definition of factorial:

$$factorial(1) = 1$$
$$factorial(n) = n \cdot factorial(n-1)$$

This definition can be used to write a recursive method which will return the value of factorial(n).

```
/** @param n is greater than or equal to 0
 *  @return the value of n!
 */
public int factorial(int n)
{  if (n<2)
      return 1;                  // base case, 1! = 1
   return n * factorial(n-1);    // recursive case
}
```

We should note a few aspects of this method:

- The API at the top has a few *javadoc keywords*:
  - `@param` is used to describe valid value(s) for any parameter to the method. This is a precondition. If the parameter `n` is less than 1, this method is not guaranteed to work.
  - `@return` defines what this method will return as its explicit result. This is an example of a postcondition (there could be others).
- The `if` statement checks for the base case; the case where the parameter's value is 1. In this case this call to the method is terminated immediately, returning the value 1.
- If the `if` condition is false, control falls through to the next statement, which is the recursive call, and the result of the multiplication is returned, terminating this call to the method.
- We could have used an `else` with the `if` but in this case it isn't needed because the `return` terminates the method.

Figure 3.15 depicts what happens when factorial(3) is called. It calls factorial(2), which in turn calls factorial(1), which returns a value, enabling the other calls to factorial to return values. The final result is 6.

This is our first example of a recursive method, and we will see others. It may surprise the student to learn that there are some problems in computer science which *must* be solved with recursive methods.

3.6.1 Exercises

1. Insert a print statement in the `factorial` method to print the value of the parameter, before the `if` statement. What is the output, when the parameter value is 5?

```
fact(3) = 3 * fact(2)
  fact(2) = 2 * fact(1)
   fact(1) = 1
  fact(2) = 2 * fact(1) = 2 * 1 = 2
fact(3) = 3 * fact(2) = 3 * 2 = 6
```

Figure 3.15: Execution of factorial(3)

2. The *fibonacci* sequence is

   1, 1, 2, 3, 5, 8, 13, 21, 34, ..
   Note that each number in this sequence is the sum of the two previous numbers. If `fib(n)` returns the n^{th} value in the sequence, we can say that:

   ```
   fib(1) = 1
   fib(2) = 1
   fib(n) = fib(n-1) + fib(n-2)   for values of n >= 3
   ```

   (a) What are the next two numbers after 34 in the fibonacci sequence?

   (b) Write a recursive method named `fib` which will return the n^{th} value in the fibonacci sequence. Use the definition given above.

   ```
   /** @return The nth value in the fibonnaci sequence.
    *  @param n must be greater than or equal to 1.
    *  This is a recursive method.
    */
   public int fib(int n)
   ```

3. Write a recursive method named `mult` with two int parameters (the second parameter must not be negative). It should return the product of the two parameters. Do *not* use the * or / operators; instead use the following:

   ```
   x * 0 is 0, for any value of x
   x * y is x + x*(y-1), for any positive value of y
   ```

   The API is:

   ```
   \** @return the product x*y, without using * or /
    *  @param y is not negative
    */
   public int mult(int x, int y)
   ```

4.

3.7 Comparing Strings and other reference types

Up to this point all of the comparison operations have involved primitive types. We now wish to discuss the comparison of reference types.

3.7.1 Comparison for equality or inequality

When comparing Strings or other reference types for equality, one should not use the == operator, nor the != operator. Instead one should use the method `.equals` (this method is defined in the String class), as shown below:

```
if (name.equals("joe") && gpa == 4.0)
   System.out.println ("joe is perfect");
```

The compiler will permit you to write the comparison as `name == "joe"` but in this case you are comparing the references, not the objects to which they refer. With Strings the `==` comparison will often work, but we can find cases for which it will not work correctly. A good habit is always to use `.equals()` when comparing reference types, unless you really mean to compare the references:

```
   if (name == null) ...  // check for null reference
```

To compare for inequality, use the logical NOT operator (!):

```
if ((! name.equals("joe")) && gpa != 4.0)
   System.out.println ("Somebody other than joe is not perfect");
```

The usage of `.equals` presumes that the class of the object to which the method is applied defines the meaning of `.equals` (the String class and most other classes in the Java class library take care of this for us). However, our Student class could define two Students to be equal if and only if they have the same name and same ssn.

Figure 3.16 shows object diagrams and the results of the two kinds of comparisons when applied to reference variables. In this object diagram the variables `s1` and `s2` refer to the same object; consequently they store the same value (the same reference), and the comparison `s1 == s2` will be true. However, the variables `s1` and `s3` refer to separate objects; consequently they store different references, and the comparison `s1==s3` will be false. The objects to which s1 and s3 refer are, presumably, equal since the corresponding fields are all equal (this assumes that the Student class provides a `.equals()` method, and the comparison `s1.equals(s3)` is true, as is the comparison `s2.equals(s3)`.

3.7.2 Ordered comparisons

To compare primitives for ordering, we can use one of the comparison operators given earlier in this chapter:

```
int x,y; ...
if (x < y) ...
if (x >= y) ...
```

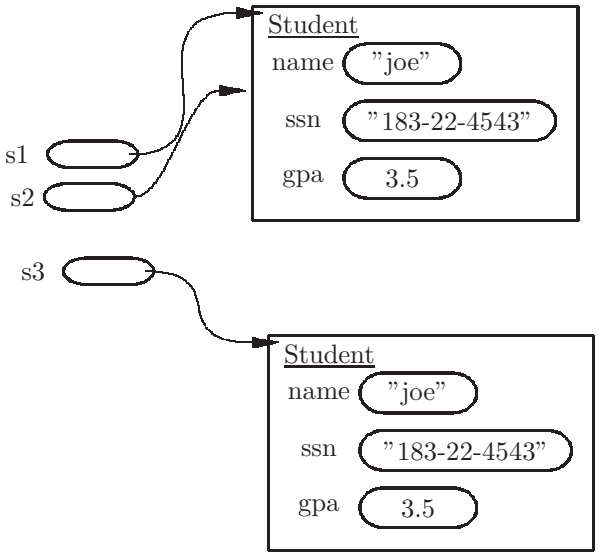

Figure 3.16: An object diagram showing the difference between comparison with == and comparison with .equals(): `s1==s2` is `true` and `s1==s3` is `false` and `s1.equals(s3)` is `true`

This cannot be done with boolean types; The comparison `true < false` will produce a syntax error at compile time.

The standard comparison method for reference types is `compareTo`, which has one parameter. It will compare *this* object with the parameter which should be an instance of the same class. The `compareTo` method will return an int which is:

- negative if *this* object is less than the parameter
- zero if *this* object is equal to the parameter
- positive if *this* object is greater than the parameter

If you have trouble remembering these return values, imagine that the returned value is like the result of the subtraction *this - parameter*.

When comparing Strings, the internal codes of the characters of the two Strings are compared, left to right. As soon as a discrepancy is found, the result is determined by the two characters being compared. In other words, comparison of Strings is essentially an alphabetic comparison: the String `s1` is less than the String `s2` if s1 precedes `s2` alphabetically. Examples of values returned by compareTo are shown in Figure 3.17. We will have more to say about compareTo in chapter 6.

comparison	result
`"abc".compareTo("def")`	negative
`"abc".compareTo("abc")`	zero
`"def".compareTo("abc")`	positive
`"abc".compareTo("aba")`	positive
`"abc".compareTo("abaci")`	negative
`"abc".compareTo("Bbc")`	positive
`"99".compareTo("100")`	positive

Figure 3.17: Returned values for the compareTo method, applied to Strings

3.7.3 Exercises

1. Which of the following contains a syntax error (assume the variables `s1` and `s2` have been declared as Strings)?

   (a)
   ```
   if (s1 == s2)
      System.out.println ("equal");
   ```

   (b)
   ```
   if (s1 < s2)
      System.out.println ("smaller");
   ```

   (c)
   ```
   if (s1.compareTo(s2))
      System.out.println ("smaller");
   ```

   (d)
   ```
   if (3.equals(4))
      System.out.println ("smaller");
   ```

2. What will be printed by each of the following (assume the variables `s1` and `s2` have been declared as Strings)?

3. (a)
   ```
   s1 = "john";
   s2 = "John";
   if (s1.equals(s2))
      System.out.println ("equal");
   else
      System.out.println ("not equal");
   ```

   (b)
   ```
   s1 = "john";
   s2 = "johnson";
   if (s1.equals(s2))
      System.out.println ("equal");
   else
      System.out.println ("not equal");
   ```

(c)
```
s1 = "john";
s2 = "johnson";
if (s2.equals(s1 + "son"))
   System.out.println ("equal");
else
   System.out.println ("not equal");
```

(d)
```
s1 = "john";
s2 = "johnson";
if (s2 == s1 + "son")
   System.out.println ("equal");
else
   System.out.println ("not equal");
```

(e)
```
s1 = "john";
s2 = "johnson";
if (s2.compareTo(s1) < 0)
   System.out.println ("smaller");
else
   System.out.println ("not smaller");
```

3.8 Selection structures in the GridWorld case study

3.9 Projects

1. The greatest common divisor of two whole numbers, x and y, is the largest divisor which they share. For example, if x=70 and y=30, the divisors of x are 2, 5, 7, 10, 14, 35 and the divisors of y are 3, 5, 10, 15. The greatest common divisor is 10.

   The Euclidean algorithm will find the greatest common divisor of two positive whole numbers efficiently. The algorithm can be summarized as follows:

   (a) Let r be the remainder when x is divided by y.

   (b) If r is 0, the greatest common divisor is y.

   (c) If r is not 0, the greatest common divisor is the greatest common divisor of y and r.

   Use this algorithm to define a method named `gcd` with two parameters, both of which should be positive integers. The method should return the greatest common divisor of its two parameters.

```
/** @return The greatest common divisor of x and  y.
 *  @param x is positive.
 *  @param y is positive.
 */
public int gcd (int x, int y)
```

2. Because there are not exactly 365 days in a solar year, the calendar must be corrected every four years by inserting an extra day. This is called a *leap year*. This is done in years which are divisible by 4 (2008, 2012, 2016, ...). However, this is not a perfect correction. Every hundred years we skip a leap year. 2100, 2200, 2300 will not be leap years even though they are divisible by 4. Yet another correction is necessary; every millenium will be a leap year even though it is divisible by 100 (2000, 3000, 4000, ...).

   Define a method named `leapYear` with one parameter, an int representing a year. The method should return a boolean – true only if the given year is a leap year.

   Hints:

   - Use the mod operator (%) to test for divisibility.
   - Check for least frequently occurring cases first: 1000, then 100, then 4.
   - Terminate the method with a `return` statement as soon as the result has been determined.

3. Roman numerals are often used to indicate the year that a movie was made; they are also sometines used in outlines. The Roman letters and their decimal equivalents are shown below:

Roman symbol	Decimal equivalent
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

   Some examples of roman numerals are shown below:

Roman symbol	Decimal equivalent
III	3
IV	4
V	5
VI	6
VIII	8
IX	9
X	10
LXIV	64
MMDCXXIX	2629

Write a method named `toRoman` with one parameter, an int. The method should return a String representation of the given int as roman numerals.

```
/** @return The Roman numeral representation of n.
 * @param n is in the range 1..3999
 */
public static String toRoman (int n)
```

Hint: Write a private helper method named `roman` with 4 parameters which returns a String:

- an int in the range 1..10
- The Roman symbol for 1, 10, 100, 1000
- The Roman symbol for 5, 50, 500
- The Roman symbol for 10, 100, 1000

If the helper method call is `roman (7, "X", "L", "C")`, its result will be `"LXX"`.

4. A circle in the x-y plane can be represented by three numbers: its radius and the x,y coordinates of its center. Two different circles in a plane can intersect in 0, 1, or 2 points. Define a method with 6 parameters, the specs for two circles, which will return the number of points in which the two circles intersect.

```
/** @return the number of intersection points of the two given circles.
 *  @param r1 is the radius of the first circle.
 *  @param x1 is the x coordinate of the center of the first circle.
 *  @param y1 is the y coordinate of the center of the first circle.
 *  @param r2 is the radius of the second circle.
 *  @param x2 is the x coordinate of the center of the second circle.
 *  @param y2 is the y coordinate of the center of the second circle.
 *  The parameters represent two different circles.
 */
public static int intersection (double r1, double x1, double y1,
                                double r2, double x2, double y2)
```

Because floating point numbers are not exact, you may use a tolerance value when comparing floating point values, e.g. `double epsilon = 1e-12`. Rather than comparing two distances for exact equality, compare for equality within this tolerance.

Hint: Define a private helper method which will calculate the distance between two points: $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

The method `sqrt` in the `Math` class will be helpful here.

Chapter 4

Iteration Structures

The computational power of computers lies in their ability to repeat a group of instructions a number of times. If it were not for this capability, computers would be as useful as old time 4-function calculating machines.

We call the section of code containing statements to be repeated a *loop*, probably because a diagram depicting the sequendce in which statements are executed would form a closed loop. Every loop has a *loop control* section and a *loop body*. The body is the statement to be repeated. The loop control section determines the number of times the body is repeated. There are several kinds of loop control, and we discuss two of them in this chapter - *while* loops and *for* loops.

4.1 Looping with `while`– pre-test loops

The `while` loops are most useful when the number of times a loop repeats cannot be calculated; it may depend on complex conditions arising as the loop body executes.

The format of a `while` loop is shown below:

```
while (boolean expression)
   statement
```

The intent here is that the statement forms the body of the loop; it is the part which is repeated. The body is performed repeatedly but only if the boolean expression evaluates to `true`. If the the boolean expression evaluates to `false`, the loop terminates, and control falls through to the next statement after the loop. A flow diagram depicting the flow of control for a `while` loop is shown in Figure 4.1.

An example of a loop is shown below:

```
credits = getCredits();
while (credits > 0)
```

```
{
 totalCredits = totalCredits credits;
 credits = getCredits();
}
```

In this example we presume there is a method, `getCredits()`, which will return the number of credits for a student (perhaps for one semester). Each time it is called it returns the number of credits for successive semesters. The value returned is added to `totalCredits` and the result is stored back into `totalCredits`. The variable `totalCredits` is used as an *accumulator*; i.e. it accumulates the total number of credits taken be a Student as the loop repeats.

We point out that:

- The loop body consists of *one* statement. If it is desired to have several statements in the body of the loop, a *compound statement* would be needed, as with selection structures; this is the case in the example shown above.
- The loop shown above will continue to execute as long as the variable `credits` is positive; when this variable is not positive, the condition is false, and the loop terminates.
- If the first call to `getCredits()` returns 0, the loop condition will be false, and the loop body will be executed 0 times.
- A `while` loop is often called a *pretest loop* because the test for continuation is done before the first execution of the loop body, as shown in Figure 4.1.

Notice that in the above example we have carefully *indented* the body of the loop so that it is clear what belongs to the loop and what does not belong to the loop. We did the same thing in chapter 3 with selection structures. It is important that the *appearance* of the program properly exposes the meaning, or *semantics*, of the program.

We now present another example of a pre-test loop. This is a mathematical example, taken from Calculus, though this example is accessible to those who have not yet had Calculus. The *exponential* function is often abbreviated as $exp(x)$ or as e^x (where the constant e is approximately 2.71828182846). Using Calculus we can derive an infinite series to calculate this function:

$exp(x) = e^x = 1 + x/1 + x^2/2! + x^3/3! + x^4/4! + ...$

To calculate this function exactly, we would need an infinite number of terms, something we really don't have the patience for. Consequently we will be satisfied with an *approximation* to the correct result by limiting the number of terms. The method is shown below:

```
/** Calculate exp(x) with a Taylor series
 *  @param x is not negative.
 *  @param epsilon is the tolerance to which the
 *  result should approximate exp(x).
```

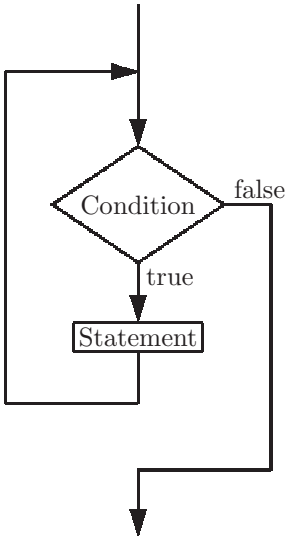

Figure 4.1: Flow diagram for a pretest loop structure

```
     *  @return exp(x), to within tolerance epsilon.
     */
    public double exp(double x, double epsilon)
    {
            double result = 1.0;
            double term = 1.0;
            int ctr = 1;                    // counter for the denominator
            while (term > epsilon)
                {   term = term * x / ctr;
                    result = result + term;
                    ctr = ctr+1;
                }
            return result;
      }
}
```

Note that:

- The parameter `epsilon` determines the degree of accuracy desired, since we are forced to return an approximation of the correct result.
- Each term is calculated using the value of the previous term. This is done by multiplying the numerator by `x` and multiplying the denominator by a counter (actually by dividing the term by the counter).
- When the value of `term` is sufficiently small (i.e. less than `epsilon`, it cannot provide a significant change to the result, and the loop terminates.

4.1.1 Infinite loops

The purpose of the loop control is to ensure that the body of the loop repeats the correct number of iterations. Care must be taken to make sure this is correct. It is possible that the loop control, if not correct, will cause the loop to repeat forever, with no termination. In other words, the boolean expression evaluates to `true` and never evaluates to `false`. This situation is called an *infinite loop* and is generally to be avoided.

An example of an infinite loop is shown below:

```
int x = 0;
while (x < 100)
    {  System.out.println ("x is " + x);
    }
```

In the loop shown above the boolean expression `x < 100` is `true`, so the print statement is executed. But the value of `x` is not changed in the body of the loop, so the boolean expression will continue to be `true`, and the loop will continue executing forever (or until the user intercedes by terminating execution with the IDE or by a system interrupt such as ctrl-alt-delete for Windows or ctrl-c for Mac OS). One way to correct this error would be to include a statement such as

```
x = x + 1;
```

in the body of the loop.

If your program contains several loops, and one of them is caught in an infinite loop, it will not be clear where the problem lies. In this case rely on a debugger to step through the statements of the program until the guilty loop control is detected.

Because of the complexity of software development, infinite loops can occur even in software which has been well tested. If you have ever noticed that your computer has "frozen", and moving the mouse or typing on the keyboard has no effect, the program (or operating system) is probably caught in an infinite loop, and is failing to accommodate input of any kind.

4.1.2 Exercises

1. Which of the following contain syntax errors (assume the variable `x` has been declard as an int)?

   (a)
   ```
   while (x > 2) x = x - 3;
   ```

   (b)
   ```
   while (x > 2 && < 10)
        { x = x - 3;
          System.out.println (x);
        }
   ```

(c)
```
while (x > 2)
     if (x < 0)
        System.out.println (x);
```

(d)
```
while (x > 0)
   x = x - 1;
   System.out.println (x): }
```

2. Show what would be printed by each of the following (assume the variable `x` has been declard as an int):

(a)
```
x = 4;
while (x >= 0)
   {  System.out.print (x);
      x = x - 1;
   }
```

(b)
```
x = 16;
while (x > 0)
   {  System.out.print (x + " ");
      x = x / 2;
   }
```

(c)
```
x = 1;
while (x < 5)
   {  System.out.print (x);
      x = x + 1;
   }
```

(d)
```
x = 10;
while (x < 5)
   {  System.out.print (x);
      x = x + 1;
   }
```

3. Which of the following is an infinite loop (assume `x` and `y` have been declared as ints)?

(a)
```
x = 12;
while (x > 0)
  {  System.out.println ("x is " + x);
     y = y + 1;
  }
```

(b)
```
x = 12;
y = 99;
while (x > 0 && y < 100)
  {  System.out.println ("x is " + x);
     x = x + 1;
  }
```

(c)
```
x = 12;
while (x == 0)
  {  System.out.println ("x is " + x);
     x = x + 1;
  }
```

(d)
```
x = 12;
while (x != 0)
  {  System.out.println ("x is " + x);
     x = x + 1;
  }
```

4. Define a method named `range` with two parameters which will print, on one line, all the whole numbers in the range from the first parameter to the second parameter. For example, `range(2,5)` should print 2 3 4 5.

```
/** Print, on one line, all whole numbers from
 *  low thru hi, inclusive.
 */
public void range (int low, int hi)
```

4.2 Looping with `for` – counter-controlled loops

In situations where we know, when writing the program, exactly how many times the loop should repeat, a `for` loop, or *counter-controlled loop* can be used.

4.2.1 Autoincrement and autodecrement

Before looking at the `for` statement, we would like to introduce an easy short cut which is frequently used in `for` statements.

A numeric variable can be incremented by 1 very easily by using the `++` notation, which is called *autoincrement*. The expression

```
x++;
```

is equivalent to

```
x = x+1;
```

This autoincrement operator increases the value of x by 1. When the following code has executed, the value of `x` will be 5;

```
int x;
x = 4;
x++;
```

It is possible to use the result of the autoincrement operator as part of a larger expression, but we do not recommend this usage.

Autodecrement is similar to autoincrement, but uses a `--` rather than `++`.

```
x--;
```

is equivalent to

```
x = x-1;
```

This autodecrement operator decreases the value of x by 1. When the following code has executed, the value of `x` will be 3;

```
int x;
x = 4;
x--;
```

4.2.2 The for loop

The format of a `for` loop is shown below:

```
for (declaration ; boolean expression ; expression)
   statement
```

As with `while` loops the statement forms the body of the loop, which is repeated. The control, however, is quite different. The loop control is achieved by the following sequence of events:

1. The declaration is established; it typically is a declaration of an `int` variable used to count the iterations. This variable is normally called the *loop variable*. This step is done just once, at the start, and never again for this execution of the loop.

2. The boolean expression is evaluated. If it is false, the loop terminates and control falls through to the next statement after the `for` loop. This boolean expression typically involves a comparison of the loop variable with some predetermined limiting value.

3. At this point the boolean expression is true, the statement (i.e. the loop body) is executed once.

4. The expression shown after the second semicolon in the `for` statement is evaluated. This expression typically involves an assignment to the loop variable.

5. Control returns to step 2 above to determine whether the loop body should be executed yet another time.

An example of a `for` loop which repeats the body exactly 10 times is shown below:

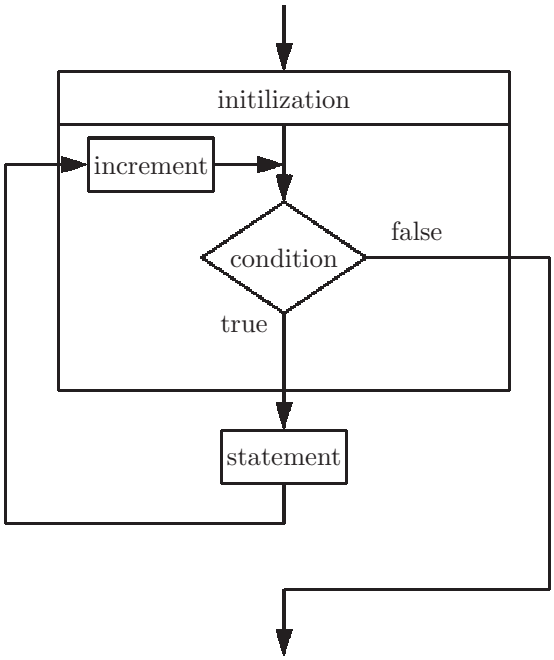

Figure 4.2: Flow diagram for a `for` loop structure

```
int sum = 0;         // declare and initialize an accumulator
for (int i=0; i<10; i++)
    sum = sum + i;
```

In this example when the loop terminates, the variable `sum` will contain the sum of the first 10 whole numbers (zero included). We note that:

- The loop variable is `i`. It is initialized to 0.
- The boolean expression `i<10` determines whether the body of the loop should be executed.
- The expression `i++` adds 1 to the value of `i` and stores the result back into `i`. It effectively increases `i` by one; we say that it *increments* `i`.
- In the loop body the statement `sum = sum+i` adds the value of `i` to the value of `sum` and stores the result back into `sum`.
- Since the loop variable is declared in the `for` statement, it is *local* to the loop. It is distinct from any variable of the same name declared outside the loop, and it will be disposed of when the loop terminates.

Figure 4.2 depicts the flow of control for a `for` loop.
Another example of a `for` loop is shown below.

```
int semesters = 7;
for (int sem=0; sem<7; sem++)
    totalCredits = totalCredits + getCredits(sem);
```

In this example it is known that the student attended for exactly 7 sememsters, so the loop should repeat exactly 7 times. Since the loop variable, `sem` is initially 0, and the loop continues to execute as long as the loop variable is *strictly* less than 7, the loop will repeat 7 times. Each time the loop repeats a method, `getCredits(int)`, is called; presumably that method returns the number of credits for a given semester (beginning with semester 0).

4.2.3 Exercises

1. Which of the following statments contain syntax errors?

   (a)
   ```
   for (int i=0; i<20; i++)
       System.out.println (i);
   ```

   (b)
   ```
   for (int i=0; i<20; i++)
       { System.out.println (i);  }
   ```

   (c)
   ```
   for (int i=0; i<20; i++)
       if (i > 10)
   ```

   (d)
   ```
   for (int i=0; i<20)
       System.out.println (i);
   ```

2. Show what would be printed by each of the following statements:

   (a)
   ```
   for (int i=0; i<5; i++)
       System.out.print (i-1 + " ");
   ```

   (b)
   ```
   for (int i=0; i<10; i++)
       if (i>5)
            System.out.print (i + " ");
   ```

   (c)
   ```
   for (int i=0; i<10; i++)
        System.out.print (i + " ");
        System.out.println ("again");
   ```

   (d)
   ```
   for (int i=0; i<10; i = i + 3)
      {
        System.out.print (i + " ");
        System.out.println ("again");
   ```

```
}
```

3. Write a method named `showRange`, with two int parameters which will print on one line all the whole numbers in the given range. Use a `for` statement.

```
/** Print all the whole numbers from low to hi, inclusive
 *  @param low The low end of the range.
 *  @param hi The high end of the range.
 */
public void showRange (int low, int hi)
```

4. Write a method named `roots`, with one int parameter. It should print a table of whole numbers from 1 to the given parameter, showing the square root of each of those whole numbers. Use a `for` statement.

```
/** Print a table of square roots for all whole numbers in the
 *  range 1..max, inclusive.
 *  @param max Should be positive
 */
public void roots (int max)
```

5. (a) Write a method named `sumInts`, with one int parameter, max. It should return the sum of all the whole numbers from 1 through max, inclusive. Use a `for` statement.

```
/** @return The sum of the whole numbers 1..max
 *  @param max Is positive
 */
public int sumInts (int max)
```

   (b) Use the internet to find a simple algebraic formula to do the same calculation without using a loop.

4.3 Equivalence of `while` and `for` loops

Every `for` loop can be written as an equivalent `while` loop which does exactly the same thing. The `for` loops are provided as a feature of Java (and most other programming languages) merely as a convenience for the programmer. Figure 4.3 shows how any `for` loop can be rewritten as an equivalent `while` loop.

```
for (init ; boolean expr ; expr)
    body
```

```
init ;
while (boolean expr)
   {  body
      expr ;
   }
```

Figure 4.3: Equivalence of `for` and `while` loops

4.3.1 Exercises

1. Show a `while` loop which is equivalent to each of the following `for` loops:

   (a)
   ```
   for (int i=0; i<10; i++)
       System.out.println (i);
   ```

   (b)
   ```
   for (int i=17; i>=0; i = i - 1)
       System.out.println (i);
   ```

   (c)
   ```
   int i=3, x;
   for (x=10; i>=0; x--)
       System.out.println (i+x);
   ```

   (d)
   ```
   for (;i>=0; i--)
       System.out.println (i);
   ```

2. Show a `for` loop which is equivalent to each of the following `while` loops:

   (a)
   ```
   int i=0;
   while (i < 10)
    {
       System.out.println (i);
       i++;
    }
   ```

   (b)
   ```
   int i=0;
   while (i < 10)
    {
       System.out.println (i);
       i = i * 2;
    }
   ```

   (c)
   ```
   int i=0;
   while (i != 10)
   ```

```
            System.out.println (i);
```

4.4 Nested loops

In our original definition of the `while` and `for` loops we stated that the loop body consisted of a single statement. This statement is not necessarily an assignment statement; it could be an `if` statement or another `for` or `while` statement. The loop body could also be a compound statement containing any number of assignment, `if` `while` , and `for` statements. In other words the body of a loop can itself be a loop; we call this a *nested* loop.

An example of a nested loop is shown below:

```
int sumOfProducts = 0;
for (int i=1; i<10; i++)
   for (int j=1; j<10; j++)
       sumOfProducts = sumOfProducts + i*j;
```

In this example we add up the values
$1 \cdot 1 + 1 \cdot 2 + 1 \cdot 3 + ... 1 \cdot 10 + 2 \cdot 1 + 2 \cdot 2 + 2 \cdot 3 + ... 2 \cdot 10 + ... 10 \cdot 1 + 10 \cdot 2 + ... 10 \cdot 10$

There is another shortcut called *assignment operators*. For any operator $\circ$, the statement

$x = x \circ y$

is equivalent to

$x \circ = y$

This means that the statement shown above,

```
sumOfProducts = sumOfProducts + i*j;
```

can be written more succinctly as

```
sumOfProducts += i*j;
```

4.4.1 Exercises

1. Which of the following contain syntax errors?

   (a)
   ```
   int x = 3;
   while (x < 10)
       for (int i=0; i<5; i++)
           System.out.println (i+x);
   ```

   (b)
   ```
   int x = 3;
   while (x < 10)
     {
       for (int i=0; i<5; i++)
           System.out.println (i+x);
       x += 2;
      }
   ```

(c)
```
{
 for (int i=0; i<10; i++)
     for (int j=0; j<10; j++)
}
```

(d)
```
 for (int i=0; i<10; i++)
     for (int i=0; i<10; i+)
       System.out.println ("i is " + i);
```

2. Show what would be printed by each of the following:

(a)
```
for (int row=0; row<2; row++)
  {
    System.out.println ("row " + row);
    for (int col=0; col<3; col++)
        System.out.print (col + " ");
    System.out.println ();
   }
```

(b)
```
int sum = 0;
while (sum < 20)
  {
    for (int i=0; i<4; i++)
        sum = sum + i;
    System.out.print (sum + " ");
   }
```

(c)
```
for (int row=0; row<5; row++)
  {
    for (int col=0; col<5; col++)
        System.out.print ("*");
    System.out.println ();
  }
```

(d)
```
int i=0, j=10
while (i+j < 12)
  while (i < 3)
     { j++;
       i++;
     }
System.out.println ("i is " + i ", j is " + j);
```

3. Define a method named `triangle` with one int parameter, which will print asterisks in the shape of a triangle with the given size. For example, if the size is 5, the output should look like this:

```
*
**
***
****
*****
```

```
/** Print a triangular shape of asterisks
 *  @param size Number of rows, and length of base
 */
public void triangle (int size)
```

4. Repeat the previous problem, but make the triangle upside-down. Name the method `triangleInv`

5. Define a method named `mult` which will display a multiplication table of the given size.

```
/** Print a multiplication table
 *  @param size Number of rows and columns, must be positive.
 */
public void mult (int size)
```

4.5 Definition of Statement - updated

Now that we have seen iteration structures, we can update our formal definition of a Java statement:

A Java stmt may be:

1. variable = expression ;
2. if (boolean expression) stmt
3. if (boolean expression) stmt else stmt
4. method call (e.g. System.out.println ())
5. { stmt stmt stmt ... }
6. while (boolean expr) stmt
7. for (declaration ; boolean expr ; expr) stmt

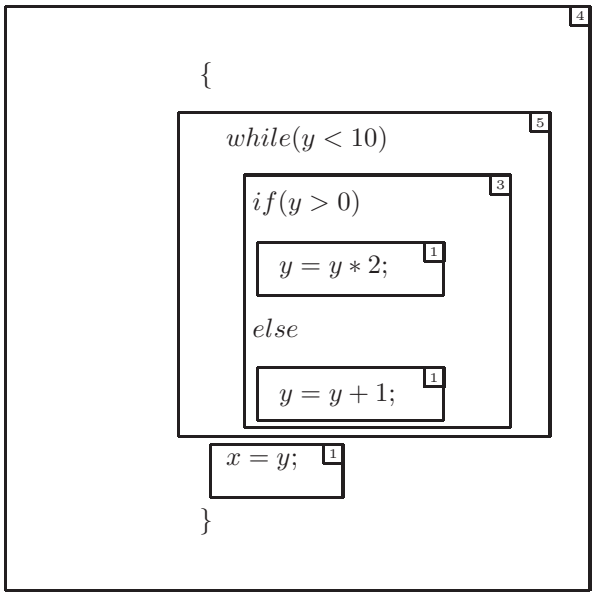

Figure 4.4: Applying the definition of Statement by drawing a box around each statement

As we have already noted our definition is recursive, and that allows statements to contain other statements, which in turn contain other statements, ... to any depth of containment levels.

Figure 4.4 shows how the definition of statement can be applied to the following:

```
{
   while ( y < 10 )
     if (y > 0)
        y = y * 2;
     else
        y = y + 1;
    x = y;
}
```

4.5.1 Exercises

1. Using the definition of a Java statement given in this section, draw a box around each statement shown below. Be sure to include the rule number in each box.

   (a)
   ```
   while ( i < 10)
       i = i+1;
   ```

(b)
```
while ( i < 10)
    if (x > 3)
        i = i+1;
```

(c)
```
if (x > 6)
   while (i < 10)
      System.out.println (i);
else
   { x = x - 1;
     System.out.println (x);
   }
```

4.6 Iterations in the GridWorld case study

4.7 Projects

1. (a) Rewrite the `fib` method from chapter 3 using a loop instead of a recursive method. `fib(n)` should return the n^{th} number in the fibonacci sequence.

   (b) Which appears to run faster, the looping version of fib or the recursive version?

   (c) What is fib(15)? fib(100)?

   (d) How can you explain the result of fib(100)?

2. A *prime* number is a whole number greater than 1, which is divisible only by 1 and itself. Some examples of prime numbers are: 2, 3, 5, 7, 11, 13, ... Define a method named `isPrime`. It should have one parameter, an int, and should return a boolean – true if the parameter is a prime number.

   ```
   /** @return true only if x is prime
    *  @param x is positive
    */
   public boolean isPrime (int x)
   ```

   Hint: Use the mod operator(%) to determine whether the given number is divisible by some other number.

3. The Taylor series expansion of the function *sin(x)* is shown below:

   $sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$

   Define a Java method to return the sin of a given value, to within a given tolerance.

```
/** @return sin(x)
 *  @param epsilon is the tolerance.
 */
public double sin (double x, double epsilon)
```

4. Write a method named `inBinary` which will return the binary represenation of a non-negative int as a String of 1's and 0's. For example if the parameter value is 18, it should return the String `"10010"`.

   ```
   /** @return The binary representation of x as a String
    *  of 0's and 1's.
    *  @param x is not negative.
    */
   public String inBinary (int x)
   ```

   Hint: Work from right to left in the binary representation of the given number. The mod operator (%) can tell you whether the low order bit is 0 or 1. The integer divide operator (/) can remove that bit.

5. Write a method to find an approximation to the square root of a double which is at least 1.0, to within a given accuracy, using a *bisection* algorithm:

   (a) Call the parameter x. Establish low and high boundaries for the result; low is 0 and hi is x.

   (b) Repeat the following as long as the result is not sufficiently accurate.

      i. An approximation to the result will be the average of low and hi.

      ii. If this approximation is too high, assign it to hi.

      iii. If this approximation is too low, assign it to low.

   ```
   /** @return an approximation to square root of x
    *  @param x is at least 1.0
    *  @param epsilon is tolerance for correct result
    */
   public double sqrt(double x, double epsilon)
   ```

Chapter 5

Collections, and Iteration Revisited

Thus far we have written programs which deal with small quantities of data. Each variable stores a single value, or a reference to a single object. However, most applications have a need to deal with large quantities of data; we do not wish to define a new variable for each data item that we need to store. Consequently we desire the capability of defining one variable which stores a reference to a *collection* of data items. These collections can be organized in a multitude of different ways, depending on how we wish to optimize the time required to access a particular value. In this chapter we examine a few different ways of organizing, and working with, collections of data items. We also begin a discussion of time efficiency, comparing the relative speeds of various operations.

5.1 Lists

A *list*, in mathematics and computer science, is defined as a collection of items with the following properties:

- Items may be added to, and removed from, a list; its size may be changed.
- The items have a particular order - the order in which they have been added to the list (though it may be possible to insert an item in the middle). The ordering of the items in a list is maintained.
- Each element has an *index* or position number associated with it. The index of the first item in the list is 0 (as with the positions of the characters in a String).
- There may be duplicated items in a list (some other collections do not have this property).

- Lists, as with other kinds of collections, are *homogeneous* – all items in a list are of the same type (we'll see considerable flexibility here, however, after we discuss inheritance).

The efficiency, or time required, for the operations of accessing, adding, removing items can vary depending on the particular implementation of the list; we shall discuss a few different implementations, noting their relative advantages and disadvantages.

The Java class library provides us with several kinds of lists. We'll begin by considering a particular implemenation called *ArrayList*.

5.1.1 Java packages and java.util

There are thousands of classes in the Java class library. Fortunately they are grouped and organized in such a way that we can deal with the ones we need and not worry about the others. A group of classes which share a common purpose or use can be grouped into a *package*. Packages, in turn, with similar purposes may be grouped together, forming a recursive hierarchy of packages. The package which deals with lists is named *java.util*. In order to use any of the classes in this package, those classes must be specified in an *import* statement at the beginning of your program. For convenience, we can have access to all classes in that package as follows:

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

5.1.2 ArrayList

The ArrayList class can be used if it is imported from java.util using the import statement given above.

5.1.2.1 Declaring and instantiating an ArrayList

There are actually a few different kinds of lists in the Java class library, though we are limiting our discussion to ArrayLists. To declare a variable which is capable of storing a reference to a list of Students, we should declare it as follows:

```
List <Student> roster;
```

This means that the variable `roster` can store a reference to any kind of list, including ArrayLists [1]. The value of the variable `roster` at this point is a *null reference*. It is a reference which refers to nothing at all; we have not yet even created the ArrayList. Because of the word `Student`, in angle brackets, this variable is capable of storing a reference to a List which stores references to Students *only*. This satsifies our requirement that all lists must be homogeneous.

In order to *instantiate* (i.e. to create an ArrayList instantiation instance of) the ArrayList we will use the `new` operator (as we did with the `Student` class):

[1] Technically, List is an interface which we cover in chapter 6, but since we believe in establishing good habits early on, we use it here

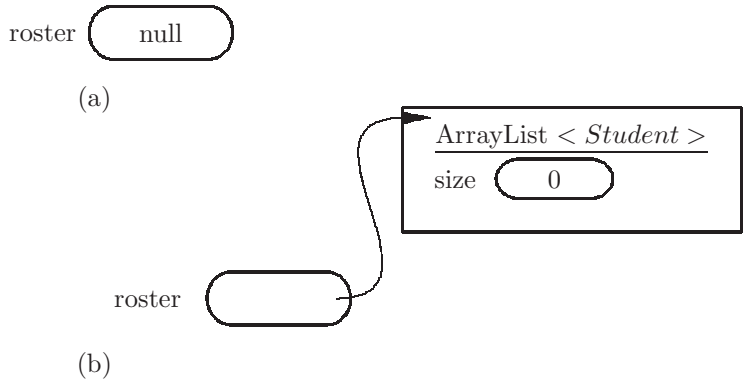

Figure 5.1: An object diagram showing the value of the variable `roster` (a) when it is declared and (b) after it has been assigned a value

```
roster = new ArrayList <Student>();
```

Notice that:

- Once again we specify the kind of objects (Student, in this case) to be stored in the ArrayList being created (recent versions of Java have relaxed this requirement).
- When we instantiate a list, we must specify what kind of list it is, in this case ArrayList.
- At this point the variable `roster` is no longer storing a null reference.

We now have an ArrayList containing zero items; its *size* is 0.

Notice also that Figure 5.1 depicts the process of declaring the variable `roster` and instantiating the ArrayList with an object diagram. In an ArrayList object we always show the size of the ArrayList as an int field, and the size is initially 0.

5.1.2.2 Adding or inserting items to an ArrayList

Once an ArrayList has been instantiated, items can be added using the *add* method. To add an item at the end of the list, supply one parameter, the item to be added:

```
Student s;
s = new Student ("joe", "256", 3.5);
roster.add (s);
```

This will add the new Student to the ArrayList referred to by `roster`. Note that lists, and all collections, do *NOT* store objects; they store *references* to objects as shown in the object diagram in Figure 5.2. Every time we add another

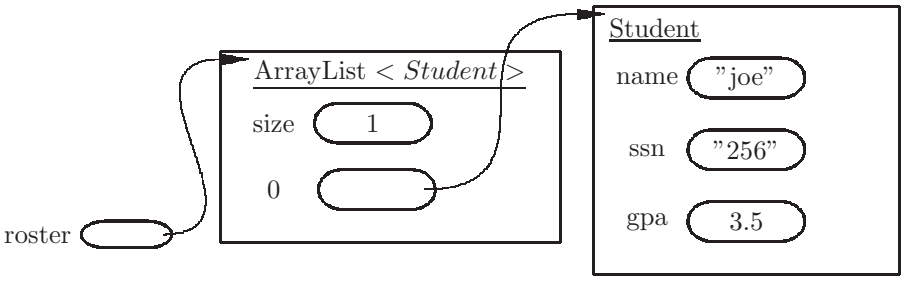

Figure 5.2: An object diagram showing the value of the variable `roster` after a Student has been added

student to the ArrayList, another reference is included in the ArrayList, and its size is automatically updated. Since we haven't looked at the source code for the ArrayList class, we'll simply make up our own field names and draw the diagram accordingly. For each reference added to the ArrayList, we show its position number (0 in Figure 5.2).

Figure 5.3 shows the object diagram for `roster` after a total of three students have been added to it.

Items can also be inserted at any position in an ArrayList, using the same method, but with an additional parameter. The first parameter is an int specifying the index, or position, at which the item is to be added. This form of the `add` method is really an *insertion*. The example below shows how a new Student can be added (inserted) at position 1:

```
roster.add (1,new Student ("joe", "304", 3.5));
```

Since it is not possible to have two items at the same position, all items with larger indices are automatically given higher index numbers. In other words, when adding at position 1 to a list of 3 items, the items formerly at positions 1 and 2 will be now be at positions 2 and 3.

Care needs to be taken when using this form of the `add` method. The index, or position, of the inserted item must be less than or equal to the current size of the list. If the index is negative, or greater than the size, a *run-time error* will occur, causing your program to come to a crashing halt. To insert an item at the beginning of the list, use an index of 0, and to insert at the end you can use an index equal to the current size (but its easier, and less risky, to use the `add` method with only one parameter).

Incidentally, this is our first example of two different methods with the same name. That's ok, as long as they have different parameter lists. They can be distinguished in a method call by the number (and types) of parameters. However, you cannot use differing return types to define two methods with the same name and same parameter types.

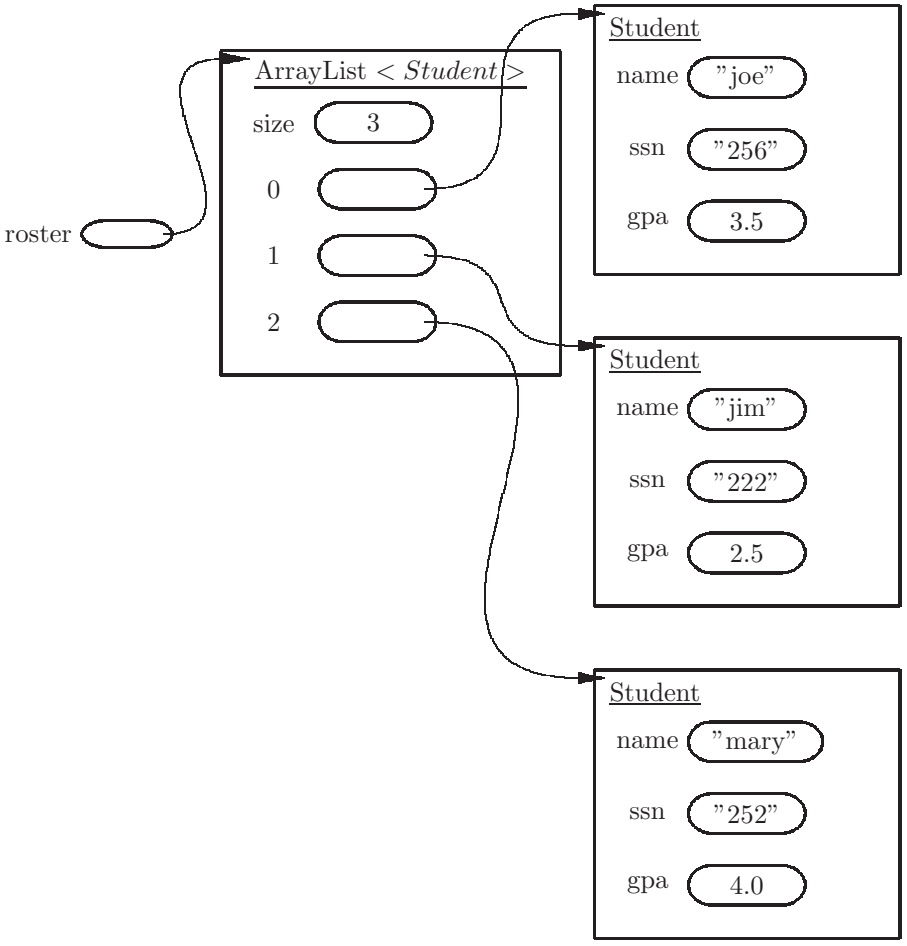

Figure 5.3: An object diagram showing the value of the variable `roster` after three Students have been added

5.1.2.3 Accessing or changing items in a list – get and set

After instantiating an ArrayList and perhaps adding several items to it, we may wish to discover the current value of some item in the list, or we may wisht to change the reference at a particular position. These operations are accomplished with the methods `get` and `set`.

To access the item at position 4 in our list, we could use the `get` method as shown below:

```
Student s;
s = roster.get(4);
```

This presumes that there are at least 5 Students in the list (remember, the first one is at position 0). The `get` method returns the reference which is at position 4, and the assignment stores that reference into the variable `s`. Again, we need to be careful that the given index is in the correct range: 0..size-1, otherwise a run-time error will occur.

The `get` method API is shown below:

```
/**
 *   @param ndx The position of the item to be returned,
 *              must be non-negative and less than size.
 *   @return The reference at the given index.
 */
E  get (int ndx)
```

The notation `E` in the code shown above is supposed to stand for the type of the items in the ArrayList. If it is an ArrayList of Students, E represents `Student`.

The `set` method is used to *store an item* at a given position in a list. For example,

```
Student s = new Student ("jim", "323", 3.2);
roster.set(12,s);
```

This code will replace the reference at position 12 with a reference to the new Student, jim, at that position. The indices of other items in the list will be unchanged. The compiler will insist that the type of the second parameter matches the type specified when creating the ArrayList, `Student` in this case.

The first parameter for both `get` and `set` must be an int in the range 0..size-1, otherwise a run-time error will result.

Note that there is an important difference between `set(int,E)` and `add(int,E)`. The `set` method does not change the size of the list, but the `add` method always increases the size of the list by one.

Primitive type	Wrapper class
int	Integer
float	Float
double	Double
boolean	Boolean
char	Character

Figure 5.4: Wrapper classes for primitive types

5.1.2.4 Collections of primitives

We've seen that collections, such as ArrayLists, store references to objects. But suppose we wish to store a collection of primitives, such as ints, floats, doubles, booleans, or chars? Java provides a 'workaround' for this situation, called *autoboxing* and *autounboxing*, using *wrapper* classes. The wrapper classes for the primitive types are shown in Figure 5.4. Each wrapper class is a simple class whose only purpose is to store a primitive value of the corresponding type. For example, an Integer object has a field with stores an int value, a Float object has a field which stores a float object, etc. When you wish to create a collection of primitives, use the wrapper class instead. For example, the code shown below will create an ArrayList to which int values can be added:

```
List <Integer> grades = new ArrayList <Integer>();
grades.add(100);
grades.add(90);
grades.add(75);
grades.add(100);
int first = grades.get(0);
System.out.println ("The first grade is " + first);
```

Note that when we add a primitive value (an int) to the ArrayList, Java automatically converts it to the corresponding wrapper class (Integer) so that the reference to it can be stored in the ArrayList. This is called *autoboxing*. Also, the `get` method returns a reference to the wrapper class (Integer), but we can store it into an int. This is called *autounboxing*.

A simpler way of restating the above paragraph is that you can have collections of primitives, but use the name of the corresponding wrapper class (shown in Figure 5.4) when declaring the collection.

We can now create a list of the first ten multiples of 10, using a `for` loop quite easily:

```
List<Integer> multiplesOf10;
multiplesOf10 = new ArrayList <Integer> ();
for (int i=0; i<10; i = i + 1)
   multiplesOf10.add (i*10);
```

5.1.2.5 A note on wrapper classes

Wrapper classes also provide other information on the corresponding primitive type. For example the `Integer` class has class variables `MAX_VALUE` and `MIN_VALUE` which store the largest possible int and the smallest possible int, respectively. Since they are class variables, the variable is preceded by the class name:

```
Integer.MAX_VALUE
Integer.MIN_VALUE
```

The wrapper classes are in the `java.lang` package, and do not need to be imported. See the API for more information on these wrapper classes.

5.1.2.6 Other operations on lists

The Java class library provides several other operations on lists; we will discuss a few of them here. For a complete list of operations see the API for `List` or `ArrayList` at `docs.oracle.com/javase/7/docs/api`.

We can *remove* the item at a given position from a list using the method shown below:

```
/** Removes the item at the given index from this list.
  * @param ndx is a valid index for this ArrayList,
  * ndx is not negative, and ndx is less than the size of this ArrayList.
  * @return a reference to the item removed.
  */
E  remove (int ndx)
```

As an example, we could remove the item at position 7 from a list named roster, and store the reference to the removed item in a variable:

```
Student s;
s = roster.remove(7);
// s now stores a reference to the removed student
```

The indices of all subsequent items in the list are decremented by one (no 'gap' is left in the list), and the size is also decremented by one. The `remove` method returns a value, but if we are interested in the removal only, and don't need the item that was removed, we can ignore the value returned:

```
roster.remove(7);
// the size of the list is decreased by 1.
```

In general, the value returned by a method can be used as part of an expression, or it can be ignored. If ignored, we say that we are using the method only for its *side effects* - in this case the removal of an item.

We can obtain the *size* of a list:

```
/**
  * @return the size of this list.
  */
int size()
```

We could print the size of a list as shown below:

```
System.out.println ("We have " + roster.size() + " students.");
```

We could use the `size()` method to determine whether a list is empty (if its size is 0), or we could use the *isEmpty* method:

```
/**
  * @return true only if this list is empty.
  */
boolean isEmpty()
```

The `isEmpty` method could be used as shown:

```
if (roster.isEmpty())
    System.out.println ("No students enrolled");
```

The `isEmpty` method returns a boolean, and therefore it can be used where a boolean expression is expected, as in an `if`, `while`, or `for` statement.

Many novice programmers would code the example shown above as:

```
if (roster.isEmpty() == true)
   ...
```

This would work, but we advise *against* this form. [2]

We can print out an entire list, simply be giving its name:

```
System.out.println (students);
```

All items in the list will be printed on one line, separated by commas, and enclosed in square brackets.

5.1.3 Exercises

1. Which of the following have syntax errors, and which will result in run-time errors (assume that the ArrayList class has been imported)?

   (a)
   ```
   List <Student> kids;
   kids = new Student();
   ```

   (b)
   ```
   List <Student> kids;
   kids = new ArrayList<Student>();
   kids.add ("jim");
   ```

[2] if the example had been `boolean b; if (b == true)...` then if the == operator is mistakenly typed as = (meaning assignment instead of comparision), the compiler will accept it, but the program will not work correctly, and many hours of debugging effort may be required to find the error.

(c)
```
List <Student> kids = null;
kids.add (new Student ("jim", "234"));
```

(d)
```
List <Student> kids;
kids = new ArrayList<Student>();
kids.add (new Student ("jim", "234"));
```

2. Show what would be printed by each of the following (assume the ArrayList class has been imported):

(a)
```
List <Student> kids = null;
System.out.printlin (kids);
kids  = new ArrayList <Student> ();
System.out.printlin (kids);
```

(b)
```
List <Student> kids  = new ArrayList <Student> ();
kids.add (new Student ("joe", "222"));
kids.add (new Student ("jim", "333"));
System.out.printlin (kids.size() + " kids");
```

(c)
```
List <Student> kids  = new ArrayList <Student> ();
kids.add (new Student ("joe", "222"));
kids.add (new Student ("jim", "333"));
System.out.println (kids.get(0).getName());
kids.remove(0);
System.out.println (kids.get(0).getName());
```

(d)
```
List <Student> kids, roster;
kids  = new ArrayList <Student> ();
roster = kids;
kids.add (new Student ("joe", "222"));
kids.add (new Student ("jim", "333"));
System.out.println ("size of roster is " + roster.size());
```

3. If `roster` is storing a reference to a list of at least 10 Students,

   (a) show how to print the name of the student at position 8 in the list.

   (b) show how to insert a new Student whose name is "alice" and whose ssn is "234" at the end of the list.

   (c) show how to insert a new Student whose name is "alice" and whose ssn is "234" at the beginning of the list.

   (d) show how to insert a new Student whose name is "alice" and whose ssn is "234" at position 3 in the list.

(e) show how to change position 7 of the list to refer to a new Student whose name is "jim" and whose ssn is "321".

(f) show how to remove the Student at position 7, and print that student's name, using just one statement.

4. Draw an object diagram showing the values of the variables num1, num2, num3, and num4 after the code shown below has executed:

```
List <Integer> num1, num2, num3;
num1 = new ArrayList <Integer> ();
num2 = num1;
num3 = new ArrayList <Integer> ();
num1.add (17)
num2.add (3);
```

5.2 Iteration revisited, with lists

Now that we have discussed loops and lists, we will see how to use a loop in conjunction with a list. This is a very common construct in programs: We have a list, or some other kind of collection, and we wish to 'visit' each item in the list, perhaps to examine it for certain properties, print it in a certain way, or even possibly to change it. Whatever the reason for visiting each item may be, it can be done easily with a loop, and a `for` loop is usually most amenable for this purpose.

For example, if the name of our list is `roster`, its original declaration might be:

```
List <Student> roster;
```

Suppose this list has been instantiated as an ArrayList, and many Student objects have been added to it, and we wish to print the name of each student. The loop can be controlled with a `for` statement as follows:

```
Student s;
for (int i=0; i<roster.size(); i++)
  { s = roster.get(i);
    System.out.println (s.getName());
  }
```

Note that:

- We use a local variable, `s`, to store a reference to a Student obtained from the list with the `get` method.
- The loop variable is initialized to 0, the position of the first item in the list.
- The loop will be terminated when the loop variable is *equal* to the size of the list (the last item in the list is at positon size-1).

- The loop variable is incremented by 1 each time the loop repeats
- The body of the loop is a compound statement in which we obtain a reference to a list item and print the name of the Student object to which it refers.

As a second example, we show a complete method which will determine whether a given list has at least one student with a perfect 4.0 gpa. This method uses a `while` loop, but it could also have been done with a `for` loop.

```
/**
 *  @return true only if there is a perfect student in the
 *  given list, students.
 */
public boolean hasAperfectStudent (List <Student> students)
{
   int i=0;                          // position of first item
   while (i<students.size())
     {  s = students.get(i);
        if (s.getGPA() == 4.0)
           return true;              // terminate the method
     }
   return false;                     // no perfect students found
}
```

In coding this method we are careful to make sure that it always returns the correct result, but we are also careful to make sure that it doesn't waste time needlessly. As soon as one student with a 4.0 gpa is encountered, we know the result should be `true`, and we terminate the method with `return true`. There is no need to continue looping. For small lists, the distinction may not be significant; even if we continue looping after finding a perfect student, it may take a small fraction of a second to complete its work. However, for large lists the time could be significant, and if the call to the method is itself inside a loop, the time could be very significant. We encourage the programmer to think about efficiency even at this early stage in learning to program.

If the loop runs to completion (all students in the list have been visited), we know that there could not have been any perfect students in the list, and the result should be `false`. In this case we terminate the method with `return false`.

5.2.1 Exercises

1. Define a method named `perfect` with one parameter, a list of Students. It should return the number of students in the list who have a perfect gpa of 4.0.

   ```
   /** @return The number of students who have a 4.0 gpa
   ```

```
 *  @param students Is not null
 */
public int perfect (List <Student> students)
```

2. Define a method named `average` with one parameter, a list of Integers. It should return the average of those integers, as a double.

```
/** @return The average of the given integers
 *  @param numbers Is not null and is not empty
 */
public int average (List <Integer> numbers)
```

3. Define a method named `sameName` with two parameters which will return a new list of students consisting of all those in the given list who have the same name as the second parameter.

```
/** @param students Is not null
 *  @param name Is the name to be matched
 *  @return List of all students with the given name
 */
public List <Student> sameName (List <Student> students,
                                String name)
```

5.3 Sets

We now discuss a different kind of collection known as the *set*. Sets are different from lists in two important aspects:

- The items in a set are not maintained in any particular order. The order in which items are obtained from the set may be completely different from the order in which they were added. The items could conceivably be reordered at any time.
- There are no duplicate items in a set. When an item is added to a set, if an item with the same value is already in the set, the size of the set is not changed. The new item is not added.

This means there will be no indices, or position numbers, for the items in a set. As with lists, a set will have a size - the number of items in the set, and sets must be homogeneous - the items in a set must all be of the same type.

The Java class library provides a few implementations of sets; we will limit our discussion to the one called *HashSet*. Like `ArrayList`, this class must be imported from `java.util`.

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

We will declare a variable to be of type `Set`, without specifying what kind of Set it may be. When instantiating it, however, we must specify the kind of Set, such as HashSet. We could create a set of whole numbers, and add several values to it:

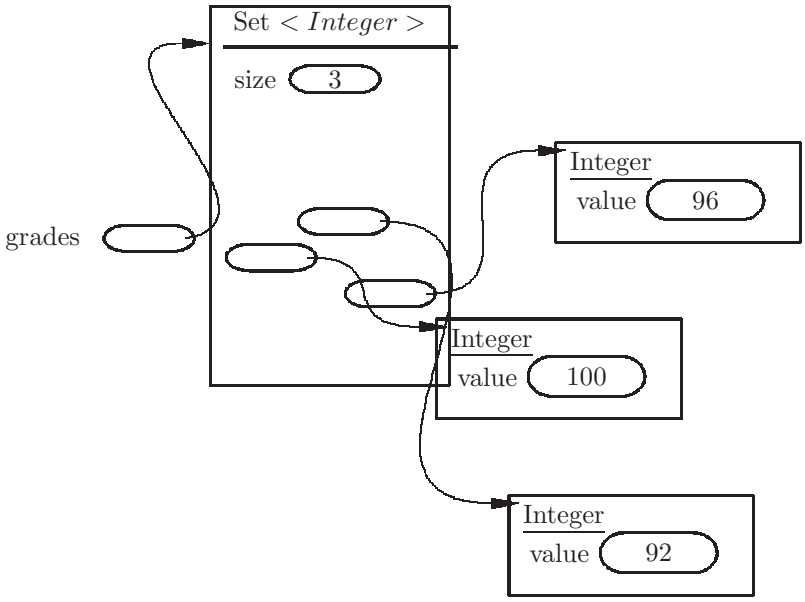

Figure 5.5: An object diagram showing a set of Integers after three items have been added. The items are stored in no particular order.

```
Set <Integer> grades;
grades = new HashSet <Integer> ();
grades.add (92);
grades.add (96);
grades.add (92);
grades.add (100);
grades.add (92);
```

At this point the size of the set would be 3 (the value 92 is added only once). In the next section we will see how we can obtain each item from a set, in a loop, possibly to print the values; in this case we would have no control over the order in which the values are printed, for example - 100,96,92. The implementation of the set controls the order. Figure 5.5 shows an object diagram for the set named `grades` created in the code segment shown above. Since we have not seen the source code for the `HashSet` class, we will simply show the references in a set in a nonlinear fashion, with no position numbers, to imply the lack of order in a set. As with ArrayLists, we will always show the size of the set, even if the size is 0.

Object diagrams can become rather large and complex. To alleviate this we will allow a slight 'shortcut': String objects and objects of wrapper classes (see Figure 5.4) may be treated as primitives; i.e. for a variable whose type is String, Integer, Double, Character, etc. we are permitted to show the value directly in the oval box, rather than as a reference to an object (which it really is). This

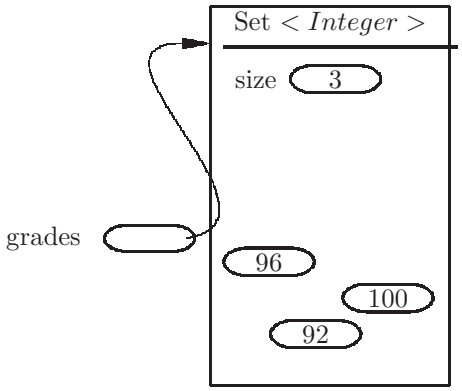

Figure 5.6: A simplified object diagram showing a set after three items have been added. Since the type of the items is Integer (a wrapper for int), they can be treated like primitives.

shortcut is possible only because the String class and the wrapper classes are *immutable*; i.e., it is not possible to change objects of these classes once they have been created. Figure 5.6 shows a simpler version of the object diagram in Figure 5.5.

Since sets do not have position numbers, the question may arise: How can we access individual items from a set – there are no `get` nor `set` methods? We will defer the answer to this quesion until the section on *iterators*. However, we can create a String representation of all the items in a set, and print it if we wish, very easily:

```
System.out.println ("The grades are " + grades);
```

This will convert each item in the set named `grades` to a String, concatenate them all together, separated by commas, into one long String inside square brackets. This is a nice feature of the HashSet class (and most other classes in the Java class library), and we will see how to provide the same feature for classes that we build ourselves.

One final remark with regard to sets is in order. If we were to create a set of Students, or some other class which we have defined ourselves, a few methods are required in that class: `equals(Object)` and (for HashSets) `hashCode()`. We will cover these in a later chapter. These methods are already provided in the String, Integer, and other classes in the Java Class Library, so they work fine.

5.3.1 Exercises

1. Which of the following contain syntax errors and which contain run-time errors (assume all classes from java.util have been imported)?

   (a)
   ```
   Set <Integer> numbers = new Set <Integer> ();
   numbers.add(18);
   ```

```
System.out.println (numbers);
```

(b)
```
Set <Integer> numbers = new HashSet <Integer> ();
numbers.add(18);
numbers.add(3);
System.out.println (numbers.get(1));
```

(c)
```
Set <Integer> numbers = new HashSet <Integer> ();
numbers.add(18);
numbers.add(3);
System.out.println (numbers);
```

(d)
```
Set <Integer> numbers = new HashSet <Integer> ();
numbers.add(18);
numbers.add(3);
numbers.remove(0);
System.out.println (numbers);
```

2. Show what would be printed by each of the following (assume all classes from java.util have been imported):

(a)
```
Set <Student> others, kids = new Set<Student> ();
others = kids;
kids.add (new Student ("jim", "321"));
kids.add (new Student ("joe", "333"));
System.out.println ("size is " + others.size());
```

(b)
```
Set <Integer> numbers;
for (int i=0; i<100; i++)
   numbers.add (i/10);
System.out.println ("size is " + numbers.size());
```

(c)
```
Set <Student> others, kids = new Set<Student> ();
kids.add (new Student ("jim", "321"));
kids = others;
System.out.println (kids);
```

3. Define a method named `copyToSet` with one parameter, a List of Students, which will return a Set of all the Students in the given List.

```
/** @return a Set of all Students in the List students.
 *  @param students, May be empty but not null.
 */
public Set <Student> copyToSet (List <Student> students)
```

5.4 Iteration through a collection with for-each, and extrema problems

The need to visit every item in a collection is very common in computer programs. As we have seen in a previous section, this can be done with a `while` or `for` loop when working with a list.

5.4.1 Iteration through a collection with for-each

In this section we introduce an easier, and better, way to visit every item in a collection. It is called a *for-each* loop (though the word *each* does not appear anywhere in this construct). The general format is:

```
for (type variable : collection)
    statement                      // loop body
```

An example is:

```
for (Student s: roster)
    System.out.println ("The student's name is " + s.getName());
```

Here we are assuming that `roster` is a list of Students. The loop control will ensure that the variable, s, will be assigned the next item in the ArrayList roster each time the loop repeats, beginning with the first Student in the list and ending with the last student in the list. If we read the loop control as: "for each Student s, in roster", even though the words 'each' and 'in' appear nowhere in the code, we have a fairly good understanding of the intent.

We now see how to obtain a reference to each item in a set. We can use a for-each loop (with sets there is no other way since sets do not have indices). The following example is a method with one parameter, a set of integers, which will return the average value of those integers, as a double.

```
/** @return the average value of the given numbers
 *  @param numbers A Set of Integers to be averaged.
 */
double average (Set <Integer> numbers)
{  int sum = 0;
   for (int i : numbers)
       sum = sum + i;                        // accumulate the sum
   return sum / (double) numerss.size();     // divide by size to get average
}
```

In the example above, we accumulate the sum of the numbers in an accumulator, `sum`, in the loop body. When the loop terminates, we calculate the average by dividing the sum by the number of numbers. Since we wish the division to be a floating point division, we cast the result of the size() method to a double, ensuring a floating point division, and return the result of the division.

We should take a little more care, however, in this example. It is always a mistake to divide by 0, even if the dividend is 0. In this example the size of the set could be 0 (the API doesn't specify that the set must not be empty). Therefore we need to check for an empty set before doing the division. Also, the API should warn us what the result will be if the parameter is an empty set. Here is an improved version of the above example:

```
/** @return the average value of the given numbers, or 0 if the
 *  the set is empty.
 *  @param numbers A Set of Integers to be averaged.
 */
double average (Set <Integer> numbers)
{  int sum = 0;
   if (numbers.isEmpty())
       return 0;                          // terminate the method
   for (int i : numbers)
       sum = sum + i;                     // accumulate the sum
   return sum / (double) nums.size();     // divide by size to get average
}
```

The API now clarifies what will be returned if the given set is empty, and there is no possibility of dividing by 0.

When working with sets, it is necessary to use a for-each loop rather than `while` or `for` loop. Moreover, we encourage the programmer to use for-each loops whenever possible, even if not required, for a few reasons:

- As you become familiar with for-each loops, it becomes clear that the syntax and logic is much more simple and clear than when using a `while` or `for` loop.
- We will see later that there are other list classes in addition to ArrayList; in this case for-each loops can be much more efficient.

For-each loops are easy and convenient; however they have one important drawback. The size of the collection involved cannot be changed in the body of the loop. This means that items cannot be added or removed.

5.4.2 Extrema problems

One very common task we will encounter in programming is the problem of finding the largest (or smallest) of a collection of values. This is called an *extremum* (singular) or *extema* (plural) problem. We suggest the follow strategy to find the smallest value in a list of values (with some simple modifications it can be used to find the largest):

1. Check to see whether the collection is empty, and handle this as a special case.

2. Initialize a local variable to the value of the first item in the collection. We will call this a 'candidate' for the smallest; it is the smallest we have seen thus far.

3. Set up a loop in which each item in the collection is visited.

4. If a visited item is smaller than the candidate, then the new value of the candidate should be the value of the visited item. This is now the smallest we have seen thus far.

5. When the loop terminates, we have visited all items in the collection, and the candidate will be the smallest

Here is a code segment in which we try to find the smallest number in an ArrayList of ints:

```
List <Integer> grades = new ArrayList <Integer> ();
// Assume the ArrayList grades has been assigned values
//  ....
int smallest;
if (!grades.isEmpty())             // Check size of the list
   {  smallest = grades.get(0);    // there is at least one number
      for (int grade : grades)     // for each grade
        if (grade < smallest)
          smallest = grade;        // new candidate for smallest
   }
// smallest grade is stored in the variable 'smallest'.
```

For our next example, we will define a method which will return a reference to the best student in a given set of students. Using the same strategy as the previous example, we examine the gpa of each student, maintaining a candidate for the best as we cycle through the loop:

```
/**  @return the student with the highest gpa from the given Set of
  *  students, or null if the set is empty.
  */
public Student getBest (Set <Student> school)
{  Student best = null;             // candidate for best student
   for (Student st : school)        // for-each
     {
       if (best == null)            // first student in the set
          best = st;                // is the best seen so far
       if (st.getGPA() > best.getGPA())
          best = st;                // new candidate for best student
     }
   return best;
}
```

In chapter 2 we discussed null references in connection with object diagrams, but in this example we are making use of a null reference to indicate that the method could not return anything meaningful in the case where it is given an empty set. This is a very common usage of a null reference; it is a reference which refers to no object at all. Note that the local variable `best` is initialized to a null reference. If there are no students in the given set, the loop will not execute, not even once, and the value of `best` will be null when the return statement is executed. Also note that we can compare a variable against `null` with an == comparison, as though we are comparing primitives (we are actually comparing references). The logic in the example above is slightly different from the logic used in the previous example, because sets have no `get(int)` method. If you define a method which may return a null reference, be sure to clarify in the API under what circumstances this will actually occur.

5.4.3 Exercises

1. Which of the following contain syntax errors?

   (a)
   ```
   Set <Integer> roster = new HashSet <Integer> ();
   for (Student st : roster)
       System.out.println (st);
   ```

   (b)
   ```
   Set <Student> roster = new HashSet <Student> ();
   for (int i=0; i < roster.size(); i++)
       System.out.println (roster.get(i));
   ```

   (c)
   ```
   List <Student> roster = new ArrayList <Student> ();
   for (Student st : roster)
       System.out.println (st);
   ```

   (d)
   ```
   Set <Student> roster = new HashSet <Student> ();
   for (Student st : roster)
       System.out.println (st);
   ```

2. Define a method named `showPositive` with one parameter, a Set of Integers, which will print all the values in that set which are positive.

   ```
   /** Print all numbers which are positive.
    */
   public void showPositive (Set <Integer> numbers)
   ```

3. Define a method named `trueBits` with one parameter, a List of Booleans, which will return the number of Booleans in the list which are true. Use a for-each loop.

```
/** @return The number of true values in the parameter, bits.
 */
public int trueBits (List <Boolean> bits)
```

4. Define a method named `longestName`, with one parameter, a Set of Students. It should return the name of the Student in the given list who has the longest name, or null if the list is empty.

```
/** @return The longest name of all students in roster, or null
 *  if roster is an empty list.
 */
public String longestName (Set <Student> roster)
```

5. Define a method named `smallestPositive`, with one parameter, a List of Doubles. It should return the smallest positive number in the given list, or null if the list is empty. Use a for-each loop.

```
/** @return The smallest positive number in the given list,
 *  or null if the list is empty.
 */
public Double smallestPositive (List <Double> numbers)
```

5.5 Iterators and selective removal from a collection

If you have been looking at the API for `ArrayList` and `HashSet`, you may have noticed that there are several *remove* methods available, which enable you to remove an item from a collection. These methods can be used if you read the API carefully. However, we may wish to remove some of the items from a collection as we visit all items in a loop. In this case we *cannot* use a for-each loop, since a removal would change the size of the collection. Instead we will use an *Iterator*.

5.5.1 Iterators

An Iterator is a class[3] in the Java class library which is designed to help you visit all items in a collection. An Iterator object needs to be obtained, after which it can be used to:

- Obtain the next item from the collection. The method signature is `E Next()`. It returns the next item in the collection (the return type, `E`, must match the type of the items in the collection).

[3]Iterator is actually an *interface* which is similar to a class; we will discuss interfaces in chapter 6.

- Check to see if there are more items in the collection. The method signature is `boolean hasNext()`. It returns `true` only if there are more items in the collection which have not yet been visited since the Iterator object was created.
- Remove the last item obtained from the collection. The method signature is `void remove()`. This is the best way to selectively remove items from a collection

.

Since Iterator is in the package java.util, it must be imported at the beginning of your source file:

```
import java.util.Iterator
```

One unusual aspect of Iterators is that they cannot be instantiated with the `new` operator; rather we use a method from one of our collection classes called `iterator()` which returns an instance of the Iterator class. Once we have the Iterator object, we can use it to control the loop as shown in the following example, in which we print the names of all students in the ArrayList `roster`:

```
Student s;
Iterator <Student> itty;        // itty is an Iterator object
itty = roster.iterator();       // instantiate with a method call
while (itty.hasNext())          // are there more students in the list?
   {  s = itty.next();          // yes, get the next student in the list
      System.out.println (s.getName());
   }
                                // All student names have been printed
```

Notice that we must specify the type of the iterator in angle brackets – it must match the type of the collection with which it is associated, in this case, Student. This is called a *generic* type. The above example could have been done (and probably should have been done - for clarity) with a for-each loop. It also could have been done with a `for` loop, using the `get(int)` method. However, as mentioned previously, there are other list classes, in addition to `ArrayList` which do not have efficient implementations of the `get` and `set` methods. For this reason, we are advised to use a for-each loop or an Iterator when possible. The next example, however, points out a stronger need for Iterators.

5.5.2 Selective removal

Often we wish to visit every item in a collection, and change the collection by removing some of the items as we cycle through the collection. In the following example, we are writing a method which is supposed to remove all the failing students from a given list of students.

```
/** Remove all students with gpa under 1.0 from the given list
 */
```

```
public void flunkOut (List <Student> roster)
{  Iterator <Student> it;
   it = roster.iterator();
   while (it.hasNext())
      {  s = it.next();
         if (s.getGPA() < 1.0)
            it.remove();          // the iterator does the remove
      }
}
```

In this example we are removing some of the students, and not removing others; we call this *removing selectively*. We could *not* have used a for-each loop since we are changing the size of the list. Conceivably, we could have done this without using an Iterator, but the logic is so convoluted and difficult that you would probably not get it correct; we will not even show you how to do this – instead use an Iterator.

Notice in this example that we *use the iterator to remove*; this is easy to forget. The `remove()` method in the Iterator class will remove the last item obtained by a call to `next`. The `remove()` method can be called only once per call to `next`.

5.5.3 Exercises

1. Which of the following contain syntax errors, and which contain run-time errors?

   (a)
   ```
   Set <Student> school = new Set <Student> ();
   school.add (new Student ("Jim", ""));
   for (Student st : school)
       if (st.getSSN().length() == 0)
           st.remove();
   ```

   (b)
   ```
   Set <Student> school = new Set <Student> ();
   school.add (new Student ("Jim", ""));
   Iterator <Student> it;
   it = school.iterator();
   while (it.hasNext())
     { Student st = it.next();
       if (st.getSSN().length() == 0)
          remove(st);
     }
   ```

   (c)
   ```
   Set <Student> school = new Set <Student> ();
   school.add (new Student ("Jim", ""));
   Iterator <Student> it;
   it = new Iterator (school);
   ```

```
while (it.hasNext())
  { Student st = it.next();
    if (st.getSSN().length() == 0)
       it.remove();
  }
```

(d)
```
Set <Student> school = new Set <Student> ();
Iterator it;
it = school.iterator();
while (it.hasNext())
  { Student st = it.next();
    if (st.getSSN().length() == 0)
       it.remove();
  }
```

2. Define a method named `retrieveStudent` with two parameters, a Set of Students, and a String representing an ssn. The method should return the Student in the set whose ssn matches the given ssn. If no students match the given ssn, the method should return a null reference. Use an Iterator to control the loop.

```
/** @return Any Student whose ssn matches the parameter ssn,
 *  or null if no such Student is found in the list.
 */
public Student retrieveStudent (Set <student> students, String ssn)
```

3. Define a method named `removeNegative` which will remove all the negative values from a given list of Integers.

```
/** Remove all negative values from nums.
 */
public void removeNegative (List <Integer> nums)
```

4. Define a method named `removeByName` with two parameters, a List of Students, and a student's name. It should remove all students with the given name from the List.

```
/** Remove all students with name given as parameter.
 */
public void removeByName (List <Student> roster, String name)
```

5. Consider the following method to print Students who have a GPA of 3.0 or greater:

```
/** Print students with GPA of at least 3.0
```

```
 */
public void showDeansList (List <Student> students)
{  Iterator <Student> itty = students.iterator();
   while (itty.hasNext())
      if (itty.next().getGPA() >= 3.0)
         System.out.println (itty.next() + " is on the Dean's List");
}
```

(a) Point out a subtle logic error in this method.

(b) Give an example of a list of students, for which this method will cause a run-time error.

(c) Give an example of a list of students, for which this method will not crash, but will produce incorrect output.

(d) Give an example of a non-empty list of students, for which this method will not crash, but will produce correct output.

(e) Show how this error can be corrected.

5.6 Arrays

The notion of ArrayList is actually a fairly recent invention in programming languages (late 1990's); ArrayLists are built upon a more primitive kind of collection called an *array*. The array dates back to the earliest days of programming languages; arrays provide a means for mapping the items in a collection directly to the computer's memory, for quick access to any item. Aside from some very different syntax in the usage of arrays, they differ from ArrayLists in the following ways:

- The size of an array cannot change, whereas an ArrayList can grow and shrink as a program executes.
- Arrays can be used without involving the Java class library (though there are classes designed to be used with arrays).
- No `import` statement is needed to use arrays.

To declare a variable as an array, use the following format:

```
type [] variable;
```

This means that the variable being declared does not store a single item, but a reference to an array of items, each of the same type. Then we can instantiate the array; it is at this point that we determine, once and for all, the size (or length) of the array:

```
variable = new type [length];
```

As an example we can create an array of ints, with length 12 as follows:

```
int [] grades;
grades = new int [12];
```

The length of the array can be determined when the program is run, but once the array is created, its length may not be changed:

```
int len;
len = getLength();    // get the length from a method call
int [] grades;
grades = new int [len];
```

The type of an array need not be a primitive type; we could create an array of 23 Student objects as shown below:

```
Student [] roster;
roster = new Student [23];
```

In this example each position in the array is storing a reference to a Student object. How would these Student objects be initialized? We would need to provide the Student class with a constructor that has no parameters, a *default constructor*:

```
// Default constructor for Student class
public Student ()
{  name = "";
   ssn  = "";
   gpa  = 0.0;
}
```

Once we have created the array, we can store a value of the appropriate type into any position of the array. As usual the position numbers begin at 0. The position of the last item in the array is length-1. This is done using square brackets, and the syntax is:

```
array-name [index] = expression;
```

For example:

```
grades[2] = 95;
```

This would mean that the value 95 is stored into position 2 (third item) of the array. Figure 5.7 shows an object diagram for the variable `grades`. Notice that since it is an array of ints, positions which have not been assigned an explicit value are given a default value of 0 (Warning: other programming languages might not be so kind as to do this for us).

To access a value from an array, again put the index in square brackets:

```
System.out.println ("The fourth grade is " + grades[3]);
```

Note that the index in square brackets can be any expression which evaluates

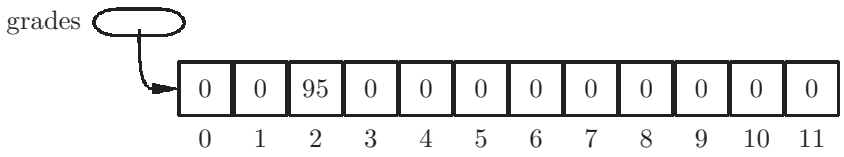

Figure 5.7: An array of ints, showing values and position numbers.

to an int. Be careful that the index is in the range 0..length-1, otherwise a run-time exception, `ArrayIndexOutOfBoundsException`, will cause your program to come to a crashing halt.[4] We have been using the word *length* instead of *size* with respect to arrays, because the length of an array can be obtained using the *field* name `length` (it is not a method name). An example which finds the average value of numbers in an array of ints is shown below:

```
// assume the array has been assigned values
int sum = 0;
double average;
for (int i=0; i<grades.length; i++)
   sum = sum + grades[i];
if (grades.length > 0)
   average = sum / (double) grades.length;
```

Because of the flexibility afforded by ArrayLists (they can grow and shrink), we will be using ArrayLists rather than arrays for most of our work. There are a few reasons, however, that programmers need to be aware of arrays:

- The ArrayList class is built using arrays; to understand how the class works one really needs to understand arrays.
- Some methods in the Java Class Library work with arrays rather than ArrayLists.
- When working as maintenance programmers, particularly when working with older, or so-called *legacy*, code, we will encounter arrays and should be able to read and understand the code.
- Arrays are closely associated with the hardware; arrays are mapped directly to the computer's memory. When you study *computer organization* or *computer architecture* you will gain a greater appreciation for arrays.

5.6.1 Initialization of arrays

Java offers a convenient way to initialize the elements of an array. The declaration can be followed by a list of items in curly braces, and separated by commas. Example:

[4] Other languages, such as C/C++ do not check for proper array indices; hence, they are vulnerable to *buffer over-run* attacks.

```
int []nums = {4, 83, -5, 0};
```

The length of the array is determined by the number of initial values in curly braces, 4 in this example.

An array of Strings can also be initialized in this way:

```
String [] names = {"joe", "jim", "sally"};
```

The length of this array is 3.

5.6.2 Passing arrays as parameters

A reference to an array may be passed into a method. As usual it is just the *reference* which is passed, not the entire array. This means that if the called method makes a change to the array, the calling method will see this update in the array; i.e. there are not two separate copies of the array.

To pass an array as a parameter, simply provide the name of the array as the actual parameter in the calling method, The formal parameter in the called method should include the type and empty brackets to show that it is an array, as shown in the example below.

```
   void callingMethod ()
   {   int [] numbers = new int[100];
       ...
       calledMethod(numbers);
       // numbers[2] is now -3
   }
...
   void calledMethod (int [] nums)
   {  nums[2] = -3;
   }
```

Note that, as usual, the name of the actual parameter (`numbers` in this case) does not have to be the same as the name of the formal parameter (`nums` in this case).

5.6.3 Vector product of numeric arrays

Arrays are often referred to as *vectors*. To find the vector product of two arrays which have the same length we simply multiply corresponding elements and sum the products. For example:

```
[2 5 7] x [3 -1 2] = 2 x 3 + 5 x -1 + 7 x 2
                   = 6 + -5 + 14
                   = 15
```

A method to find the vector product of two arrays is shown below:

```
/** @return the vector product of two arrays of int
 *  @param a1 and a2 have the same length
```

```
*/
int vectorProduct(int [] a1, int [] a2)
{  int [] result = new int [a1.length];
   int sum = 0;
   for (int i=0; i<a1.length; i++)
      sum = sum + a1[i]*a2[i];
   return sum;
```

5.6.4 Exercises

1. Which of the following contain syntax errors and which contain run-time errors?

   (a)
   ```
   int grades;
   grades = new int [15];
   grades[2] = 90;
   grades[3] = 10;
   grades[4] = grades[2] + grades[3];
   grades[grades[3]] = 17;
   ```

   (b)
   ```
   int [ ] grades;
   grades[2] = 90;
   grades[3] = 10;
   grades[4] = grades[2] + grades[3];
   grades[grades[3]] = 17;
   ```

   (c)
   ```
   int [ ] grades;
   grades = new int [15];
   grades[2] = 90;
   grades[3] = 10;
   grades[4] = grades[2] + grades[3];
   grades[grades[2]] = 17;
   ```

   (d)
   ```
   int [ ] grades;
   grades = new int [15];
   grades[2] = 90;
   grades[3] = 10;
   grades[4] = grades[2] + grades[3];
   grades[grades[3]] = 17;
   ```

2. ```
   int [] nums = new int [10];
   for (int i=0; i<10; i++)
      nums[i] = i * 10;
   ```

   Assuming the code shown above has been executed, show what would be printed by each of the following:
   ```

$$\begin{pmatrix} 4 & 8 & -3 & 0 & 4 \\ 0 & 0 & 3 & 0 & 2 \\ 14 & 0 & -3 & 1 & 1 \end{pmatrix}$$

Figure 5.8: An example of a matrix of numbers with 3 rows and 5 columns

(a)
```
for (int i=0; i<10; i++)
    System.out.print (nums[i] + " ");
```

(b)
```
for (int i=1; i<10; i++)
    System.out.print (nums[i] + nums[i-1] + " ");
```

(c)
```
for (int i=1; i<10; i++)
    nums[i-1] = nums[i];
System.out.println (nums[3]);
```

(d)
```
for (int i=1; i<10; i++)
    nums[i] = nums[i-1];
System.out.println (nums[3]);
```

3. Given an array of Students named `students`, show the code which could find the average GPA for those students.

4. Given an array of Students named `students`, show the code which could be used to find the position of the Student with the highest GPA.

5. Given an array of ints named `fib`, show the code which could be used to fill that array with the numbers in the fibonacci sequence.

5.7 Matrices: Two Dimensional Arrays

In mathematics we define a matrix to be an arrangement of values into rows and columns, in which all rows have the same number of values (and all columns have the same number of values).[5] Fig 5.8 shows a matrix of numbers, as you might see it in a mathematics textbook.

In Java we can implement matrices as two-dimensional arrays. To declare a two-dimensional array of ints, use two pairs of square brackets:
`int [][] myMatrix;`
Then to instantiate the array, specify the number of rows in the first pair of brackets, and the number of columns[6] in the second pair of brackets. For ex-

[5] In Java this restriction does not apply, as the number of columns in each row may vary, but we will not be concerned with this capability.

[6] Alternatively, one could think of the first dimension as the number of columns, and the second dimension as the number of rows.

ample, to work with a matrix of ints with 3 rows and 5 columns:
`myMatrix = new int[3][5];`

As with one-dimensional arrays, two-dimensional arrays can be declared and initialized in one statment (and the sizes need not be specified):

```
int [] [] myMatrix = {{2,3,4},
                      {7,2,0}};
```

In this example the array has 2 rows and 3 columns.

To access a particular value from a two-dimensional array, provide integer expressions for both the row and column:
`myMatrix[0][1]`
For this example, the value would be 3.

To change a value in a two dimensional array, one also needs to provide integer expressions for the row and column numbers:
`myMatrix[1][0] = 19;`
In this example, the 7 would be clobbered, and replaced by 19.

5.7.1 Examples of Matrix Arithmetic

5.7.1.1 Multiplication by a scalar

As an example of arithmetic involving two-dimensional arrays, we discuss the multiplication of a two-dimensional array of numbers by a single number. The single number, mathematically, is called a *scalar*. The product of a matrix multiplied by a scalar is simply a matrix of the same dimensions in which each element is multiplied by the scalar. For example:

$$\begin{pmatrix} 4 & 8 & -3 & 0 & 4 \\ 0 & 0 & 3 & 0 & 2 \\ 14 & 0 & -3 & 1 & 1 \end{pmatrix} \times 3 = \begin{pmatrix} 12 & 24 & -9 & 0 & 12 \\ 0 & 0 & 9 & 0 & 6 \\ 42 & 0 & -9 & 3 & 3 \end{pmatrix}$$

To do this in Java, we will use a nested loop. The outer loop will repeat once for each row in the matrix, and the inner loop will repeat once for each column in a row. In the inner loop we multiply each element of the given matrix by the given scalar to produce the corresponding element in the result. A method to multiply a matrix by a scalar is shown below:

```
/** @return the product resulting when the given matrix is
 *  multiplied by the given scalar.
 *  @param matrix is not empty
 */
public static int[][] multByScalar(int[][]matrix, int scalar)
{  int rows = matrix.length;
   int cols = matrix[0].length;
   int[][]result = new int[rows][cols];
```

```
        for (int row=0; row<rows; row++)             // outer loop
             for (int col=0; col<cols; col++)        // inner loop
                 result[row][col] = matrix[row][col] * scalar;
        return result;
    }
```

In this method, note that to obtain the number of rows and columns in the given matrix, we use the `length` variable:

```
rows = matrix.length;
cols = matrix[0].length;
```

Also note that the result matrix must have the same dimensions as the given matrix

5.7.1.2 Addition of matrices

To add matrices we simply add corresponding elements of the two matrices to produce the sum matrix. The two matrices being added must have the same dimensions. Below we show the mathematical sum of two matrices:

$$\begin{pmatrix} 4 & 8 & -3 & 0 & 4 \\ 0 & 0 & 3 & 0 & 2 \\ 14 & 0 & -3 & 1 & 1 \end{pmatrix} + \begin{pmatrix} 0 & -4 & 3 & 2 & -1 \\ 99 & 0 & -9 & 0 & 0 \\ 14 & 1 & 9 & 3 & 2 \end{pmatrix} = \begin{pmatrix} 4 & 4 & 0 & 2 & 3 \\ 99 & 0 & -6 & 0 & 2 \\ 28 & 1 & 6 & 4 & 3 \end{pmatrix}$$

To calculate a matrix sum in Java we use a nested loop, as in the previous section on multiplication by a scalar. In the body of the inner loop, we add corresponding elements of the two given matrices, to produce one element of the result matrix. A Java method to add two (non-empty) matrices is shown below:

```
/** @return the matrix sum of the two given matrices.
 *  @param The given matrices have the same dimensions, and
 *          neither of the given matrices is empty.
 */
public static int[][] add (int[][] m1, int[][] m2)
{   int rows = m1.length;
    int cols = m1[0].length;

    int[][]result = new int[rows][cols];
     for (int row=0; row<rows; row++)
        for (int col=0; col<cols; col++)
            result[row][col] = m1[row][col] + m2[row][col];
     return result;
}
```

5.7.1.3 Multiplication of matrices

Matrix multiplication is a bit more complicated than the other operations we have discussed. In order to multiply two matrices they must be *conformable*, i.e.

Dimensions of A	Dimensions of B	Dimensions of $A \times B$
`A[2][3]`	`B[3][5]`	`Product[2][5]`
`A[7][4]`	`B[4][9]`	`Product[7][9]`
`A[2][3]`	`B[5][3]`	Not conformable
`A[3][3]`	`B[3][3]`	`Product[3][3]`

Figure 5.9: Matrices must be conformable in order to be multiplied

$$\left(\begin{array}{ccc} \underline{2} & \underline{5} & \underline{3} \\ 3 & 0 & 7 \\ 2 & 0 & 3 \\ 1 & 1 & 4 \\ 6 & 6 & 6 \end{array}\right) \times \left(\begin{array}{c|c} 1 & 3 \\ 2 & 1 \\ 4 & 2 \end{array}\right) = \left(\begin{array}{cc} 2 \cdot 1 + 5 \cdot 2 + 3 \cdot 4 = 24 & . \\ . & . \\ . & . \\ . & . \\ . & . \end{array}\right)$$

Figure 5.10: Calculation of the value at row 0, column 0 in the multiplication of two matrices

they must have the correct dimentsions. If A and B are matrices, and we are to find the matrix product $A \times B$, then the number of columns in A must equal the number of rows in B. The number of rows in the result would equal the number rows in A, and the number of columns in the result would equal the number of rows in B. Fig 5.9 shows the dimensions of the result for several examples of matrix multiplication. Note that matrix multiplication is not commutative; i.e. $A \times B = B \times A$ is not always true.

We now explain how to find the matrix product $A \times B$, assuming the matrices are conformable. To find the value of row 0, column 0, in the product, we use the vector product of row 0 of A with column 0 of B, as shown in Fig 5.10

To find the value at row r, column c, of the product we use the vector product of row r in A with column c in B. The complete product of the two matrices shown in Fig 5.10 is shown in Fig 5.11.

A Java method to multiply nonEmpty matrices is shown below. We use a loop within a loop within a loop to calculate the product.

```
/** @return the matrix product of the two given matrices.
*  @param The number of columns in m1 must equal the number
*         of rows in m2.
*  @param Neither of the given matrices are empty.
*/
```

$$\begin{pmatrix} 2 & 5 & 3 \\ 3 & 0 & 7 \\ 2 & 0 & 3 \\ 1 & 1 & 4 \\ 6 & 6 & 6 \end{pmatrix} \times \begin{pmatrix} 1 & 3 \\ 2 & 1 \\ 4 & 2 \end{pmatrix} = \begin{pmatrix} 24 & 17 \\ 31 & 23 \\ 14 & 12 \\ 19 & 12 \\ 42 & 36 \end{pmatrix}$$

Figure 5.11: Multiplication of matrices

```
public static int[][] mult (int[][] m1, int[][] m2)
{   int rows = m1.length;              // rows in the result
    int cols = m2[0].length;           // columns in the result
    int n = m2.length;                 // conforming dimension
    int[][]result = new int[rows][cols];

     for (int row=0; row<rows; row++)
        for (int col=0; col<cols; col++)
            { int sum = 0;
              for (int i=0; i<n; i++)  // find vector product
                  sum = sum + m1[row][i] * m2[i][col];
              result[row][col] = sum;
            }
     return result;
}
```

Matrix multiplication has many applications in simulations, weather forecasting, statistics, economics, and engineering. Researchers are always looking for fast ways of multiplying huge matrices.

5.7.2 Exercises

1. Given the matrix, `m`, show the value of each expression shown below:

```
int [][] m = {{2,5,7},
              {3,0,-2}};
```

   (a) `m[1][0]`

   (b) `m[0][1] - m[1,2]`

   (c) `m[m[1][0] + m[1][2]][1]`

2. Using the matrix given in the previous problem, show the matrix (as in Fig 5.8) after each of the following statements has executed.

   (a) `m[0][1] = 17;`

   (b) `m[0][1] = m[0][0];`

   (c) `m[0][m[1][1]] = m[0][0];`

3. Given the matrix, `names`, show the value of each expression shown below:

```
String[][] names = {{"sal","jim"},
                    {"flo","Joe"},
                    {"ann","sal"}};
```

   (a) `names[1][1]`

   (b) `names[0][1] + names[1][0]`

(c) `(names[1][1] + names[2][0]).charAt(4)`

4. Given the following matrices, show the result of each operation shown below, if possible:

$$A = \begin{pmatrix} 4 & 8 & -3 & 0 & 4 \\ 0 & 0 & 3 & 0 & 2 \\ 14 & 0 & -3 & 1 & 1 \end{pmatrix} B = \begin{pmatrix} 2 & 5 & 0 & 2 & -1 \\ 9 & 0 & -9 & 0 & 0 \\ 4 & 3 & -3 & 1 & 1 \end{pmatrix} C = \begin{pmatrix} 1 & 3 \\ 3 & 1 \\ 2 & 2 \\ 0 & -1 \\ -2 & 5 \end{pmatrix}$$

(a) $A \cdot 3$

(b) $A + B$

(c) $A + C$

(d) $A \times B$

(e) $B \times C$

(f) $C \times B$

5. Write a java method to find the sum of a scalar plus a matrix. Simply add the scalar to each element of the matrix:

```
/** @return the sum of the given matrix and the given
 *          scalar.
 */
int[][] addScalar (int[][] m, int scalar)
{   . . .  }
```

6. Given the java code shown below, show what would be printed.

```
int m[][] = new int[3][4];
for (int r=0; r<3; r++)
   for (int c=0; c<4; c++)
       m[r][c] = r+c;

System.out.println (m[1][1]);
for (int r=0; r<3; r++)
   for (int c=0; c<4; c++)
       m[r][c] = m[r][(c+1)%4];

System.out.println (m[1][1]);
System.out.println (m[0][3]);
```

7. A Latin square is a square matrix of numbers in which:

   - The first row has no duplicate values.

- All values in the first row of the square appear in each row of the square.
- All values in the first row of the square appear in each column of the square.

Implement the methods shown in the API given below in the class `ArrayTester`.[7]

```
/** Exercise on two dimensional arrays
 *  @author ...
 *  @version (Feb 2019)
 */
public class ArrayTester
{
    /**@return an array containing the elements of column c of
     * arr2D in the same order
     * Pre:  c is a valid column index in arr2D
     * Post: arr2D is unchanged
     */
    public static int[] getColumn (int [][]arr2D, int c)
    {
        // put your solution here
    }

    /** @return true iff every value in arr1 is also in arr2
     *  Pre:  arr1 and arr2 have the same length.
     *  Post: arr1 and arr2 are unchanged.
     */
    public static boolean hasAllValues(int[] arr1, int[] arr2)
    {
        // put your solution here
    }

    /** @return true iff arr contains any duplicates */
    public static boolean containsDuplicates (int[] arr)
    {
        // put your solution here
    }

    /**  @return true iff the given matrix is a Latin Square
     *   @param:  square has equal number of rows and columns.
     *            square has at least one row.
     */
    public static boolean isLatin (int [][] square)
    {
```

[7] This was a free-response question on the 2018 Advanced Placement CS exam.

```
        // put your solution here
    }
}
```

5.8 Collections in the GridWorld case study

5.9 Projects

1. Implement the following Set operations:

   (a) Define a method named `union` which has two parameters, each of which is a Set of Strings. The method should return a new set consisting of all Strings which occur in either, or both, of the given sets.

   ```
   /** @return a new set which is the union of set1 and set2.
    */
   public Set <String> union (Set <String> set1, Set <String> set2)
   ```

   (b) Define a method named `intersection` which has two parameters, each of which is a Set of Strings. The method should return a new set consisting of all Strings which occur in both of the given sets.

   ```
   /** @return a new set which is the intersection of set1 and set2.
    */
   public Set <String> intersection (Set <String> set1, Set <String> set2)
   ```

   (c) Define a method named `difference` which has two parameters, each of which is a Set of Strings. The method should return a new set consisting of all Strings which occur in the first set, but not in the second set.

   ```
   /** @return a new set which is the difference, set1 - set2.
    */
   public Set <String> difference (Set <String> set1, Set <String> set2)
   ```

   (d) Define a method named `concat` which has two parameters, each of which is a Set of Strings. The method should return a new set consisting of all Strings which result from concatenating each String in the first set with each String in the second set. For example, if the two sets are: {`jim, tom, john`} and {`my, son`}. The result would be {`jimmmy, jimson, tommy, tomson, johnmy, johnson`}.

   ```
   /** @return a new set which is the concatenation of set1 with set2.
    */
   public Set <String> concat (Set <String> set1, Set <String> set2)
   ```

2. Poker.

(a) Define a class named `Card` with two fields, rank and suit, both of which are Strings. Include public accessor methods for these fields. Also define a method named `toString()` which returns a String representing a Card. For example, if the rank is `"Jack"` and the suit is `"hearts"`, this method would return `"Jack of hearts"`.

(b) Define a class named `Deck` with at least one field, a List of Cards. The constructor should initialize the field to the 52 different cards in a deck of playing cards. The suits are `"spades", "hearts", "diamonds", "clubs"`. The ranks are `"Two", "Three", ... "King", "Ace"`.

Define a method named `getCard` with one parameter, an int representing the position of a card in the deck. This method should remove the card at the given position from this deck, and return the removed card.

```
/** Remove the Card at position ndx from this Deck.
 *  @return the removed Card
 *  @param ndx is not negative, and less than the size of this Deck.
 */
public Card getCard(int ndx)
```

Define a method which returns the size of this Deck (i.e. the number of cards currently in this Deck).

(c) Define a class named `Poker` with at least one field storing a Deck. This class should have a method named `dealHand(int n)` which will return a List of n cards from its deck. Those cards should also be removed from the deck, in case `dealHand` is called more than once.

```
/** Deal a poker hand.
 * @return n cards from the deck.
 * These cards are removed from the deck.
 */
public List <Card> dealHand (int n)
```

Help: To deal cards randomly from the deck, use a random number generator from java.util, Random:

```
Random rand = new Random();
```

Each time you call the `nextInt(int n)` method, it will return a random int in the range [0..n-1].

Write a method to test your work by dealing 4 Poker hands, with 5 cards in each hand (there should be no duplicate cards).

(d) Change the Deck class so that it uses an array of Cards instead of a List of Cards. You should not need to change any other classes because the API for the Deck class is not changing, only the implementation is changing. This is an example of *object abstraction* which is discussed in chapter 6.

3. Sorting a list of numbers

   (a) Define a class named `Sorter`. The purpose of this class will be to store a List of numbers, and to arrange them in increasing order. This is called *sorting* and is one of the most important applications of computers. This class shold have one field, a list of Doubles.

   (b) Include a constructor which will initialize the list of Doubles, perhaps using a random number generator (see previous project).

   (c) Define a private method named `swap` with two int parameters. This method will exchange the values in the given positions of the array of Doubles.

```
/** Exchange positions j and k in the list of numbers
 */
private void swap (int j, int k)
```

   (d) Define a method named `posSmallest` with one parameter. It should return the position of the smallest value in the list, beginning at the given start position.

```
/** @return The positon of the smallest value in numbers,
 *  beginning at position start.
 */
private int posSmallest (int start)
```

   (e) Define a method named `sort`. It should arrange the values in the list in ascending order. It can do this with one easy loop in which it calls posSmallest and swap. For each position, p, in the list, swap it with the position of the smallest from p to the end of the list.

```
/** Sort the list of numbers in ascending order
 */
public void sort ()
```

   This project describes an *algorithm* known as *selection sort*. If you continue to study computer science, you will learn many other (sorting) algorithms.

4. Build a simulation for weather forecasts. Use several two dimensional arrays (all of which have the same dimensions):

   - A matrix in which each cell stores the temperature, in degrees Fahrenheit, at that location in the simulation
   - A matrix in which each cell stores the barometric pressure, in mm of mercury, at that location in the simulation
   - A matrix in which each cell stores the relative humidity, as a percentage, at that location in the simulation

Your simulation should update these arrays in a series of steps, enabling you to predict the air temperature, pressure, and humidity at some location any time in the future.

Work on the assumption that if the pressure is lower in a cell, than in a neighboring cell, that will cause air to flow from the high pressure cell to the low pressure cell. This will effectively change the pressure, temperature, and humidity in both cells. The extent to which these things change depends on how much air move, which is determined by the difference in pressure in the two cells.

Chapter 6

Abstraction, Inheritance, and Polymorphism

In this chapter we return to the process of *class design*. We deal with the question, How can we design classes which which work correctly and are easy to use and maintain? We also deal with issues related to duplicated code (undesirable), code reuse (desirable), encapsulation (desirable), and object-oriented design of software.

6.1 Software engineering

In the early days of software development the immense complexity of software was not well understood. People assumed that by putting enough programmers on a project, and by testing the software which they produced, a large and reliable software system could be produced in a reasonable time. However, this was not the case; there were many failed projects, and many projects were not reliable or were otherwise faulty. In cases where the software performed adequately, it was rarely delivered on time and within budget. In the meantime huge advances have been made in the field of computer hardware. The speed and memory capacity of computers have both improved at an amazing rate, along with improved reliability of hardware.

It has been said that if the automotive industry had accomplished technical developments on par with the developments of the computer hardware industry, a Rolls-Royce would have a top speed of 200 miles per hour, fuel economy of 500 miles per gallon, and would cost $13.50.

When your own personal computer crashes, it is almost always the result of a software error, and rarely the result of a hardware failure. Why has the field of computer software lagged so far behind the field of computer hardware? We feel it is largely due to the complexity of software. In order to deal with this problem, a discipline known as *Software Engineering* was established to apply engineering principles to the design and development of software. This

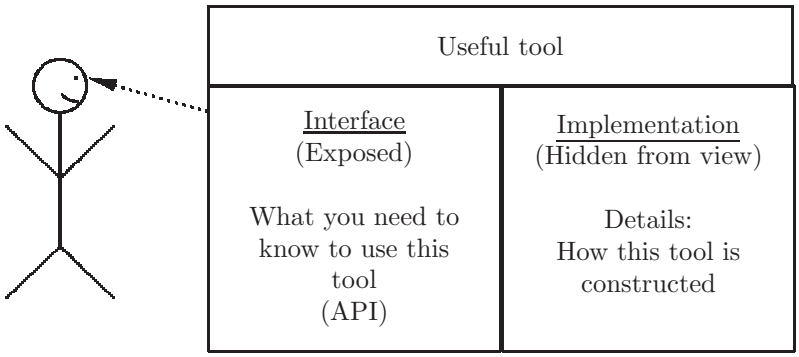

Figure 6.1: Abstraction: There is no need to know the implementation details of the tools being used.

chapter is essentially an introduction to software engineering, and in particular, object-oriented software engineering.

6.2 Abstraction

Because of the complexity of software, it is important that we be able to deal with a portion of a software system without having a detailed understanding of how the complete system works. Our goal will be to build tools (the Java Libary is a good example) which can be used to build other useful tools, which in turn are used to build other tools.... In the process we would like to be able to use a tool without worrying about its internal details – all we need to know is what the tool is supposed to do for us, and how can we use it properly. This is an example of *abstraction* and is depicted in Figure 6.1. Abstraction, in a more general sense, is the process of separating ideas from specific instances of those ideas.

6.2.1 Duplicated code

Suppose there is a segment of code in our program which seems to serve a useful purpose, and we find a need for this code segment in several other places in the program. For example, we are finding the average GPA of a set of Students:

```
// roster is a set of students
double average;
int sum = 0;
for (Student s : roster)
   sum = sum + s.getGPA();
average = sum / (double) roster.size();
```

Assume we have tested this code and it seems to be correct. Now we discover other places in the program where we need to find the average GPA of the

students in `roster`. It would be easy to copy and paste this code where it is needed.

Alternatively, we could write a method to accomplish this task, and simply call the method when needed. Clearly, that would make our program shorter (fewer lines of code), but memory and storage are cheap – this is not a problem (also, some programmers get paid by the line of code produced).

Suppose that rather than writing a method to find the average, we have copied and pasted this code segment in over 50 different places in our program. We now learn that sometimes the set of students, `roster`, might be empty. The code that we have introduced will not work when the set of students is empty, because we would divide by 0 to calculate the average gpa – this is a bug. By using copy and paste we have introduced over 50 bugs in our program. In many cases this cannot be fixed with a simple editor command to find and replace. Instead the programmer will have to search and find every place this code segment was used and correct it manually. This is even more of a problem if the faulty code had been pasted in many different source files. This is a serious problem resulting from *duplicated code*.

Now consider the alternate strategy; instead of using copy and paste, we define a method to find the average GPA:

```
/** @return the average gpa of the given set of students
 */
public double average (Set <Student> roster)
{
  int sum = 0;
  for (Student s : roster)
    sum = sum + s.getGPA();
  return sum / (double) roster.size();
}
```

We need to correct the mistake in *one place only*, the method which calculates average GPA. The corrected version is shown below:

```
/** @return the average gpa of the given set of students
  *  or 0.0 if roster is an empty set.
 */
public double average (Set <Student> roster)
{
  int sum = 0;
  if (roster.isEmpty())              // avoid division by 0
     return 0.0;

  for (Student s : roster)
    sum = sum + s.getGPA();

  return sum / (double) roster.size();
}
```

In every place where we need to find the average GPA we simply call the method which returns the average. This has two clear advantages over the copy and paste strategy:

- When the bug surfaces, we need to make the correction in *only one place*:
- It is now easy to find the average GPA for *any* set of students, not just the one named `roster`:

  ```
  double avg = average (someRoster);
  ```

This example points out the potentially disastrous pitfalls that can result from duplicated code. Try to avoid duplicated code if at all possible. The avoidance of duplicated code is one example of an abstraction.

6.2.2 Method abstraction

If we were to view the API for the `average` method in the previous section, without looking at the method body, this would be another example of abstraction, which we call *method abstraction* and is also called *control abstraction*. The API tells us how to use the method and what it returns; there is no need to look at the details of how it works.

Another example of method abstraction would involve using methods to build other methods. A generalized version of this concept would be the building of *software tools* to be used in the construction of other software tools. This idea is depicted in Figure 6.2.

As an example we consider the problem of arranging a list of items in correct order. For example, we may wish to arrange the items in a particular subsequence of a list in ascending order (a more general version of this problem is called *sorting* and was introduced as a project in chapter 5). In other words if we are given the list [4,0,9,2,1,6,-2] and we wish to arrange the sequence at positions 2,3,4 in order, the list would become: [4,0,1,2,9,6,-2].

We now define a method to arrange an arbitrary subsequence of length 3 in ascending order. Here is version 1:

```
/** Arrange the 3 items in the given list, at positions start, start+1,
 *  start +2 in ascending order.
 *  @param start The first position of the subsequence to be arranged
 *  order; start must not be negative and start must be less
 *  than numbers.size()-2.
 *  @param numbers A list of numbers, size is at least 3.
 */
public void sort3(ArrayList <Integer> numbers, int start)
{  int tmp;
   if (numbers.get(start) > numbers.get(start+1))
       {  tmp = numbers.get(start);       // swap first and second
          numbers.set(start, numbers.get(start+1));
          numbers.set(start+1, tmp);
```

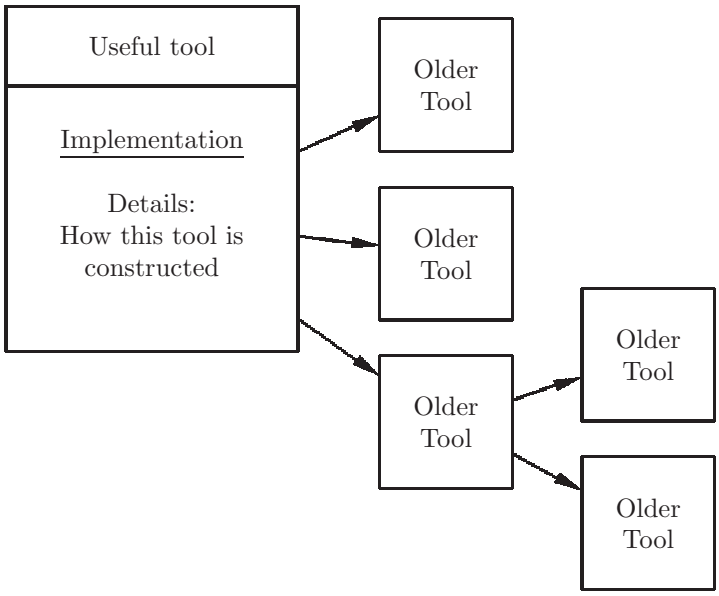

Figure 6.2: Abstraction: Build useful tools in order to build other useful tools

```
        }
    if (numbers.get(start+1) > numbers.get(start+2))
        {  tmp = numbers.get(start+1);       // swap second and third
           numbers.set(start+1, numbers.get(start+2));
           numbers.set(start+2, tmp);
        }
    if (numbers.get(start) > numbers.get(start+1))
        {  tmp = numbers.get(start);       // swap first and second
           numbers.set(start, numbers.get(start+1));
           numbers.set(start+1, tmp);
        }
}
```

In order to swap, or exchange, the values at two positions in the list, we use a local variable, `tmp`:

1. Store the first item in `tmp`.
2. Store the second item into the position of the first item.
3. Store `tmp` into the position of the second item.

Does the method shown above actually accomplish what we claim? We would recommend extensive testing before actually using this method – the logic is not simple (and many students have attempted this problem with much

more complicated logic). Rather than addressing this issue, we are going to use method abstraction to simplify this method.

The first thing we notice is that we have some code which is not perfectly duplicated code, but pretty close to it. Look at the sections where we are swapping the values at two positions in the list. Method abstraction suggests that this operation be done in a method, which is then invoked when needed; we'll call this method `swap`. Here is version 2:

```
/** Arrange the 3 items in the given list, at positions start, start+1,
 *  start +2 in ascending order.
 *  @param start the first position of the subsequence to be arranged
 *  order; start is not negative and start is less
 *  than numbers.size()-2.
 *  @param numbers A list of numbers, size is at least 3.
 */
public void sort3(ArrayList <Integer> numbers, int start)
{  if (numbers.get(start) > numbers.get(start+1))
         swap (numbers, start, start+1);
   if (numbers.get(start+1) > numbers.get(start+2))
         swap (numbers, start+1, start+2);
   if (numbers.get(start) > numbers.get(start+1))
         swap (numbers, start, start+1);
}

/** Exchange the values in positions first and second in the given list
 */
private void swap(ArrayList <Integer> numbers, int first, int second)
{  int tmp;
   tmp = numbers.get(first);
   numbers.set(first, numbers.get(second));
   numbers.set(second, tmp);
}
```

We have abstracted the process of swap to its own method. Our sort3 method is now easier to read (it is shorter), and if we had an error in our swapping, it would occur only once, in the `swap` method. Incidentally we have made `swap` a *private* method because we see no immediate need for this method outside of the class in which it exists. Here we are hiding details that are not needed by someone who is using the sort3 method. The private `swap` method is sometimes called a *helper* method.

We see version 2 as a considerable improvement over version 1, but we can do better. In version 3, shown below, we will define another private helper method to sort a subsequence of length 2. Then we can use that in our `sort3` method:

```
/** Arrange the 3 items in the given list, at positions start, start+1,
 *  start +2 in ascending order.
```

```
 *  @param start the first position of the subsequence to be arranged
 *  order; start is not negative and start is less
 *  than numbers.size()-2.
 *  @param numbers A list of numbers, size is at least 3.
 */
public void sort3(ArrayList <Integer> numbers, int start)
{   sort2(numbers, start);
    sort2(numbers, start+1);
    sort2(numbers, start);
}

/** Arrange the 2 items in the given list, at positions start and
 *  start+1 in ascending order.
 *  @param start the first position of the subsequence to be arranged
 *  order; start is not negative and start is less
 *  than numbers.size()-1.
 *  @param numbers A list of numbers, size is at least 2.
 */
private void sort2 (ArrayList <Integer> numbers, int start)
{  if (numbers.get(start) > numbers.get(start+1))
      swap (numbers, start, start+1);
}

.... swap method same as in version 2.
```

Method abstraction has once again provided a nice solution to this problem. Version 3 has some distinct advantages over version 2:

- We have shortened the sort3 method considerably by introducing another helper method.
- If we needed to define a sort4 method, to sort subsequences of length 4, it would be a fairly easy extension to what we have done; we could use just three calls to sort3.

6.2.3 Object abstraction and encapsulation

We would now like to address the issue of abstraction with respect to objects. Objects consist of state (fields or instance variables) and behavior (methods). Abstraction tells us to hide unnecessary details, so in designing a class we will make the fields *private*. This will have several advantages:

- Any programmer who needs to use our class will look at the API. They will not see anything which is private, and moreover they should not *need* to see anything which is private. The details are hidden from view.

- Any method in some other class (call it a *foreign* method) which tries to access a field in our class will not compile. This affords the following advantages:
  - Foreign methods are not able to store erroneous or non-valid values into our fields. If the field `gpa` in our `Student` class were public, a method in some other class would be able to assign it a negative value, which is clearly not appropriate.
  - We now have the freedom to change our mind about field names and types. Changing things which are public would require users of our class to recompile and retest – a major inconvenience.
- The fields constitue the internal state of an object of our class; all access from foreign classes should be through public methods.

The practice of making fields private is often referred to as *encapsulation*. Our fields are 'protected' from the abuse of foreign classes, just as the capsule of a pill protects the contents from the external environment.

6.2.4 Exercises

1. Define the method `sort4` which will arrange a subsequence, of length 4, of a list in increasing order. Use calls to the `sort3` method.

```
/** Arrange the 4 items in numbers at positions start through
 *  start+3 in ascending order.
 *  @param numbers A list of whole numbers with length at least 4.
 *  @param start A position in the the list; not negative and
 *  less than size of the list - 3
 */
public void sort4 (ArrayList <Integer> numbers, int start)
```

2. We have defined a method which will print the best Student in each of four lists of Students (see `ch6/TopStudents.java` in the code repository).

   Use method abstraction to eliminate duplicated code from the method `topStudents`.
   Hint: Define a private helper method to return the best Student in a List of Students.

3. We wish to find the prime factors of a given int. We will do this by building some useful tools first.

   (a) Define a method named `isPrime()` which determines whether a given int is prime.

```
/** @return true if n is a prime number
 */
private boolean isPrime (int n)
```

(b) We can now produce a list of prime numbers fairly easily. Define a method named `primeList()` with an int parameter, which returns a list of all prime numbers less than or equal to the given parameter.

```
/** @return list of primes less than or equal to n
 */
private List <Integer> primeList (int n)
```

(c) Use the primeList method to find the prime factors of a given int. Define a method named `primeFactors` which returns all the prime factors of a given int in a list.

```
/**
 * @return a list of the prime factors of n.
 * @param n > 0
 */
public List <Integer> primeFactors (int n)
```

6.3 Inheritance

We now wish to extend our Student class example. We have graduate students and undergraduate students, and there are some differences between these two kinds of students:

- Undergraduate students are allowed to participate in intercollegiate sports. Graduate students are not permitted to do so (NCAA rules).
- All graduate students hold a bachelor's degree; undergraduate students generally do not.
- Graduate students are permitted to register for courses at the 600 level; undergraduate students are not permitted to do so.
- All graduate students are registered for a 3-credit 'Thesis' course, which is not included in their GPA.

We could replace our Student class with two new classes, Undergrad and GradStudent:

```
/** An Undergraduate student has a name, an ssn, and a gpa.
 *  Also, undergrads are permitted to play sports.
 */
public class Undergrad
{
private String name;
private String ssn;
private double gpa;
private boolean athlete;
```

```
/** Construct a new Undergrad with the given name
 *  and ssn.  gpa is initially 0.0
 */
public Undergrad (String newName, String newSSN)
{  name = newName;     // initialize fields from parameters
   ssn = newSSN;
   gpa = 0.0;          // initialize field to default value
   athlete = false;    // initialize to default value
}

// accessor methods
/** @return The name of this Undergrad
 */
public String getName()
{   return name;  }

/** @return The ssn of this Undergrad
 */
public String getSSN()
{   return ssn;  }

/** @return The gpa of this Undergrad
 */
public double getGPA()
{   return gpa;  }

/** @return true only if this Undergrad is an athlete
 */
public boolean getAthlete()
{   return athlete;  }

// mutator methods
/** Change the name of this Undergrad to the given name
 */
public void setName (String newName)
{   name = newName; }

/** Calculate the gpa of this Undergrad, if
 *  if the number of credits is positive
 */
public void calcGPA (int gradePoints, int credits)
{  if (credits > 0)
      gpa = gradePoints / (double) credits;
}
```

```
/** Change the 'athlete' status of this Undergrad
 */
public void setAthlete(boolean ath)
{  athlete = ath;  }
}
```

We also have a Java class for graduate students:

```
/** A GradStudent student has a name, an ssn, and a gpa.
 *  Also, GradStudent has an undergrad degree
 */
public class GradStudent
{
private String name;
private String ssn;
private double gpa;
private String degree;

/** Construct a new GradStudent with the given name
 *  and ssn.  gpa is initially 0.0
 */
public GradStudent (String newName, String newSSN, String degr)
{  name = newName;     // initialize fields from parameters
   ssn = newSSN;
   gpa = 0.0;          // initialize field to default value
   degree = degr;
}

// accessor methods
/** @return The name of this GradStudent
 */
public String getName()
{   return name;  }

/** @return The ssn of this GradStudent
 */
public String getSSN()
{   return ssn;  }

/** @return The gpa of this GradStudent
 */
public double getGPA()
{   return gpa;  }

/** @return This the degree of this GradStudent
 */
```

```
public String getDegree()
{   return degree;  }

// mutator methods
/** Change the name of this GradStudent to the given name
 */
public void setName (String newName)
{   name = newName; }

/** Calculate the gpa of this GradStudent, if
 *  if the number of credits is more than 3.
 *  Assumes this GradStudent is registered for Thesis
 *  which is excluded from GPA.
 */
public void calcGPA (int gradePoints, int credits)
{  if (credits > 3)
      setGPA (gradePoints / (double) (credits-3));   // exclude Thesis
}
}
```

This may seem like a lot of work for some minor changes to our program, but with copy and paste it does not take long. The problem, as noted earlier in this chapter, is that there is a lot of duplicated code here (it's so easy to do that with copy and paste). Much of these two classes are identical. Fortunately, object oriented languages such as Java give us a way to eliminate this duplicated code; it is called *inheritance*. We can define a new clss which inherits the (non-private) fields and methods of an existing class, in the same way that a person might inherit the traits of their parents.

Inheritance allows us to define a new class using an existing class. The existing class is sometimes called a *base class* or a *superclass*. The new class is called a *subclass*. The subclass automatically has access to all fields and methods from the superclass which are not private. This means that we will be able to eliminate the duplicated code by using Student as a superclass, and Undergrad and GradStudent as subclasses. A superclass may itself be a subclass of some other class; moreover, a class may have more than one subclass, but may have only one superclass. There is an existing class called *Object* which is, directly or indirectly, a superclass of all classes.

A simpler way of restating the above paragraph is that inheritance forms a tree-like *hierarchy* of classes with a class named `Object` at the *root* (i.e. at the top). This hierarchy is depicted in Figure 6.3. Take note of a few aspects of this class diagram:

- The arrows always point from subclass to superclass, and *never* from superclass to subclass.
- The arrowheads are hollow (unfilled) triangles.

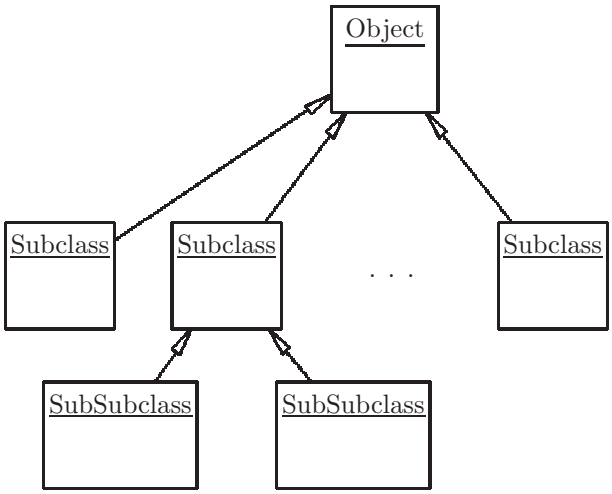

Figure 6.3: Class diagram showing Object at the root

- The format of a class diagram is specified in some detail by the Unified Modeling Language (UML) which we will not go into further at this time.

6.3.1 Is-a versus Has-a

To clarify the notion of inheritance we say that every instance of a subclass *is-an* instance of its superclass. In this case every Undergrad *is-a* Student, and every GradStudent *is-a* Student. But it is NOT the case that every Student *is-an* Undergrad; nor is it true that every Student *is-a* GradStudent. Inheritance is clearly a one-way street.

We should be careful to distinguish between inheritance and *composition*. Composition refers to the fields of a class. To describe the composition of the Undergrad class we would say that every Undergrad *has-a* name, every Undergrad *has-an* ssn, every Undergrad *has-a* gpa, and every Undergrad *has-an* athlete status.

Figure 6.4 shows some examples to help distinguish between *is-a* (inheritance) and *has-a* (composition).

When designing classes for your program, if it makes sense that every X *has-a* Y, then you should make Y a field in the X class. If it makes sense that every X *is-a* Y, then X should be a subclass of Y. As you'll see if and when you study object-oriented design in more depth, there will be cases where it is not clear whether inheritance or composition is appropriate; in such cases it is usually better to use composition.

Inheritance	Composition
Every UnderGrad *is-a* Student	Every Student *has-a* name
Every GradStudent *is-a* Student	Every GradStudent *has-a* degree
Every Car *is-a* Vehicle	Every Car *has-an* Engine
Every Whale *is-a* Mammal	Every Whale *has-a* Habitat
Every HashSet *is-a* Set	Every HashSet *has-a* size
Every ArrayList *is-a* List	Every ArrayList *has-an* array of items

Figure 6.4: Distinguishing between Inheritance (*is-a*) and Composition (*has-a*)

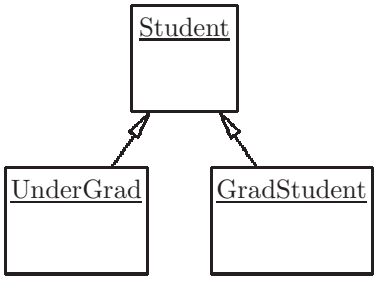

Figure 6.5: Class diagram showing relationship of Student classes

6.3.2 Factoring duplicated code and defining subclasses

To define a subclass in Java we use the keyword *extends*. This conveys the intent that the subclass consists of everything in the superclass, in addition to other fields and/or methods. The general format, when defining a subclass is:

```
public class subclass-name extends superclass-name
{ ...  fields, constructors, methods ...  }
```

Note that there can be only one superclass name after the keyword `extends` because a class can have only one superclass.

We can now redefine our Student classes using inheritance. `Student` will be the superclass; `GradStudent` and `Undergrad` will be the subclasses. Figure 6.5 shows the class diagram which will result.

In order to decide which fields and methods are to be placed in which classes, we return to the notion of *factoring* duplicated code. All fields and methods which are identical in `GradStudent` and `Undergrad` will be factored into the `Student` class, whereas all fields and methods which are different will be retained in `GradStudent` and `Undergrad`.

Figure 6.6 shows which fields of `GradStudent` and `Undergrad` are identical and therefore can be factored to the `Student` class. Fields or methods which occur in one class but not the other are indicated by a $\surd$. For fields and methods which occur in both classes, the figure shows whether they are the same or different.

Using the information in Figure 6.6 we can now define our three classes. All fields which are the same in the `GradStudent` and `Undergrad` classes will be fac-

Fields	UnderGrad	GradStudent
name	same	same
ssn	same	same
gpa	same	same
athlete	√	
degree		√

Methods	UnderGrad	GradStudent
getName()	same	same
getSSN()	same	same
getGPA()	same	same
getAthlete()	√	
setName()	same	same
calcGPA()	different	different
setAthlete()	√	
getDegree()		√

Figure 6.6: Fields and methods which are the same in `UnderGrad` and `GradStudent` can be factored to the `Student` super-class

tored to the `Student` class: `name, ssn, gpa`. All methods which are the same in both classes will be factored to the `Student` class: `getName(), getSSN(), getGPA(), setName()`. Note that the method `calcGPA(int,int)` occurs in both the `GradStudent` and `Undergrad` classes; however, it cannot be factored to the `Student` class because the method body (i.e. the implementation) is different in `GradStudent` and `Undergrad`.

```
/** A Student has a name, an ssn, and a gpa.
 *  This class serves as a superclass for various
 *  kinds of students.
 */
public class Student
{
private String name;
private String ssn;
private double gpa;

// accessor methods
/** @return The name of this Student
 */
public String getName()
{   return name;  }

/** @return The ssn of this Student
 */
```

```
public String getSSN()
{   return ssn;  }

/** @return The gpa of this Student
 */
public double getGPA()
{   return gpa;  }

// mutator methods
/** Change the name of this Student to the given name
 */
public void setName (String newName)
{   name = newName; }

/** Change the gpa of this Student to the given gpa
 */
public void setGPA (double newGPA)
{   if (newGPA >=0)          // check for valid value
       gpa = newGpa;
}
}
```

We now handle the two subclasses, `GradStudent` and `Undergrad`. In the `Undergrad` class we will exclude those fields and methods which have been factored to the `Student` class:

```
/** Every Undergrad is a Student
 *  Undergrad is a subclass of Student.
 *  Every Undergrad has an Athlete status (boolean).
 *  Undergrads are not registered for Thesis.
 */
public class Undergrad extends Student
{
private boolean athlete;

/** @return true only if this Undergrad is an athlete
 */
public boolean getAthlete()
{   return athlete;  }

/** Calculate the gpa of this Undergrad, if
 *  if the number of credits is positive
 */
public void calcGPA (int gradePoints, int credits)
```

```
{  if (credits > 0)
      setGPA (gradePoints / (double) credits);
}

/** Change the 'athlete' status of this Undergrad
 */
public void setAthlete(boolean ath)
{  athlete = ath;  }
}
```

In the `GradStudent` class we will exclude those fields and methods which have been factored to the `Student` class:

```
/** Every GradStudent is a Student.
 *  GradStudent is a subclass of Student.
 *  A GradStudent has a degree.
 *  All GradStudents are registered for 3-credit Thesis,
 *  which is not part of the GPA.
 */
public class GradStudent extends Student
{
private String degree;

/** @return This the degree of this GradStudent
 */
public String getDegree()
{   return degree;  }

/** Calculate the gpa of this GradStudent, if
 *  if the number of credits is more than 3.
 *  Assumes this GradStudent is registered for Thesis
 *  which is excluded from GPA.
 */
public void calcGPA (int gradePoints, int credits)
{  if (credits > 3)
      setGPA (gradePoints / (double) (credits-3));
}
}
```

6.3.2.1 Constructors

The reader may have noticed that constructors are absent from the code presented thus far. Constructors require careful attention when using inheritance. We will include constructors in all three of our classes, and each constructor will

be responsible for initializing the fields of its own class. In the Student class the constructor is the same as shown previously:

```
/** Initialize the fields of this Student
 *  @parm ssn Must be a valid ssn.
 *  gpa is initially 0.0
 */
public Student (String name, String ssn)
{   this.name = name;
    this.ssn = ssn;
    gpa = 0.0;
}
```

In the Undergrad class, keep in mind that every Undergrad has a name and an ssn. Therefore when an Undergrad is created, the creator will have to provide a name and an ssn. The constructor will then call the constructor in the superclass to initialize the appropriate fields. This is done with a call to *super*. A call to `super` in a constructor is a call to the constructor in the superclass, and the actual parameters in the call should correspond to the formal parameters in the superclass' constructor. This call to `super` must be the first statement in the subclass' constructor. After calling `super`, the Undergrad constructor will then initialize the athlete status to a default value, `false`. The constructor for Undergrad is:

```
/** Initialize the fields of this Undergrad.
 *  @param ssn Must be a valid ssn.
 *  Athlete status is initially false.
 */
public Undergrad (String name, String ssn)
{  super (name, ssn);       // call constructor in Student class
   athlete = false;
}
```

We use a similar strategy for the constructor in the GradStudent class. In this case the constructor will need another parameter for the GradStudent's degree:

```
/** Initialize the fields of this GradStudent.
 *  @parm ssn Must be a valid ssn.
 *  @param degree should include degree title and institution
 */
public GradStudent (String name, String ssn, String degree)
{  super (name, ssn);       // call constructor in Student class
   this.degree = degree;
}
```

6.3.3 Making use of inheritance

We close this section with a brief example showing how these classes can be used. Some other class, call it the *client*, could have a method containing the following code segment:

```
UnderGrad  younger = new GradStudent("jim", "322-23-3234");
GradStudent older = new GradStudent("sue", "240-44-2222");
younger.setAthlete(true);
older.setDegree ("B.A. from Penn State");
System.out.println ("Our new students are " +
                    younger.getName() + " and " +
                    older.getName());
```

Conceivably, we could also create a Student who is neither an UnderGrad nor a GradStudent, but just a plain Student:

```
Student stud = new Student("joe", "223-98-1782");
```

This student would be neither a GradStudent nor an UnderGrad, and this begs the question: does it make sense to have this kind of student in our program? We'll come back to this question later.

6.3.3.1 Assignment of references to variables

Once we have declared variables which store references to various kinds of students, we can instantiate those classes and assign the reference to the appropriate variable. A reference to an Undergrad can be assigned to a variable declared as `Undergrad`, and a reference to a GradStudent can be assigned to a variable declared as `GradStudent`, and a reference to a Student can be assigned to a variable declared as `Student`.

Moreover, since every `Undergrad` *is-a* `Student` and every `GradStudent` *is-a* `Student`, we can do the following:

```
Student stud1, stud2;
stud1 = new Undergrad("jim", "322-23-3234");
stud2 = new GradStudent("sue", "240-44-2222");
```

It would be a mistake to go the other way:

```
Undergrad younger = new Student ("jim", "322-23-3234");   // ERROR
```

The compiler will not allow this because it is *not* true that every `Student` is-an `Undergrad`.

The variables `stud1`, declared to be of type `Student` is now storing a reference to an `Undergrad`, and `stud2`, also declared to be of type `Student` is storing a reference to an `GradStudent`. Also note that these can change as the program executes:

```
stud1 = new GradStudent("joe", "323-87-0102");
```

So `stud1` is now storing a reference to a GradStudent.

	Variable			
	st		ug	
Code	static type	dynamic type	static type	dynamic type
Student st;	Student			
Undergrad ug;	Student		Undergrad	
ug = new Undergrad(”jim”,”22”)	Student		Undergrad	Undergrad
st = ug;	Student	UnderGrad	Undergrad	Undergrad
st = new GradStudent (”joe”,”33”);	Student	GradStudent	Undergrad	Undergrad

Figure 6.7: Static type versus dynamic type. Static type changes as the code executes.

The question to be addressed now is, What is the type of stud1 – UnderGrad or Student? We need to distinguish *two kinds of type*: *static type* and *dynamic type*.

Static type:

- Static type is the type of the variable shown in the declaration.
- Static type is determined when the program is compiled.
- Static type does not change as the program executes. It is in effect for the lifetime of the variable.

Dynamic type:

- The dynamic type of a variable is the type of the reference assigned to the variable.
- The dynamic type of a variable is determined when the program executes.
- The dynamic type of a variable can change as the program executes.

Figure 6.7 shows a code segment in which the the static and dynamic types of variables are shown as the code is executed. Static type is important because it will determine whether your program compiles without errors. Dynamic type is important, as we shall see in the next section, because it will determine which method is being called.

6.3.3.2 Assignment to subclass from superclass – casting

We now consider the case where we may wish to assign to a variable whose static type is a subclass, but we are assigning from a variable whose static type is the superclass; this can work, but only if the dynamic type of the variable being assigned is correct. Here is an example:

```
Student stud1 = new Undergrad("jim", "322-23-3234");
Undergrad younger;
```