Appendix A
Guidelines for Program Style

1. Use spacing and indenting in the conventional way

1.1 Indentations from the left margin should be in units of three or more blanks. It is easiest to set your tab stops at three to five characters each (0.25 to 0.4 inches apart) and use tabs to indent. At the left margin should be imports, the class heading, and the left and right braces that set off the body of the class. At the first level of indentation should be the words `private` and `public` for variable and method declarations, along with the left and right braces that set off the body of the methods.

```
import javax.swing.JOptionPane;
public class Whatever
{
     private String itsFirstName = "Darryl";

     public void getFirstName()
     {    return itsFirstName;
     }
}
```

1.2 All statements should be indented at least two levels further to the right than the class heading. Each of the keywords `if`, `else`, `do`, `for`, and `while` (for statements that contain statements) should have the subordinate statements indented one level further to the right than the keyword itself. If that subordinate material is enclosed in braces, the left brace should be directly below the first letter of the keyword and the right brace should be somewhat lower down and aligned with the left brace. There are three exceptions: (a) The multiway selection structure with the `else if` combination on one line together is usually more natural, as if it were one word `elsif`; (b) the `while` that goes with a `do` should be indented somewhat further than the `do` and have a left brace just before it, so it does not look like the beginning of a new statement; (c) the `catch` that goes with a `try` should be formatted the same way for the same reason.

1.3 No other indenting to the right should be used, except for at least a double indenting for a continuation of one statement onto a second or even third line. These continuations should come just before an operator (usually +) or just before a left parenthesis or just after a comma. Do not allow a line to run over 80 characters (thereby wrapping around when it is printed); press the ENTER key at a good point and indent the continuation.

1.4 A right brace should always be lined up with the corresponding left brace and one or more lines lower than it. These braces should be the first characters on their lines. Beginning programmers will be far less likely to have unmatched or mismatched braces if they follow this Allman style.

1.5 Operators (e.g., + <= ! ? != * &&) should have at least one space on either side of them. After all, they are equivalent to an entire word or phrase in English, and you put spaces around words, don't you?

1.6 If you want to use spacing to group a sequence of statements together, put a blank line between logical subunits of a method. Put a comment heading on such subgroups if you wish. But keep the statements indented as described above. And do not put blank lines between every pair of statements; that just wastes paper and does no good. In any case, the division between two methods should be stronger (more blank lines or a line of marks such as =======) than the divisions (blank lines) within methods.

2. Choose appropriate names of variables, methods, and classes

2.1 Names of final class variables or final instance variables should be entirely in capital letters. Use an underscore, as in NUMBER_OF_ITEMS, to set off separate English words in these identifiers. You may also use this form for local final variables if you wish.

2.2 All other names of variables and methods should begin with a lowercase letter. All names of classes should begin with a capital letter. All the other letters in a variable name, method name, or class name should be lowercase except for the first letter of a separate English word, which should be capitalized. Examples are variable `itsFirstName`, method `getFirstName()`, and class `ListOfFirstNames`.

2.3 Do not use a single letter for a variable name unless it is a variable that is local to a single method, and even then do so sparingly. Do not use a single letter for any method or class name. The only justification for a single letter is that it is only used in a short range of statements after its declaration.

2.4 Use names that clearly indicate the meaning of the variable, method, or class. It is generally not difficult to think of such a name except for some local variables which indicate a generic sort of value with no particular relation to other values. If you do not wish to type a longer name, even though you know it more clearly communicates the meaning, use temporarily a unique short name such as xzx and, after typing most of the uses of the name, do a find-and-replace to make it right.

2.5 Do not name a local variable or parameter the same as an instance variable or class variable. Only name two local variables the same if the range of statements over which one can be used does not overlap with the range for the other. An acceptable use is separate cases of `for (int k =...)`.

3. Choose good comments and declarations

3.1 Put a comment on almost every method, describing what the statements accomplish and what has to be true before the method is called (the precondition). An exception can be made for those for which the name makes the meaning perfectly obvious, particularly constructors and method names that are standard names such as `equals`, `compareTo`, `toString`, `getFirstName`, and `setLastName`.

3.2 Do not put a comment on much of anything else. Anyone who would profit from a comment such as "add 1 to k" on the statement `k++` should not be reading the coding anyway. Most of the time that you would use a comment to explain what a variable stands for, the variable name should be changed to incorporate that comment.

3.3 Use a named constant (final variable) for any value that is used in two or more methods of one class, except generally 0, 1, and "". The name serves as a comment describing what the value is for.

3.4 Declare the loop control variable of a for-statement inside its heading when feasible.

3.5 Declare a variable locally if possible. If not, declare it private instead of public, if feasible.

3.6 Assign a value to each class variable in its declaration. Assign a value to each instance variable in all constructors unless it is done in the declaration itself or you have a solid reason for not doing so. Initialize each local variable in its declaration or in the statement right after its declaration (the latter is sometimes necessary when you need an if-else statement to determine the initial value). Do not assign to a variable a value that will never be accessed by any statement under any circumstances.

4. Shorter is better, all other things being equal

4.1 If you can express the logic more compactly without losing clarity, do so. In particular, if you have any segment of coding that could be eliminated without any difference in effect, eliminate it. For instance, the following three phrases should almost always be rewritten:

```
else {}      // simply omit it
if (condition) {} else...  // rephrase as if (! condition)...
x = x;       // totally pointless
```

4.2 If you assign an expression to a variable just so you can execute only one statement using it shortly thereafter, you should usually just use the expression in the statement itself. For instance, `x = y + z;... callMethod(x)` should be written as `callMethod(y + z)` if (a) those are the only uses of the variable `x`, and (b) that method call is not inside a loop that the assignment of `x` is outside of, and (c) `y` and `z` do not change their values between the two statements.

4.3 However, if an application of the above principle would produce a fairly lengthy statement (say 1.5 lines of coding) when you would otherwise have two moderately short one-liners, your coding will probably be clearer with the one-time use of a variable.

4.4 When you have an if-else statement with the same statement as the last action in both subordinate parts, factor it out by putting it after the if-else statement. And when you have the same statement as the first action in both subordinate parts, you can often factor it out by putting it before the if-else statement.

4.5 If you have a calculation inside a loop that is known to produce the same result each time, factor it out: Make the calculation one time outside the loop, assign it to a local variable, and use that local variable instead.

4.6 If you need to process the values in a list of items one at a time, and the first one has to be processed differently from the rest, process that first one before starting the loop. And process the last value after the loop when the last value must be processed differently from the rest. For instance, avoid the following:

```
for (int k = 0;  k < someString.length();  k++)
   if (k == 0)...
```

4.7 Make a method out of a group of three or more statements that would otherwise have to be written three or more times at various places in one or more classes. It is often a good idea to make a method out of an even smaller group or out of a group that is used only in two different places.

4.8 Every method should fit on a single screen (maximum of thirty lines), unless perhaps (a) it is a quite lengthy switch statement or (b) it has no loops and not even much in the way of `if` statements (e.g., lots of output statements or lots of adding components to a container). Even better, try to keep a method's coding below ten statements.

4.9 Use the boolean values `true` and `false` only as actual parameters or to assign to variables. Do not use them in a conditional expression; `if (b == true)` is better expressed as `if (b)`, and `if (b == false)` is better expressed as `if ( ! b)`.

5. More advanced points

5.1 **Generalize**: Write a method to perform a more general task if you can do so without significantly slowing down execution of the particular task you need done. For instance, do not declare a parameter as String if Comparable will do just as well, and do not declare it as Comparable if Object will do just as well. Also, do not use a specific numeric value in a method if (a) it works as well to pass in the numeric value as a parameter, and (b) one could conceivably use the method with other numeric values.

However, do not write a method whose body is basically `if(c) part1 else part2` if `part1` is only executed when the method is called from one place and `part2` is only executed when the method is called from another place. It is far better to have two separate methods, each called from a different place. Example for Chapter Nine: Do not have one ActionListener object listen to several different buttons unless the action to be taken is nearly the same coding for all buttons.

5.2 **Explain**: Give a prompting message when asking for input from the user at the keyboard.

5.3 **Be robust**: If an unexpected situation could throw a RuntimeException, guard the dangerous statement with an appropriate test. If it is too complex to guard against, be prepared to catch the Exception and recover enough to go on. If you cannot usefully continue, at least print an appropriate message explaining the problem and save what you can of the work so far.

5.4 **Do one thing well**: If a method returns a value, then it is highly preferable that it not cause a change in any object (i.e., it should be what this book calls a "query method"). Two reasonable exceptions are (a) a method to get input from the user may change the input object (e.g., a file pointer), and (b) a method that takes an action may return a value that describes what action it took, such as whether its attempt to delete was successful or how many objects it modified in the process of doing what it does.

5.5 **Additional style principles used in this book, though not universally accepted**:
- Do not use `continue`. Do not use `break` except within a `switch` statement.
- Do not use `protected` or default visibility.
- Do not declare two variables on the same line unless they are very intimately related, such as `x` and `y` for coordinates of a point.
- Do not put other methods in a class with a `main` method.
- List field variables in the order (a) public class variables, (b) private class variables, (c) instance variables.
- Use "the" as a prefix of non-final class variables and "its" as a prefix of instance variables.

Appendix B Indexed Glossary of Terms

All terms that are used in more than a single section of Chapters One through Thirteen are defined in this glossary. Numbers in brackets [ ] are pages where the word or phrase is defined or illustrated. We omit Chapters Fourteen through Eighteen because those are all about data structures.

abstract class [11-4,11-7]: a class where one or more methods are marked "abstract" and have no body. You cannot call a constructor for an abstract class. An abstract class exists in order to be subclassed.
action method [2-21]: a method that changes the state of a variable but does not return a value. By contrast, a query method returns a value but does not change the state of any object or non-local variable. Try to avoid methods that do both.
actual parameter [3-14,3-16]: an expression supplied in the parentheses of a method call. By contrast, a formal parameter is a variable declared in the parentheses of a method heading. When the method executes, each formal parameter in the heading is initialized to the value of the corresponding actual parameter.
algorithm [3-21]: a step-by-step procedure for solving a problem in a finite number of clearly-stated steps.
alias [3-16]: a variable that refers to the same object that another variable refers to. A formal object parameter is an alias for the corresponding actual parameter.
All-A-are-B looping action [5-20,6-16,8-20]: logic to look at what is usually many values and see whether every one of those present has a specified property.
analysis [1-26,2-28,3-19,6-29]: the process of finding out the details of what a client wants a particular piece of software to do, so there is no question about what the software should or should not do.
anonymous class [10-18]: a class without a name by which you can refer to it.
anonymous object [10-8]: an object created by a method without using a variable to store a reference to that object.
appletviewer [8-3]: a utility program free of charge from Sun Microsystems, Inc., which inspects a web page and displays only the applets on that web page.
application program [1-4]: a class that contains a method with the heading `public static void main (String[] args)`. An application program named `SomeClass` can then have that main method executed by the runtime system with a command that begins `java SomeClass`. The rest of the words on the command line are passed to the `args` parameter.
argument [2-13,3-14]: a value in the parentheses of a method call; it is also known as an actual parameter. That value is passed to a (formal) parameter in the method heading.
array [7-9,7-23]: a reference to a sequence of variables that are all of the same type, e.g., to 100 int variables or to 5000 String variables. If `x` is an array variable, then the number of variables referred to is `x.length`, and you may refer to a single such variable by `x[k]` where `k` is an int expression with a value in the range `0..x.length-1`.
ascending order [7-23,7-24,13-1]: an arrangement whereby each value is greater than or equal to the value just before it in the sequence.
assembler language [1-28,12-30]: a simple, almost one-for-one, correspondence between English-like code words and a computer's numeric machine code.
assignment [2-22]: giving a value to a variable, thereby replacing its previous value if any.
backslash character [6-10]: the character used to indicate that the character immediately following it does not have its usual meaning. For instance, '\n' means that the 'n' denotes a newline, not a lowercase 'n'.
big-oh behavior [13-8,13-21,13-22]: the function of N (the size of the input) for which the execution time for N values is asymtotically proportional to that function.
binary operator [Interlude-2]: an operator with two operands.

binary search [13-9,13-10]: finding a value in a sequence by looking at the middle value to decide whether to next look in the first half or in the second half.

binding [11-6]: The compiler binds a method call to a particular method definition when it can, namely, for a class method or for a final instance method. This is early binding, which reduces execution time compared with late binding (done at runtime).

block [2-16]: a sequence of statements with a matched pair of braces `{` and `}` marking the beginning and end of the sequence.

body of a method declaration [1-14,1-17]: the part of a method declaration that starts with the first left brace `{` and ends with the right brace `}` that corresponds to it, including the contents of those matched braces.

boolean method [2-19] a method that returns a value of `true` or `false`.

boolean operator [2-24]: a symbol that combines one or two true-false expressions to obtain a new boolean expression. `&&` and `||` apply to two boolean expressions; `!` applies to just one.

bounding box of a figure [8-6]: the smallest rectangle with vertical and horizontal sides that completely encloses the figure. Its sides are at the smallest and largest x-values of points in the figure, and its top and bottom are at the smallest and largest y-values of points in the figure.

braces [1-3,2-31]: the wiggly grouping symbols `{` and `}`. Don't call parentheses `( )` braces. This book uses the Allman style in the placement of braces in coding; many people use the Kernighan & Ritchie style (right brace at the end of a line of code).

brackets [7-9]: the straight-sided grouping symbols `[` and `]` used in Java to select a component of an array. Don't call parentheses `( )` brackets.

buffering [12-5,12-24]: An input file is buffered if it gets one big chunk of data at a time from the hard disk and saves it in RAM until you ask for it, typically in small chunks. An output file is buffered if it saves up in RAM the small chunks of data you give it until it has a big chunk of data that it can write to the hard disk or screen all at once. If you have a reason to move the saved-up output to its ultimate destination before the buffering algorithm chooses to do so, you flush the buffer.

bug [2-27,8-18]: a failure of the software to have an effect its specs say it should have, or any effect that the software has that its specs do not say it should have. Thus, (a) a program that has no specs has no bugs, (b) a program that prints an explanatory message has a bug if the specs do not say it should print anything, (c) a program that crashes does not have a bug if the specs say it is supposed to crash at that time.

busywait [8-15,10-25]: execution of statements that accomplish nothing except to make time pass. Used in animation by programmers who do not know how to put their thread to sleep (e.g., by using a Timer).

byte [6-20,11-11,12-27]: one byte of storage requires 8 bits (on/off values or high/low voltages), so there are 256 different possible values for one byte.

cast [6-2,6-12,6-14]: the placement of a type description in parentheses in front of an expression. `(int) x` tells the compiler to use the value in `x` in modified form, without a decimal point. `(Sub) x` tells the compiler to treat `x` as though it refers to a member of class Sub. The cast does not change the value stored in `x`.

chaining [5-10]: If a method call returns an object, it can be used as the executor of another method call. This is chaining. An example, is `sam.getCD().getName()`, wherein the first method call returns a CD object, and the second method call asks that CD object for its name.

checked exception [9-10]: a non-RuntimeException. Any method that contains a statement that can throw a checked Exception must handle the potential Exception in one of two ways: Put the statement within a try/catch statement that can catch it, or put a throws clause `throws XXException` at the end of the method heading.

class [1-3,3-7]: the primary organizing unit in Java. An object class is intended as a "factory" that can be used to create objects. A utilities class provides class methods that can be called without an executor, i.e., an object to which the message is sent.

class definition [1-14]: the compilable unit that begins with the word `class` and any modifier words before it, and ends with the right brace that matches the first left brace after the word `class`.

class diagram [2-26,2-27,3-8,3-31]: one or more rectangles connected by arrows using UML notation. The rectangles are divided into three parts. The top part has the class name and the bottom part lists any of its method calls that you wish to mention. A dependency of the form `X uses Y` is indicated in UML by an arrow with a dotted line. A generalization of the form `X is a kind of Y` is indicated in UML by an arrow with a solid line and a big triangular head.

class method [2-13,2-14,5-2,6-15]: a method that does not have an executor for the method call. It is signaled by the word `static` in the method heading. You may call a class method with the name of its class in place of an executor.

class pattern for reusable software [6-2]: The compilable file contains a public object class and also a non-public class. The latter has the `static void main` method, which just creates one or more objects to do the job and, at the end of the program, executes `System.exit(0)` if graphics were used. `main` is used to test the object class that will presumably be used in a larger piece of software.

class variable [5-4,5-5]: a variable declared outside of every method and with the word `static` in its declaration. It exists independently of any object of that class.

client-server relation [2-27]: when a method in one class C uses an object from another class S, then C is the client in the relationship and S is the server.

coding [1-26,2-20]: the translation of a design into a specific programming language.

command [1-3]: an instruction or message sent to an object or class.

command-line argument [2-5,7-32,12-16]: a String value on the command line after `java ClassName`. The command-line arguments are passed in to the `String[] args` parameter of the main method. `args.length` tells how many values are passed in.

comment [1-3,3-23,6-11]: material in a program intended to explain the operation of the program to a human reader. The compiler ignores comments, which are (a) any material after // in a program, down to the end of its line, or (b) anything between /* and the next */. You cannot have spaces between the two characters.

compareTo method [6-10,6-11,6-13,11-5]: the one method required by the Comparable interface. The Sun library String and Double classes have a `compareTo` method. It returns an int value: `x.compareTo(y) < 0` means that `x` comes before `y` and `x.compareTo(y) > 0` means that `x` comes after `y`. The exact value of the int it returns has no meaning.

compiler [1-4,1-5,1-6,1-17,1-18,1-28]: a program that translates an entire file from one form to another form. A class X is stored in a plain text file typically named `X.java`, readable by humans. The compiler translates this file into a form the runtime system can use, which is stored in a file named `X.class`.

component [7-11]: one of the many variables that make up an array. One component is referenced by the array name followed by an int-valued index in brackets.

composition [6-16]: A class of objects is formed by composition if its only or its primary instance variable is an object of another class.

concatenation [Interlude-3,6-9]: the combination of two strings of characters, one attached to the end of the other.

condition [2-9,2-10,2-15,2-21,2-25]: another word for a boolean expression, which is an expression that is either true or false.

conditional operator [6-7]: the operator ?: in an expression such as `c ? t : f`; it evaluates and returns just one of `t` and `f: t` if `c` is true, `f` if `c` is false. The two parts after the question mark have to be of the same type (int or boolean or Worker or whatever).

constructor [4-8,4-9,4-10,4-12,4-17,4-20,4-21,5-2]: a method whose name is the same as the class it is in. Calling a constructor produces a new object value of that class. Within the constructor you may use `this` to refer to the object created, although a constructor is not an instance method.

continuation condition [3-2]: A `while` statement states a continuation condition followed by the subordinate statements. The continuation condition must be true in order for the subordinate statements to be executed. A `for` statement and a `do-while` statement also have continuation conditions.

conversion [6-21]: Assigning a char value to an int variable or an int value to a double variable causes a widening conversion, a.k.a. promotion; the compiler does this automatically. The opposite is a narrowing conversion; it requires a cast.

count-cases looping action [6-3,8-19]: a pattern in which you initialize a counter before beginning a loop, then increment it once each time through the loop. When the loop finishes, the counter tells how many iterations were done.

count-controlled loop pattern [6-18,8-18]: a loop that executes a number of times given by an expression whose value is known before the loop begins.

cowboy hats [6-31] a metaphor for a class cast.

crash-guard [6-21,6-22,9-3]: `1.` a condition that must evaluate as true in order for another expression to be evaluated, when that second expression would throw an Exception if the crash-guard condition were false, as in `(x != 0 && y > 2 / x)`; 2. a condition that must evaluate as false in order for another expression to be evaluated, when the other expression would throw an Exception if the crash-guard were true, as in `(x == 0 || y <= 2 / x)`.

CRC cards [13-31]: index cards that categorize classes by their responsibilities and collaborations. CRC cards are used in object design and logic design.

dangling else problem [2-30]: a statement of the form `if (c1) if (c2) st1 else st2` where the programmer meant the `else` to go with the first `if`, though the compiler always matches it with the second `if`.

data flow diagram [13-28]: a diagram of the flow of data among internal processes, data stores, and environmental entities.

debugging [4-27,4-34]: finding the logic errors in some coding.

declaration [1-2,1-31,5-7]: `SomeClass sam` creates a variable of type SomeClass and declares `sam` as the name of that variable. The phrase `SomeClass sam` is a variable declaration. For instance, `String sue` declares a variable for storing a String object. The heading of a method declares what its name is, what must be supplied in the parentheses of the call, and what is returned by the call.

decrement [Interlude-1]: to subtract 1 from the value of a variable, thereby changing it.

default [1-9,3-11,4-8,4-12,10-11]: a language element supplied by the compiler because it is optional in a particular position in the coding and the programmer does not supply it. Examples: The default constructor when you do not supply one is `public SomeClass(){super();}`. The default executor of a call of an instance method is `this`, which refers to the executor of the method the call is inside of. The default extension of a class is `extends Object`.

definition [1-8,1-14,1-32,2-7]: The body of a method is the definition of what happens when you call it. By contrast, a declaration tells how to call it and how to use what it returns to you.

dependency [2-26]: When a class uses methods defined in another class, that is a dependency relationship.

descending order [13-1]: an arrangement whereby each value is less than or equal to the value just before it in the sequence.

design [q.v.]: see "object design" and "logic design".

driver method [4-15]: a main method whose only purpose is to test one or more method definitions.

efficiency [13-22]: a measure of the execution time, storage space, or programmer effort required to execute an algorithm.

elementary sorting method [13-8]: a sorting method whose big-oh behavior is N-squared. Insertion sort, selection sort, and the infamous bubble sort are elementary.

empty string [6-10]: the String value whose length is zero. It can be written as `""`.

encapsulation [4-6,5-4,6-8]: preventing outside classes from changing field variables directly (i.e., without calling a method in the class). The standard way to encapsulate is to declare field variables as `private`.

equals method [3-4,4-17,6-9,6-13,11-9]: a method of the Object class that tells whether two objects are exactly the same object. It is usually overridden in a subclass to tell whether two objects have the same essential contents, even when the objects themselves are different objects. The `equals` method is never used for numbers.

event-driven programming [10-3,10-10]: attaching a listener object to a component and arranging to have a change in the component send an action message to the listener object.

exception [6-8,6-9,7-11,7-15,9-3,9-4,9-7,9-9,9-10,9-11,9-13]: an object that a method throws to notify the runtime system of a problem that crashes the program (unless you have the appropriate try/catch statement). Examples are a NumberFormatException (thrown by trying to parse a string of characters as a numeral when it is not), an ArithmeticException (caused by trying to divide by zero), an IndexOutOfBoundsException (thrown by e.g. `si.charAt(k)` if it is false that `0 <= k < si.length()`), and a NullPointerException (thrown by e.g. `si.charAt(k)` if `si` has the value `null`).

executor [1-9,2-21,10-11]: When a method call has a variable before the dot (period), as in `sam.moveOn()`, that variable refers to the executor of that call. If the statements within some method M include a method call whose executor is not stated, then the executor of the method called is M's executor.

exit method [4-2]: Terminates the program. Required when GUI components are used.

field variable [5-4]: a variable that is declared outside of every method in a class. It may be an instance variable or class variable. If the declaration of a field variable assigns it a value, that takes effect for an object's instance variable when the constructor is called, for a class variable when the program begins. The runtime system supplies a default value if you do not, zero or `null` or `false`, as appropriate.

final [5-7,5-8,5-13,11-7]: A final entity is one that cannot be changed once it is given a value. Putting `final` on a variable means it cannot be changed after it is first assigned a value. Putting `final` on a method means it cannot be overridden. Putting `final` on a class means it cannot be subclassed.

formal parameter [3-14,3-16]: a variable declared in the parentheses of a method heading. By contrast, an actual parameter is an expression supplied in the parentheses of a method call. When the method executes, each formal parameter in the heading is initialized to the value of the corresponding actual parameter.

functor, function object [10-9]: an object whose only purpose is to supply one or more methods to another method for a polymorphic call. It does not have any instance variables except perhaps those that are parameters of the function that the method provides. A functor is passed to a formal parameter whose type is an interface.

garbage collection [3-11]: recycling of an object that your program has previously created but currently has no variable that refers to it. That is, reusing space in RAM that is no longer needed. The Java runtime system does this automatically.

generalization [2-26]: When one class inherits from (extends) another class, it is a generalization of that other class.

graphical user interface (GUI) [10-1]: interaction with the user through graphical components such as buttons, textfields, sliders, and textareas.

hardware [1-17]: the physical components of the computer (chip, RAM, disk, monitor, etc.). By contrast, the programs that you run and all the classes they use are software.

heading of a method declaration [1-3,1-6,1-14]: the part of a method declaration up to but not including the first left brace `{`. In particular, it includes the parentheses after the method name.

heap sort [13-24,13-25,13-26,13-27]: sorting a partially-filled array by (a) arranging it so that it is a heap (the element at each index k is larger than or equal to the two elements at indexes 2*k+1 and 2*k+2, insofar as those indexes are within range), then (b) repeatedly removing the element at index 0 (which will be the smallest of all) and then shifting elements to make the (slightly-smaller) array a heap again.

hierarchy [3-15,10-7]: a relation among things where (a) one thing is the "top thing", (b) every other thing has a thing which is immediately higher than it is, and (c) there is at most one way to go from one thing to another in such a way that each step along the way is from a thing to its immediately higher thing.

HTML [8-2]: HyperText Markup Language, a code used to indicate how things should be displayed on a web page.

identifier [1-14]: any name of a variable, a method, or a class.
immutable [5-8,6-9,11-22]: what we call a class for which the values of instance variables of an object cannot changed after that object has been constructed.
implementation [10-3]: 1. the translation of a design or plan into actual Java coding (or another language). 2. a class extends a class but implements an interface.
increment [Interlude-1]: to add 1 to the value of a variable, thereby changing it.
indenting [3-3,4-5]: indicating which statements are subordinate to which other statements, or which statements are inside which method, or which declarations are inside which class.
independent method [5-3,7-23,12-3]: a class method that could be placed in any other class because it does not refer to any instance variables or class variables of its class. All of the information it uses to do its job is in its parameters.
indirect recursion [13-19]: a method M calls another method which calls M.
infinite loop [9-4]: a loop, conventionally indicated by `for(;;)`, that executes until a return statement executes or an exception is thrown.
inherit [1-10,4-10]: to have available in a class all of the public methods defined in a superclass of the class. The phrase `extends SomeClass` in the heading of X makes SomeClass the superclass of X. Each instance of the subclass X can then use the public methods of SomeClass as if they were defined in X, unless X redefines them with the same name and parameter types.
initializer list [7-12]: a list that follows the assignment symbol `=` in an expression of the form `Type[ ] x = {...}`. The comma-separated values listed inside those braces create the array `x` of the corresponding length and initialize it to have the values listed.
initializing [4-12]: assigning a value to a variable for the first time. An instance variable should be initialized in its declaration if its initial value is the same for all instances, otherwise it should be initialized in every constructor's coding. A class variable should always be initialized in its declaration.
inner class [10-11]: a class that is defined inside of another class and has the word `static` in its heading. Instantiation of an object of the inner class requires an object of the outer class that is then permanently associated with the inner object.
insertion sort [7-26,13-5,13-11,13-12,13-23]: sorting a list by (a) separating one element from the rest; (b) sorting the rest of the list; and then (c) inserting the one element where it goes in the sorted group.
instance [1-9]: an object created by execution of a phrase of the form `sam = new SomeClass()` or its equivalent. The phrase puts a reference to that object in the variable `sam`. The newly-created object belongs to SomeClass.
instance method [1-9,1-14,2-13,5-6]: a method that requires an executor for the method call. It is signaled by the absence of the word `static` in the method heading.
instance variable [4-6,4-12]: a variable declared outside of every method and without the use of the word `static`. Each object of that class has its own value for that variable.
interface [6-11,11-4,11-5]: a compilable unit; the heading `public interface X` means `X` cannot contain anything but non-final instance method headings and final class variables. A non-abstract class with the heading `class Y implements X` must define all methods in that interface.
internal invariant of a class [7-21,7-22,11-14,11-18,11-20]: a condition on the private structure of instances of a class that is always true when any method of the class is called and always true when any method of the class finishes execution. The internal invariant may be temporarily false during execution of such a method.
interpreter [1-28]: a program that translates one line of source code (written by the programmer) to object code (executable by the chip) at a time. It executes that object code before going on to translate the next line.
iteration [3-4]: one execution of the subordinate statements of a `while` or `for` or `do-while` statement.
iterative development [3-23,8-9,8-10,8-17,9-23]: the development of a larger software project in stages, where the version created at the end of each stage comes closer to doing what the final project should do.

iterator-controlled loop pattern [8-18]: a loop that keeps track of its position in a sequence of values and executes once at each position, except that it may terminate early before reaching the end of the sequence if it accomplishes its purpose. The position in the sequence advances by one step on each iteration.

javadoc [2-14,3-28] a formatting tool for automatically documenting source code.

keyword [1-15]: a word (made up of letters) whose meaning is specified by the language. All the words in a program are keywords except names of variables, methods, and classes. Java keywords are always entirely in lowercase letters, even `instanceof`. You cannot declare a keyword to have another meaning.

lexicographical ordering [6-10]: the ordering of characters by their Unicode values, and of Strings of characters by the Unicode values of their characters. It is alphabetical ordering if only between lowercase letters or only between uppercase letters. It is different from numerical ordering, e.g., 2 is before 13 but "2" is after "13".

library class [6-5]: a class intended for use in several different software situations, some of which can be expected to arise in the future.

linking [1-17,1-32]: When the runtime system uses a class definition, it links in all other class definitions needed by that class.

listener object [10-8]: an object that is attached to another object, that listens for a particular kind of change in that other object, and that executes a specific method when the change occurs.

literal [4-3]: a sequence of characters representing a value whose meaning is known. They include String literals (in quotes), numeric literals (made up primarily of digits), char literals (in apostrophes), and `true`, `false`, and `null`.

local variable [3-16,5-7,7-11,7-17,7-19]: a variable declared within the body of a method. These local variables have no connection with variables outside of that method, and they have no initial value. You can only use a local variable after the point where it is declared and inside whatever braces containing the declaration.

logarithm base 2 of a number [13-10]: the number of times you have to repeatedly double `x`, starting from `x = 1`, to make `x` greater than or equal to the number. This value (rounded up from the "true" log) is denoted log2 in this book. So `log2(x)` is 5 for any value of `x` in the range from 17 to 32 inclusive.

logic design [2-20,2-29,3-19,6-29]: a description entirely in English (or whatever natural language you are most comfortable in) of the process by which a method solves a problem.

loop [3-4,3-6]: 1. the repeated execution of a group of statements. 2. a group of statements that is repeated along with the condition that decides whether to repeat.

loop-control variable [3-26,Interlude-4]: the only variable that (a) appears in the continuation condition for a looping statement and (b) is modified during execution of the loop.

loop invariant [13-6,13-7,13-10,13-14,13-18]: a condition that is always true at the time the continuation condition of a while-statement or for-statement is evaluated. It should be (a) trivially true the first time it is evaluated and (b) incrementally true in that its truth at the beginning of one iteration implies its true at the end of that iteration. It should also be a condition that provides a clear proof that the loop produces the result it is supposed to produce.

machine code [1-28]: the sequence of numeric codes that a computer directly understands.

main method [1-3,1-14,6-33]: a method with the heading `public static void main (String[] args)`. It can be executed by the runtime system with a command of the form `java SomeClass` where `SomeClass` is the class containing the main method. Such a class is called an application program. This book rarely puts any other method in the same class with a main method, though it is legal to do so.

max-heap [13-24]: a data structure in which each element is at least as large as its children (of which there are 0, 1, or 2).

member of a class [6-8]: a variable, method, or class that is declared inside the class but not declared inside any braces within that class.

merge sort [12-7,13-17,13-18,13-19,13-20,13-21,13-23]: sorting a list by (a) dividing it in two halves of equal size; (b) sorting each half separately; and then (c) inserting the elements of one half where they go in the other half.

method [1-6,2-9]: a sequence of statements with a heading by which it can be called (executed) by some statement.

method body [1-14]: the part of the method definition starting with the first left brace and ending with the matching right brace.

method call [1-9,2-9]: a specific request that the runtime system carry out the process specified by the method named.

method definition [1-8]: the part of a class that tells how a method is called and what statements it will execute when it is called.

method heading [1-14]: the part of a method definition up to but not including the first left brace within the definition.

min-heap [13-24]: a data structure in which each element is at least as small as its children (of which there are 0, 1, or 2).

modal input [6-21]: The software executes a command that forces the user to respond before the software will do anything else.

model/view/controller pattern [6-32,6-33,6-34,10-16]: a separation of software into three areas: the controller receives the input, the view displays the output, and the model stores the data.

modular programming [7-19]: the creation of software for one purpose as a number of modules (classes) designed so that, when you want to use the software for a different purpose, you just pull out a few of the inappropriate modules and slot in others.

multiway selection format [2-31,6-11]: formatting that puts an `if` directly after an `else` instead of indented on the next physical line. The purpose is to make it clear that the choice is from among three or more alternatives.

natural constructor [4-10]: a constructor whose primary purpose is to initialize all or most instance variables to its parameter values.

newline character [4-4]: the character that causes the output to appear further down on the screen. The pair `\n` indicates a newline character within a string literal.

nested class [6-8]: a class that is declared inside of another class X. If it is declared as `static`, the only difference from declaring it outside X is its visibility: (a) The nested class can access the private variables and methods of X; (b) No class outside of X can access the name of the nested class if that nested class is declared as private, even if its variables and methods are public. If the nested class is not private, outside classes refer to it as `X.ThatNestedClassName`. If the nested class is declared without the use of the word `static`, it is an inner class.

null [4-5,4-12,4-19,5-5,6-8]: the value assigned to a variable to indicate that the variable does not reference any object at all. If you then execute a command with that variable as the executor, you cause a NullPointerException to be thrown.

object [1-2,2-3]: a set of values stored in RAM that describes a conceptual entity. It contains the information about that entity that is relevant to the software being developed.

object class [1-14,2-7]: a class that has either instance methods or instance variables.

object code [1-18]: the file that a compiler produces from the source code. The object code is not in human-readable form but the source code is.

object design [1-11,1-24,3-19,4-19,4-28,6-30,6-31,9-16]: determining the kinds of objects that a piece of software needs to get its job done, especially the capabilities (methods) that those objects must have. The key technique is to find objects that have the capabilities the software needs; if you know of none, then look for objects to which needed capabilities can be added (typically by subclassing existing classes); if that is still not enough, design suitable objects from scratch.

object diagram [2-27,6-31,7-13] a picture with boxes and arrows to show the relationships among some objects.

operand of an operator [2-24,2-32]: one of the expressions that the operator combines in order to calculate a new value. Most operators in Java have two operands.

operator [2-24,Interlude-2]: a symbol that, when applied to one or more expressions according to the rules of Java, produces a value of a particular type. For instance, + is an operator that applies to two int values to produce an int value, and `!` is an operator that applies to one boolean value to produce a boolean value.

overloading a method name [4-20]: having two methods of the same name in the same class. The two must have different numbers and/or types of parameters.

overriding a method definition [4-20]: having an instance method in a class X with the same signature (including number and types of parameters) as an instance method in a superclass of X. Suppose the signature is `doStuff(int)` and `sam` refers to an object of class X. Then `sam.doStuff(5)` calls the subclass method and `sam.super.doStuff(5)` calls the superclass method.

package [4-3]: a way of organizing classes. Each class in Java is categorized by the package it is in. If you put the phrase `package X;` at the top of a file, that puts the classes in that file into package X. If your file does not have this package directive, its classes are in the default package associated with that folder.

parallel arrays [9-21]: two or more arrays x, y, z, ... such that, for a particular index k, x[k], y[k], z[k],... are instance variables of a single conceptual object. Normally it is better to declare an object class with those instance variables so you can just have a single array of such objects.

parameter [2-13,3-14]: When you have a method call, the values in the parentheses after the method name are the actual parameters of the call. They supply additional information so the method can do its job. Those actual parameter values are passed to the corresponding formal parameters named in the method heading.

parsing [4-19,6-18]: analyzing a string of characters (usually received as input) into its component parts to find out what meaning was intended.

partially-filled array [7-18,11-26]: an array that has useable values only in part of the array. In this book, we normally put the useable values at components indexed `0...itsSize-1`, where `itsSize` is no more than the length of the array.

pixel [1-1,8-4]: the smallest unit on a monitor that can be drawn; a graphical dot.

polymorphism [4-22,7-4,7-5,8-25,10-4]: what occurs when a particular method call in a program could invoke any of two or more different method definitions at different execution times, depending on the class of the executor of that method call. The two or more method definitions must necessarily have the same signature.

postcondition [7-20]: a statement of what will change as a result of calling a method, and/or what will be returned, assuming the precondition for that method is met.

precedence of operators [2-25,4-23,4-24]: the rules that determine which of two operators is evaluated first, when parentheses do not make it clear.

precondition [2-17,2-27,6-22]: what has to be true when a method begins execution, in order that the method produce the expected result. The precondition has been verified by the calling method, so the method called is not to test for it.

primitive type [6-20]: a value that does not refer to an object. The four primitive integer types are long (8 bytes), int (4 bytes), short (2 bytes), and byte (1 byte). One byte of storage is 8 bits, so there are 256 different possible values for one byte. The two primitive decimal number types are double (8 bytes, about 15 digits) and float (4 bytes, about 7 digits). The other two primitive types are boolean and char.

private [3-9,4-6,5-3,5-9,7-15,12-26]: A method declared as `private` can only be called from within the class where it is defined. A variable declared as `private` can only be mentioned within the class where it is defined. Note that it is private to the class, not private to the object; an instance method can refer to a private instance variable of a non-executor object of that class. By contrast, a method/variable declared as `public` can be called/used from any class.

promotion [6-2,6-21]: what occurs when e.g. you assign an int value to a double variable or combine an int value with a double value using an operator. The runtime system will promote the int value to a number with a decimal point. By contrast, assigning a double value to an int variable would be a "demotion", which requires that you put an `(int)` cast in front of it to convert it to an int value.

prompt [4-4,6-6,6-32]: a string of characters printed by the software immediately before user input, to tell the user what kind of input is expected.
prototype [1-23,4-1,5-25]: a piece of software that does only a small part of what the final product is intended to do. Typically you have many methods simply print a message that they were called, rather than doing what they are supposed to do.
public [1-6,1-9,1-10,3-9,4-6,5-3,12-26]: A method declared as `public` can be called from any class. By contrast, a method declared as `private` can only be called from within the class where it is defined.
query method [2-21]: a method that returns a value but does not change the state of any object. By contrast, an action method changes the state of a variable but does not return a value. Try to avoid methods that do both.
queue [6-1,7-34,7-35]: a list of values for which the primary operations are adding a new value and removing an old value; the latter operation always removes the value that has been on the list for the longest period of time. So it is often called a FIFO (first-in-first-out) data structure.
quick sort [13-11,13-13,13-14,13-15,13-22]: sorting a list by (a) dividing it into two groups, those less than a chosen value and those greater than the chosen value; (b) sorting each of the two groups separately; and then (c) placing the larger group after the smaller group.
recursive call [1-30,5-27,5-28,13-12]: a method call that occurs within the body of the method being called. If the call is made only when a particular parameter or attribute of the executor is a smaller positive integer than it was on the current call, then you will not have an infinite sequence of recursive calls.
return type [3-18,Interlude-1]: the type of value (e.g., boolean, String, int) returned by a call of a method. It goes just before the method name in the method heading. A method that returns no value must have `void` just before the method name in the heading.
reusable software [1-25,4-16]: software developed for one application that can easily be used in other applications developed later.
robust program [2-9,7-6]: a program that handles unexpected or unusual situations in a reasonable manner without crashing.
runtime exception [9-3,9-5,9-10]: an object from a subclass of RuntimeException. It can be thrown by a method without being acknowledged by a throws clause in the method heading. It is normally thrown only when a bug in the software is detected, although NumberFormatException is also a runtime exception.
runtime system [1-5,11-6]: the process that carries out the instructions in a piece of software.
scientific notation [6-1,6-2,11-11]: a numeric form having a number from 1.0 up to but not including 10.0, followed by 'E' and then an int value indicating the power of 10 to multiply by. For instance, -2.75E6 represents -2750000 and 3.0E-6 represents 0.000003.
scope of a variable [5-7]: the range of coding where the variable can be used without being directly preceded by a dot and an object or class reference.
selection sort [13-2,13-12]: sorting a list by (a) finding the smallest element; (b) placing it at the beginning of the list; and then (c) sorting the rest of the list.
sending a message to an object [1-19]: having an object do what a named message says to do, where the object is the executor of the method call.
sentinel-controlled loop pattern [6-2,8-18]: a loop that processes each value in a sequence of values until it reaches a special value, different from normal values, signaling the end of the sequence. That sentinel value is not to be processed.
sequential search [6-18,13-10]: Finding a target value in a list of values by going through the list from one end towards the other end until you find it.
shadow [4-11]: a local variable of the same name as a field variable, or a field variable of the same name as a field variable of its superclass. Avoid the shadows.
shell sort [13-23]: See Major Programming Project 13.6 for a description.

short-circuit evaluation [2-24,2-32]: If the first operand `x` is true in `x || y`, or is false in `x && y`, then the second operand `y` is not evaluated, and the result is the value of `x`. It "short-circuits" the evaluation of `y`.
signature of a method [4-19,4-20]: the name of the method followed by parentheses containing the types of its parameters. Different methods can have the same name if they have a different parameter pattern (what is inside the parentheses) or are in different classes. This is overloading of method names if they are in the same class.
simulation [1-25,2-4]: using computer software to see what would happen in a real situation.
software [1-17]: the programs that you run and all the classes they use. By contrast, the physical components of the computer (chip, RAM, disk, monitor, etc.) are hardware.
Some-A-are-B looping action [5-20,7-22,8-19]: logic to look at what is usually many values and see whether at least one of them has a specified property.
source code [1-18]: A file containing a human-readable compilable Java class, which must be translated to object code before it can be executed by the runtime system.
special assignment operator [6-4]: An operator such as `+=` or `/=` that changes the variable named on its left by adding, dividing by, etc., the value on its right.
specification (specs) [1-26,2-27,8-17]: a clear, complete, and consistent (i.e., unambiguous) statement of what a piece of software will do under all conditions in which it is intended to be used.
stack [2-1]: a list of values for which the primary operations are adding a new value and removing an old value; the latter operation always removes the value that has been on the list for the shortest period of time. So it is often called a LIFO (last-in-first-out) data structure.
stable sorting method [13-23]: a sorting method such that two values that are equal end up in the same relative order in the sorted list that they had in the unsorted list. This is often important for sorting objects, though irrelevant for sorting numbers.
statement [1-3]: the basic unit of a method body; the body is a sequence of statements.
structured natural language design [3-20,3-30]: a design of an algorithm entirely in English or another natural language, except (a) steps that are to be done only if a certain condition holds, or for as long as a certain condition holds, are indented beyond the description of the condition, and (b) values computed in one step and used in another step may be given a variable name.
stubbed documentation [6-26,7-2,11-2]: a list of the method headings of a class with comments describing their functions, and not much else. It can be compilable if you have minimal bodies, e.g., `{return 0;}`.
subclass [1-10,4-27]: a class defined with the phrase `extends SomeClass` in the heading; it is a subclass of the class named SomeClass. The subclass inherits each public method defined in SomeClass, i.e., statements and declarations in the subclass can use those public methods as if they were defined in the subclass.
subordinate [2-10,2-30,3-2,3-3,3-13,Interlude-4,Interlude-5]: When one statement is part of another, the first is said to be subordinate to the second. A `while` statement contains subordinate statements. An `if-else` statement contains two groups of subordinate statements, one to be executed if the `if` condition is true and the other to be executed if the `if` condition is false. If you want several statements to be subordinate to a `while` or `if`, you must enclose them in braces.
Sun standard library [1-31] hundreds of classes that come with the installation of Java that you obtain from Sun Microsystems, Inc., e.g., String, Math, Object.
superclass [1-10]: A class from which another class X inherits public methods. `SomeClass` is a direct superclass of X if X has the phrase `extends SomeClass` in its heading.
terminal window [1-4,1-6,1-14,1-17,1-18,12-7]: a window on the monitor, generally created by selecting MS-DOS Prompt, where you can enter commands to be executed by the operating system, such as `java SomeClass`.
test plan [7-8,8-8,9-15]: a large enough number of test sets that you can be confident that almost all of the bugs are out of the program when all of the test sets produce the expected results.

thread of execution [9-11,11-30,11-33]: An event-driven program has a thread of execution that responds to a user action when an event such as a button click or mouse movement occurs. If the user performs some other action (perhaps another button click) before that method finishes executing, the program will not respond to the second event -- the thread of execution that responds to events is busy. But you can have that event-handling thread spin off another thread of execution to respond to the first action. That lets the main event-handler go back to listening for events and therefore be able to respond to the second event.

timing execution [6-20]: determining how many milliseconds it takes to execute a piece of coding.

titlecase [1-15]: writing a name so that every letter is in lowercase except that the first letter of each English word inside the name is capitalized.

trace of a program [5-9,6-34]: a listing of the values that the most important variables have at various points in the execution of the program for one particular set of inputs. It is used for studying and debugging the program.

Turing machine [3-27]: an extremely simple and impractical computing device for which it is easy to prove what is and what is not computable.

type-checking an expression [6-35,6-36]: verifying by hand that the values that occur in an expression are of the type required by the expression.

UML [2-26,3-7,3-8,3-31,7-13]: the Unified Modeling Language for picturing relationships among classes and/or objects.

unary operator [Interlude-2]: an operator with one operand.

Unicode [6-14]: A specification that assigns an integer in the range 0 to 65535 to each character. It assigns consecutive integers to `'A'` through `'Z'`, also to `'a'` through `'z'`, and also to `'0'` through `'9'`. Characters with Unicode values below that of a blank are called whitespace. You may assign a char value to an int variable, in which case its Unicode value is used for the promotion.

utilities class [5-2,5-18]: a class that does not have any instance methods or instance variables or main method. Therefore, it is pointless to create any object of that class.

vacuously true [3-13]: a statement about all things in a group that is true because the particular group is empty. For example, "all living kings of Mexico are tall" is true because, if it weren't, you would be able to find a living king of Mexico who is short, and you can't because Mexico does not have any living kings.

variable [1-2,1-20]: a part of RAM where information (often a number) is stored. If a portion of the software gives a name to the variable, we often say that name is the variable.

virtual machine [1-28]: an abstract design of a computing machine.

visibility modifier [12-26]: any of `protected`, `private`, and `public`. `A protected` variable or method is accessible only by classes in the same package (e.g., disk folder) and by subclasses. A variable or method with no visibility modifier has default visibility, which means that it is accessible only by classes in the same package.

waterfall model [1-27]: the model of software development that has these five steps in this order: analysis, design, coding, testing, and maintenance.

wrapper class [11-12]: a class that does not add significant functionality; it only changes the name or the form of the values stored.

whitespace [6-16]: any one or more characters whose Unicode values are less than or equal to that of the blank character. Whitespace includes tabs, ends-of-lines, and ends-of-pages. But the official Java definition is only the characters with Unicode values of 9 through 13 and 28 through 32.

worst-case execution time [13-8]: the big-oh of the execution time for the case(s) that take the longest execution time. It is often more than the average-case execution time.

zero-based indexing [5-10,6-9,7-11]: assigning an index number to each component starting from zero instead of from 1.

Appendix C
Common Compile-Time Errors

The following are most of the compiler error messages you will get with your programs and what the most likely cause of the error is. Also be alert for the following points with the terminal window:

(a) You can't use arrow keys or highlighting;
(b) You have to click the closer X on an applet or frame or use CTRL/C before you can get the C:> prompt and give more instructions in the DOS window;
(c) You have to use CTRL/C to kill a program that has fallen into an infinite loop;
(d) If you have several hundred error messages, you probably saved your program as a document (with all the formatting codes), not a plain text file. Use an appropriate "Save as" to remove all formatting.

You will save yourself some trouble if you verify the following facts about your program before you compile it. For the purposes of this description, a brace-pair is a left brace at the beginning of a line plus a right brace further down and aligned with it; all material between them should be indented further than the two braces:

(a) The unindented part of your program is 0 or more import directives ending in semicolons, followed by a line beginning "public class Whatever" with no semicolon at the end, followed by a brace-pair directly below the "p" in "public". All other material in the file is between those braces and indented further at least one level from the left margin.
(b) All lines that are indented one level begin with either "public" or "private". Each ends in a semicolon (field variables) or in parentheses (method headings). Those that end in parentheses have a brace-pair directly below the "p" in "public" or "private". All other material in the program is indented at least two levels and is between some pair of braces for a method.
(c) No lines that are indented at least two levels begin with "public" or "private".

Can't convert boolean to java.lang.Boolean.
You misspelled boolean with a capital 'B'; it needs a small 'b'.

Can't make a static reference to nonstatic variable data in class X.
You declared the `data` variable <u>outside</u> of the `main` method (or outside of some other class method) instead of <u>inside</u> it.

Class Container not found.
You did not put `import java.awt.Container` at the top of your file, or you misspelled the name of the class.

Class Turtle not found.
You declared a variable of type Turtle but you did not compile the Turtle class first. Or you compiled it but in a different folder from the one you are now in. Or your particular computer has other software on it that interferes with the compiler finding the Turtle class.

Class or interface declaration expected. [suppresses other errors]
You misspelled the word `import`, or you have typed something after the right-brace that marks the end of the class.

'class' or 'interface' keyword expected. [suppresses other errors]
You have declared a method or variable outside of the class definition. Perhaps the preceding line ends in a right brace which, because you earlier accidentally left out a left brace, is taken as the end of the entire class when you meant it to be the end of the previous method.

Cannot resolve symbol. Symbol: data
The compiler cannot find a declaration of data. Perhaps you forgot to declare data in this class. Or you misspelled it. Or you are using it as an instance variable for one kind of object and it belongs to another kind of object.

Duplicate method declaration: void whatEver(int)
Maybe you meant to declare two methods with the same name and different parameter structures, but they do not differ in the parameters after all. Or you misspelled the method name, or you thought that just having a different return type would be okay (it isn't).

ExceptionInInitializerError in X.<init>
Your initialization of a class variable caused problems. An example is `private static int x = y / z` when `z` is a class variable whose value is zero.

Exception in thread "main" java.lang.NoClassDefFoundError X/java
You typed `java X.java`. You should type `javac X.java` to compile it. Or your program uses another class named X which you have not written with the name spelled that way.

Exception in thread "main" java.lang.NoSuchMethodError: main
You typed `java X` for an object class or a utilities class. You cannot "run" such classes. You should only try to run an application class, that is, a class that contains a `main` method.

Identifier expected. [suppresses other errors]
The method heading is `whatEver(x)` where you forgot to say what type `x` is. Or the preceding line ends in a right brace which, because you earlier left out a left brace, is taken as the end of the method instead of the end of a compound statement. Or you made an earlier error that confused the compiler so much that this is one of many error messages about things that are not really errors. Fix the earliest error first, then see if this one and others go away.

Incompatible type for =. Explicit cast required to convert double to int.
You tried to assign a decimal number to a whole-number variable. Use `(int)` to lose the fractional part.

Incompatible type for declaration. Explicit cast required to convert double to int.
You declared an int variable and initialized it to a double value, which is the wrong type.

Incompatible type for if. Can't convert int to boolean.
You had `if (x = 5)` but you need a double-equals `==` instead of an assignment symbol =.

Incompatible type for method. Can't convert anyOldThing to int.
The heading of the method you are calling has int for the parameter type, so you cannot put just anyOldThing in the parentheses of the method call and expect it to work; make sure it is an int.

Incompatible type for return. Can't convert java.lang.String to int.
The heading of the method you are in promises that it returns an int, but you have a String value following the word `return`. Either change the return type or change the value being returned.

Incompatible type for return. Explicit cast needed to convert X to Comparable.
You left out the phrase `implements Comparable` in the class heading for X. The heading of the method you are in promises it returns a Comparable value but you return an X value.

Invalid declaration.
The semicolon is missing at the end of the previous statement. Or you declared a variable directly after `else` or after `if(...)` (with no intervening left brace) or as the only statement in the body of a looping statement; this is illegal. You probably need to declare it before the compound statement, perhaps without assigning it a value.

Invalid expression statement. [suppresses other errors]
You called a method but forgot that every method call requires parentheses.

Invalid left hand side of assignment.
The part to the left of an assignment operator such as `=` or `+=` must be a variable. Perhaps you wrote `if(x + 2 = y)` when you mean to test for equality instead.

Invalid type expression.
The semicolon is missing at the end of this statement.

'}' expected.
The semicolon is missing at the end of this statement. Most of the time, an "expected" message means you left something out and the compiler is only wildly guessing what that something was.

'(' expected.
Perhaps you forgot that you must have a left parenthesis immediately after every `while` or `if`, and of course a matching right parenthesis at the end of the `while` or `if` condition.

')' expected. [suppresses other errors]
Perhaps you declared a variable inside the parentheses for an `if` or `while` condition; a declaration (e.g., `int n`) should only be a stand-alone statement or inside the parentheses of a for-statement. Or maybe you left out a plus sign between Strings or left out some other operator.

';' expected. [suppresses other errors]
You used a reserved word (such as const or break or native) as the name of a variable or method. Or maybe you just forgot to put the semicolon at the end of a statement.

Method takeCd() not found in class X.
You forgot to put "extends Vic" in the class heading so it could inherit `takeCD()`. Or you misspelled the name of the method you are calling (check the capitals). Or you have the wrong parameter pattern (maybe `takeCd` has a parameter). Or the executor's class is not the class that contains the `takeCd()` method.

No constructor matching X() found in class X.
Okay, so you have a constructor in X, but not one with EMPTY parentheses.

No method matching whatEver() found in class X.
You forgot to declare the method named `whatEver`, or you misspelled the name (check the capitals), or you declared it in another class besides X, or the method heading calls for one or more parameters and you supplied none in the parentheses.

No method matching whatEver(int, int) found in class X.
You have the wrong number of parameters for the method call, compared with the method heading, or they are of the wrong type. The call has two int parameters, the heading does not.

public class X must be defined in a file called "X.java".
The class heading is `public class X` but the name of the file is not precisely `X.java`. Perhaps a letter is wrongly capitalized or you misspelled it, either in the class name or the file name or even the javac command. Or you might have saved it without using Text Document, so WordPad added ".txt" to the name. Type `dir x*` to see all files that begin with "X".

Return required at end of int whatEver(double).
The method heading for `whatEver` promises it will return an int value, so the last statement in the method must be either an unindented return statement or an if-else statement with a return statement for all alternatives.

'return' with value from void whatEver(double).
The method heading for the `whatEver` method you are in promises it will not return any value (that is what void means), but you went ahead and put a value after the word `return` anyway.

Statement expected.
You declared a variable as private or public inside a method. You may only use private or public outside of a method. This message gives rise to later spurious errors such as "Identifier expected".

Statement not reached.
You have a statement after a return statement. But return means to exit the method immediately.

Undefined variable or class name: recieve.
You misspelled the variable name ("i before e except after c"), or you forgot to declare it before you used it. Or it is the name of a library class and you forgot the import statement. Or you are using Vic but it is not in your current folder. Or it is the loop control variable declared within a for-statement and you are trying to use it after the end of the for-statement.

Unclosed String literal.
If you start some quoted material on a line, you have to finish it on the same line. So the number of quotes on each line must be even. You probably need to put a quote at the end of the line pointed to, then a plus sign and a new quote at the beginning of the next line.

Variable 'sue' is already defined in this method.
You have declared `sue` twice in the same method. If you have previously declared `sue` and now you want to assign it 7, write `sue=7` instead of `int sue=7`.

Wrong number of arguments in method.
You spelled the name of the method right but you have the wrong number or type of parameters.

```
public class LotsOfErrors    // does not compile correctly
{
   private int data = 12;

   public void first()
   {  int sue = 5;
      private int sam = 2;  /// Statement expected.
      sue = 12.3;            // Incompatible type for =.
      sue - 3 = 12;         /// Invalid left hand side...
      methodSecond (false);  // Incompatible type for method.
      if (sue = 5)           // Incompatible type for if.
         int steve = 4;      // Invalid declaration.
      int sue = 7;           // Variable 'sue' already defined...
      sue = sam + 1         /// ';' expected.
      return 3;              // 'return' with value from void...
      int don = 3;           // Statement not reached.
   }

   public static int methodSecond (int par)
   {  data = 20;             /// Can't make a static reference...
      if (sue == 12)          // Undefined variable: sue.
         return true;         // Incompatible type for return.
      first;                 /// Invalid expression statement.
      Boolean x = false;      // Incompatible type for...
      String x = "go" "ne";  /// ')' expected
      if (int dru == 12)     /// ')' expected
         first(2);            // Wrong number of arguments...
      furst(3);               // No method matching furst()...
   }

   //public void main (args){} // Identifier expected.
}
```

Appendix D Java Keywords

Java has forty-eight keywords. You cannot declare the name of a variable, method, or class to be one of these keywords. If you try to do so, the compiler may give a rather puzzling error message.

The keywords come in the several categories, listed below. Where the keywords are listed on two lines, the second line consists of keywords defined in the text in one place and never used anywhere else in the text (thus you do not need to remember them).

8 primitive types of values:

```
boolean char int double long
byte short float
```

3 constant values (technically these are two boolean literals and one null literal instead of keywords):

```
true false null
```

14 statement indicators (the first word in the statement except for `else` and `catch`):

```
if else while do for return try catch throw
switch case default continue break
```

11 modifiers (can appear before the return type of a method):

```
public private static final abstract
protected synchronized native volatile transient strictfp
```

6 global (can be used outside of a class body):

```
class interface extends implements import
package
```

7 miscellaneous:

```
void new this super throws instanceof
finally
```

2 with no meaning, reserved so that people who accidentally put these C++ words in their programs will receive useful error messages:

```
const goto
```

Appendix E Sun Library Classes

The following are all of the Sun standard library packages, classes, and interfaces described in this book. We list them by package, and within each package we list one class per line, with indentations to show which are subclasses of which others in that package. Classes on the same level are listed alphabetically.

Interfaces have the word "interface" after them. After each class or interface we list the signatures and return types of all of their methods that are mentioned in this book other than those listed for the indicated superclass or interface. Class methods are indicated by "static" before the method name. The numbers in brackets are page references.

java.lang

```
    Object  [4-22,7-4,7-35]  boolean equals(Object), String toString()
        Boolean  [11-12,11-13]  new(boolean), new(String), boolean booleanValue(),
                    static Boolean TRUE, static Boolean FALSE
        Character  [11-13,11-36]  new(char), char charValue(),
                    static boolean isDigit(char), static boolean isWhiteSpace(char),
                    static boolean isLetter(char), static char toLowerCase(char),
                    static boolean isLowerCase(char), static boolean isUpperCase(char)
        Comparable interface  [6-10,6-11,6-12,11-5,11-12,13-1]  int compareTo(Object)
        Math  [6-21,6-40,11-35,11-36]  static double abs(double),
                    static double min(double,double), static double max(double,double),
                    static int min(int,int), static int max(int,int), static long min(long,long),
                    static long max(long,long), static double pow (double,double),
                    static double cos(double), static double sin(double), static double E,
                    static double log(double), static double exp(double), static double PI,
                    static double random(), static double sqrt(double),
                    static double rint(double), static double ceil(x)
        Number  [11-12]  static double doubleValue(), int intValue(), long longValue(),
                    float floatValue(), byte byteValue(), short shortValue()
            Byte  [11-12]  new(byte), new(String)
            Double  [6-2,6-8,6-40,11-12]  new(double), new(String),
                        static double parseDouble(String)
            Float  [11-12]  new(float), new(String)
            Integer  [4-18,11-12,11-13] new(int), new(String),
                        static int parseInt(String), static String toString(int,int),
                        static int parseInt(String,int),
                        static int MAX_VALUE, static int MIN_VALUE
            Long  [11-12] new(long), new(String), static long parseLong(String),
                        static long MAX_VALUE, static long MIN_VALUE
            Short  [11-12]  new(short), new(String)
        Runnable interface  [11-30,11-31]  run()
        String  [3-4,4-22,5-10,5-32,6-9,6-10,6-11,6-14,6-15,6-40,7-35,7-36] int length(),
                    boolean equals(String), String substring(int,int), String substring(int),
                    char charAt(int), int compareTo(String), String concat(String),
                    int indexOf(String), int indexOf(String,int), boolean endsWith(String),
                    boolean startsWith(String), String trim(), String toLowerCase(),
                    String toUpperCase(), int indexOf(char), int indexOf(char,int),
                    String replace(char,char), char[] toCharArray(), new(char[]), boolean
                    equalsIgnoreCase(String), int compareToIgnoreCase(String)
        StringBuffer  [7-36,7-37]  new(String), int length(), char charAt(int),
                    setCharAt(int,char), String toString(), StringBuffer append(int,String),
                    StringBuffer delete(int,int), StringBuffer insert(int,String),
                    StringBuffer replace(int,int,String)
```

- System [1-23,1-24,Int-3,4-1,5-3,5-4,5-9,5-10,6-20,7-34,7-35,9-11] static long currentTimeMillis(), out.println(String), out.print(String), static exit(int), static arraycopy (Object[],int,Object[],int,int)
- Thread [9-11,9-35] new(Runnable), start(), static boolean interrupted(), static sleep(int), interrupt(),
- Throwable [9-9,9-34,9-35] new(String), new(), String getMessage(), printStackTrace(), fillInStackTrace(), String toString()
  - Error [9-35]
    - AWTError [9-35]
    - LinkageError [9-35]
    - ThreadDeath [9-35]
    - VirtualMachineError [9-35]
  - Exception [6-8,6-38,9-3,9-4,9-9,9-10,9-12,9-34] new(), new(String)
    - InterruptedException [9-11,9-12] new(), new(String)
    - RuntimeException [6-8,9-3,9-10,9-12,9-13,9-35] new(), new(String)
      - ArithmeticException [6-8,9-3] new(), new(String)
      - ArrayStoreException [7-35] new(), new(String)
      - ClassCastException [6-12,6-31,9-3] new(), new(String)
      - IndexOutOfBoundsException [6-9,6-14,7-10,9-3] new(), new(String)
        - ArrayIndexOutOfBoundsException [7-10] new(), new(String)
        - StringIndexOutOfBoundsException [7-10] new(), new(String)
      - NegativeArraySizeException [7-10,7-19,9-3] new(), new(String)
      - NumberFormatException [6-8,9-3] new(), new(String)
      - NullPointerException [6-8,6-22,7-14,9-3] new(), new(String)
      - IllegalStateException [14-6,15-19] new(), new(String)
      - IllegalArgumentException [15-23,15-24,15-25,18-7,18-10] new(), new(String)
      - UnsupportedOperationException [15-22,15-26,16-10] new(), new(String)
    - SecurityException [12-31] new(), new(String)

javax.swing

- JApplet (extends java.applet.Applet) [8-1,8-2,8-3,10-2,10-5,10-6,10-33,10-40] Container getContentPane(), JMenuBar getJMenuBar(), setJMenuBar(JMenuBar), addKeyListener(KeyListener)
- BorderFactory [10-38] static Border createLineBorder(Color)
- Box [10-36] createGlue()
- BoxLayout (implements LayoutManager) [10-7,10-36] new(Container,int), static int X_AXIS, static int Y_AXIS
- ButtonGroup [10-30,10-31,10-42] new(), add(AbstractButton), remove(AbstractButton)
- JComponent (extends java.awt.Container) [10-5,10-17,10-18,10-27,10-38] setToolTipText(String), setBorder(Border), setFont(Font), setIcon(ImageIcon)
  - AbstractButton [10-12,10-29,10-30,10-32,10-40,11-8] String getText(), addItemListener(ItemListener), addActionListener(ActionListener), setText(String), setMnemonic(char), setActionCommand(String)
    - JButton [10-12,10-13] new(String)
    - JMenuItem [10-33,10-34] new(String)
      - JMenu [10-33,10-34] new(String), JMenuItem add(JMenuItem), add(String)
    - JToggleButton [10-29] boolean isSelected(), setSelected(boolean)
      - JCheckBox [10-29,10-30] new(String,boolean)
      - JRadioButton [10-29] new(String,boolean)
  - ImageIcon [10-38] new(String), paintIcon(Component,Graphics,int,int)

JComboBox [10-26,10-27,10-28,10-40] addActionListener(ActionListener), new(Object[]), setSelectedIndex(int), int getSelectedIndex(), setMaximumRowCount(int), Object getSelectedItem(), addItem(Object), removeItemAt(int), setActionCommand(String)
JFileChooser [12-33] new(), int showSaveDialog(null), static int APPROVE_OPTION, static int CANCEL_OPTION, static int ERROR_OPTION, int showOpenDialog(null), File getSelectedFile()
JFrame [10-1,10-2,10-3,10-4,10-5,10-27,10-32,10-39] new(String), String getTitle(), setTitle(String), Container getContentPane(), addWindowListener(WindowListener), addKeyListener(KeyListener), JMenuBar getJMenuBar(), setJMenuBar(JMenuBar)
JLabel [10-8] new(String), setText(String), String getText()
JList [10-37] new(Object[]), setVisibleRowCount(int), int getSelectedIndex(), Object getSelectedValue(), setSelectedIndex(int), addListSelectionListener(EventListener)
JMenuBar [10-33] new(), new(String), JMenu add(JMenu)
JOptionPane [4-3,4-4,4-5,4-36,5-31,10-15] static showConfirmDialog(null,Object), static String showInputDialog(Object), static int YES_OPTION, static int showMessageDialog(null,Object), static int YES_NO_OPTION, static int PLAIN_MESSAGE, static int ERROR_MESSAGE, static int WARNING_MESSAGE, static int QUESTION_MESSAGE, static int INFORMATION_MESSAGE
JPanel [10-5,10-7,10-38] new(), new(String)
JScrollPane [6-23,6-24,6-25,6-40] new(Component)
JSlider (implements SwingConstants) [10-19,10-20,10-21,10-22,10-40] new(int), addChangeListener(ChangeListener), int getValue(), setMinimum(int), setMaximum(int), setMinorTickSpacing(int)
JTextArea (extends JTextComponent) [6-23,6-24,6-25,6-40] new(int, int), append(String)
JTextField (extends JTextComponent) [10-8,10-9,10-10,10-12] new(String), new(int), setActionCommand(String), addActionListener(ActionListener)
JPasswordField [10-10] new(int), char[] getPassword()
Timer [10-23,10-24,10-25,10-42] new(int,ActionListener), start(), stop(), setRepeats(boolean)
SwingConstants interface [10-20] static int HORIZONTAL, static int VERTICAL, static int NORTH, static int SOUTH, static int EAST, static int WEST, static int CENTER, static int BOTTOM, static int TOP

javax.swing.event
ChangeEvent (extends EventObject) [10-20,10-42]
ChangeListener interface [10-20,10-43] stateChanged(ChangeEvent)
ListSelectionEvent [10-37]
ListSelectionListener interface [10-37] valueChanged(ListSelectionEvent)

javax.swing.text
JTextComponent (extends javax.swing.JComponent) [10-8,10-18] setText(String), String getText()

java.applet
Applet (subclass of Panel -> Container -> Component) [8-32,10-38] init(), start(), stop(), URL getDocumentBase(),AudioClip getAudioClip(URL,String)
AudioClip interface [10-38] play(), loop(), stop()

java.awt

BorderLayout (implements LayoutManager, SwingConstants) [10-5,10-36] new(int,int)

CardLayout (implements LayoutManager) [10-37] new(), first(Container), last(Container), next(Container), previous(Container)

Color [8-1,8-2,8-4] new(int,int,int), static Color black, static Color cyan, static Color darkGray, static Color lightGray, static Color magenta, static Color green, static Color yellow, static Color red, static Color blue, static Color white, static Color orange, static Color gray, static Color pink

Component [8-2,8-3,8-6,8-32,10-2,10-7,10-39] paint(Graphics), int getWidth(), int getHeight(), setSize(int,int), Graphics getGraphics(), setVisible(boolean), setBounds(int,int,int,int), addMouseListener(MouseListener), repaint()

Container [10-5,10-36,10-39] setLayout(LayoutManager), add(Component,int), add(Component), LayoutManager getLayout()

Window [10-4] addWindowListener(WindowListener)

FlowLayout (implements LayoutManager) [10-5,10-7] new()

Font [8-30] new(String,int,int), static int PLAIN, static int BOLD, static int ITALIC

Graphics [8-1,8-2,8-4,8-31] Color getColor(), setColor(Color), drawString(String,int,int), setFont(Font)

Graphics2D [8-2,8-4,8-31] draw(Shape), fill(Shape)

GridLayout (implements LayoutManager) [10-36] new(int,int), new(int,int,int,int)

LayoutManager interface [10-5,10-36] layoutContainer(Container)

Polygon (implements Shape) [8-30,8-31] new(int[], int[], int), new(), addPoint(int,int), translate(int,int,int)

Rectangle (implements Shape) [8-5,8-31,8-32] new(int,int,int,int), grow(int,int), translate(int,int), setLocation(int,int), setSize(int,int)

Shape interface [8-4,8-31] boolean contains(Point2D)

java.awt.geom

Ellipse2D (implements Shape) [8-6]

Ellipse2D.Double [8-6,8-32] new(double,double,double,double)

Line2D (implements Shape) [8-2,8-4,8-6,8-31] ptLineDist(Point2D)

Line2D.Double [8-2,8-4,8-6,8-32] new(double,double,double,double), new(Point2D,Point2D)

Point2D (implements Shape) [8-31]

Point2D.Double [8-31] new(double,double), double getX(), double getY(), setLocation(double,double)

Rectangle2D (implements Shape) [8-6]

Rectangle2D.Double [8-6,8-32] new(double,double,double,double)

RoundRectangle2D (implements Shape) [8-6]

RoundRectangle2D.Double [8-6,8-32] new(double,double,double,double,double,double)

java.awt.event

ActionEvent (extends EventObject) [10-10,10-13,10-42] String getActionCommand()

ActionListener interface [10-10,10-13,10-14] actionPerformed(ActionEvent)

ItemEvent (extends EventObject) [10-30,10-31,10-42] int getStateChange(), static int SELECTED, static int DESELECTED

ItemListener interface [10-30,10-31] itemStateChanged(ItemEvent)

KeyAdapter (implements KeyListener) [10-32] keyTyped(KeyEvent) and 2 others

KeyEvent [10-32] char getChar()

MouseAdapter (implements MouseListener) [10-35]

MouseEvent (extends EventObject) [10-35,10-42] int getX(), int getY()

MouseListener interface [10-34,10-35,10-43] mouseClicked(MouseEvent), mouseEntered(MouseEvent), mouseExited(MouseEvent), mousePressed(MouseEvent), mouseReleased(MouseEvent)

WindowAdapter (implements WindowListener) [10-35]

WindowEvent (extends EventObject) [10-3,10-4]

WindowListener interface [10-3] windowClosing(WindowEvent) and 6 others

java.util

AbstractCollection (implements Collection) [15-35] abstract size(), abstract iterator()
  AbstractList (implements List) [15-35] abstract size(), abstract get(int)
    ArrayList [7-32,7-33,15-35,18-3,18-7] new(), new(Collection), int size(), Object get(int)
    Vector [14-37] new(), new(int), Object elementAt(int), Object firstElement(), setElementAt(Object,int), addElement(Object), Object lastElement(), removeElementAt(int), removeAllElements(), insertElementAt(Object,int), int size(), ensureCapacity(int), Enumeration elements()
      Stack [14-37] push(Object), Object pop(), Object peek(), boolean empty(), int search(Object)
AbstractMap (implements Map) [16-33] abstract Set entrySet()
Arrays [13-33] static boolean equals(Object[],Object[]), static sort(Object[],int,int), static boolean equals (anyPrimitiveType[],sameType[]), static sort(anyPrimitiveType[]), static binarySearch(Object[],Object), static sort(anyPrimitiveType[],int,int), static fill(Object[],Object), static fill(anyPrimitiveType[],sameType), static sort(Object[]),
Collection interface [15-4,15-23,15-36] boolean add(Object), boolean add(Collection), boolean isEmpty(), int size(), Iterator iterator(), clear(), boolean contains(Object), boolean containsAll(Collection), boolean retainAll(Collection), boolean equals(Object), boolean remove(Object), boolean removeAll(Collection), Object[] toArray(), Object[] toArray(Object[])
  List interface (extends Collection) [15-34,15-37] Object get(int), remove(int), set(int,Object), add(int,Object), addAll(int,Collection), ListIterator listIterator(), ListIterator listIterator(int), int indexOf(Object), int lastIndexOf(Object), List subList(int,int)
  Set interface (extends Collection) [15-23] // unique elements, no more methods
Comparator interface [13-33,18-5] int compare(Object,Object), boolean equals(Object,Object)
EventListener [10-20] // no methods, so this is a "tagging" interface
EventObject [10-13] Object getSource()
HashMap (implements Map) [16-28,16-33] new(), new(Map)
Iterator interface [15-16,15-17,15-18,16-9,16-10,16-23,17-8,17-19,18-8] boolean hasNext(), Object next(), remove()
  ListIterator interface [15-26,15-32,15-37] add(Object), set(Object), int nextIndex(), boolean hasPrevious(), Object previous(), int previousIndex()
Map interface [16-8,16-32,16-33,17-10] put(Object,Object), Object get(Object), int size(), boolean containsKey(Object), Object remove(Object), boolean isEmpty(), clear(), boolean containsValue(Object), putAll(Map), Set entrySet(), Set keySet(), Collection values()
Map.Entry interface [16-13,16-33] getKey(), getValue(), setValue(), equals(Object)
NoSuchElementException (extends RuntimeException) [15-18,16-9,16-10] new(), new(String)
Random [4-16,4-18,6-37] new(), int nextInt(), int nextInt(int), double nextDouble(), long nextLong(), float nextFloat(), double nextGaussian()
StringTokenizer [12-10,12-34] new(string), boolean hasMoreTokens(), int countTokens(), String nextToken()
TreeMap (implements Map) [16-28,16-33] new(), new(Map)

java.io

File [12-33] boolean exists(), delete(), boolean canRead(), boolean canWrite()
IOException [9-7,9-9,9-12,12-5,12-7,12-31,15-2] new(), new(String)
RandomAccessFile [12-22,12-34] new(String,String), writeInt(int), int readInt(),
writeChars(String), String readLine(), writeDouble(double),
double readDouble(), writeChar(char), char readChar(),
writeBoolean(boolean), boolean readBoolean(), close(),
int length(), seek(long)
Reader [12-4,12-5,12-25] abstract read(char[],int,int), abstract close()
BufferedReader [9-7,12-2,12-3,12-4,12-5,12-33] new(FileReader),
new(InputStreamReader), String readLine()
InputStreamReader [9-7,12-3,12-4] new(InputStream)
FileReader [9-7,12-2,12-3] new(String)
StreamTokenizer [12-10] new(Reader), int nextToken()
Writer [12-6,12-7,12-23] abstract write(char[],int,int), abstract flush(), abstract close()
FileWriter [12-6,12-7] new(String)
PrintWriter [12-6,12-7,12-33] new(Writer), flush(), close(), println(String),
print(String)

java.io.print

PrinterAbortException [9-10] new(), new(String)

java.text

NumberFormat [6-37,6-38] static setMaximumFractionDigits(int), format(double),
static setMinimumFractionDigits(int),
static NumberFormat getNumberInstance()
DecimalFormat [6-38] new(String)

java.net

ServerSocket [12-31,12-32] new(int,int), accept(), close()
Socket [12-31,12-32] new(String,int), InputStream getInputStream(),
OutputStream getOutputStream(), close()

Appendix F Major Programming Projects

Problem number C.N indicates chapter C and problem N for that chapter. Project descriptions marked with & are purposely ambiguous and incomplete. For them, you are to write out an analysis of the situation in the form of several questions addressed to the client to clear up ambiguities, plus answers that the client might plausibly respond with. You should write out the top-level logic design and object design before attempting implementation or lower-level design. Your instructor may ask you to submit the analysis and design for evaluation before you go on to the implementation stage.

3.1 **ShiftToEnd:** Write an application program using Vics that clears all the slots to the stack for the first sequence of slots, then puts those CDs in its last few slots. For instance, if there are 5 slots with 2 CDs in them somewhere, you should end with those CDs only in the fourth and fifth slots. Do not assume that the stack is initially empty. Extra Credit: Repeat for all sequences of slots.

3.2 **DoubleUp:** Write an application program that first clears the slots of the second sequence to the stack, then finishes with twice as many CDs in the first sequence's slots as it had to start with (or fewer if the first sequence becomes filled). Precondition: The second sequence is known to have more CDs in its slots than the first sequence has.

3.3 **FillTheFirst:** Write an application program that puts all the CDs from the second, third and fourth sequences of slots into the first sequence's empty slots. But if there are too many CDs for that, put the excess in the second sequence's empty slots (and if there are still some left over, fill in the third sequence's empty slots to the extent possible).

3.4 **MoveBackByOne:** Write a class with an instance method that moves back by one slot every CD for which, at the time execution of the method begins, (a) the CD is beyond the current slot, and (b) the CD is directly preceded by an empty slot. Test it with an appropriate main method.

3.5 **ShiftOver:** Write a class with an instance method that has a Vic parameter: Each empty slot of the parameter's sequence is to be filled by the next available CD from the executor's sequence, until the executor's sequence runs out of CDs or the parameter's sequence runs out of empty slots. Test it with an appropriate main method.

3.6 **FillSlots:** Write a class with an instance method that moves all CDs in a sequence's slots to the front of the sequence, without having more than one of its CDs on the stack at any given time. Test it with an appropriate main method.

3.7 **AddBinaries:** Write an application program that "adds" the second sequence to the first sequence, treating each sequence as if it is a binary number written in reverse order of digits, with a CD considered a 1 and the absence of a CD considered a 0. Precondition: The first sequence is at least as long as the second sequence and the last slot of the first sequence is empty (this avoids number overflow).

4.1 **StudentGrades**: Write a complete Student class that allows you to use the statements such as the following in methods in outside classes, with the obvious meanings. Then write an application program that creates one Student object, reads in test grades until a negative "grade" is seen (as a signal that the list of grades has ended), then prints out all available information about the Student:

```
Student bob = new Student ("Robert", "Newhart");
bob.storeGrade (23);          // bob scored 23 on this test
return bob.getTotalGrades();  // total of test scores to date
return bob.getAverageGrade(); // for all test scores to date
return bob.getName();         // returns "Robert Newhart"
```

4.2 **Counter:** Write a complete Counter class that allows you to use statements such as the following in methods in outside classes, with the obvious meanings. Then write an application program that creates one Counter object, reads in a sequence of numbers until the user indicates it is time to quit, calls `recordNumber` for each number in the sequence, and then prints out all available information about the Counter:

```
Counter sam = new Counter();
sam.recordNumber(x);       // sam adds x to its values counted
return sam.getPositives(); // number of positive numbers
                           // recorded so far
return sam.getNegatives(); // number of negative numbers
                           // recorded so far
return sam.getZeros();     // number of zeros recorded so far
```

4.3 **Dates:** Write a complete Date class with a constructor `new Date(month, day, year)` that creates a Date object with the given month, day and year. Use a hypothetical calendar in which every month has 30 days. A day value outside the range 1 through 30 is to be replaced by 1. For a negative month number, add N*12 to it and subtract N from the year, to make the month number 1 through 12. Make the opposite adjustment for month numbers greater than 12. Write an `add` method, a `subtract` method, a `toString` method, and a `compareTo` method. Also have two more constructors, one to construct a Date with the same values as a given Date, and another to construct a Date with only the year supplied (make it January 1). Extra credit: Use a real calendar for the number of days in each month, except ignore the possibility of leap years.

4.4 **DiceGame:** Write a subclass of BasicGame for which one game is the roll of two dice, reporting the two values and the total. The player wins if the roll is a 7 or 11, loses if it is 12 or 2, and ties in all other cases. Extra credit: There are no ties. If the player's roll total of X is other than 7, 11, 2, or 12, the player rolls again until a total of either 7 or X comes up. The player wins only if X comes up before 7.

4.5 **MakeThreeEqual:** Write a subclass of Vic with instance methods that (a) tell how many CDs the executor has and (b) put a given number of CDs in its first few slots (the given number is a parameter). Then write an application program for Vics that finds out how many CDs each of the first three sequences of slots has. Find the smallest number N of those three numbers. For each of the sequences that has more than N CDs in its slots, clear out all of its slots to its stack and then put back N CDs in its first few slots. For purposes of this program, N is zero if the machine has only one or two sequences of slots.

4.6 **GuessMyNumber:** Revise the GuessNumber class so that the computer lies about having chosen a number. Instead, it tells the user "too high" or "too low" depending on which will make the user take the greatest number of guesses to get it right. It only says the user is right when there is only one possible answer consistent with its previous answers and the user has guessed it.

4.7 **Mastermind4:** Revise the Mastermind game to have the user guess four digits, not three.

4.8 **GuessThatNumber:** Write a subclass of BasicGame in which the computer asks the user to choose a whole number from 1 to 100, inclusive. The program then makes a number of guesses. After each guess, the user tells whether the guess is too high, too low, or exactly right. Once the computer "knows" what the correct number is, the computer tells the user the correct number and the number of guesses tried. Have the program ask the user after each game whether she wants to play again. Design the program so that it always guesses the correct answer by the seventh try at latest. Hint: Have two instance variables that keep track of the smallest and largest values that the guess could be (initially 1 and 100). Guess halfway between them. After each wrong guess, update these smallest and largest values appropriately.

4.9 **RandomWalk:** Position five Turtles on the drawing area 100 pixels apart. Have each one draw a circle 25 pixels in radius. Then have each Turtle repeat the following 1000 times, painting as it goes: Choose one of the directions North, South, East, or West at random and move one pixel in that direction. This is called a random walk, so you should have five random-walk paths. Record the number of Turtles that ended up outside their circles. Repeat at least 20 times until you have a good estimate for the probability that a 1000-step random walk ends up more than 25 steps away from its starting point.

4.10 **CoinMatch:** Create a Die class that keeps track of the number (1 through 6) that was on top of the Die when last rolled (initialize it with 1). However, the constructor should allow any whole number of sides to the Die greater than 1. For instance, if constructed with two sides, it would in effect be a coin. Provide methods to roll the Die and to retrieve what was last on top.

Write a subclass of BasicGame that uses the Die class to create two coins, one for the user and one for the computer opponent. Flip the coins, recording a win for the user if they match on heads, a win for the computer if they match on tails, and no-win if they do not match. Report on the status after each flip. The winner of the entire game is the one who gets 10 wins first.

4.11 **DoorLock &:** A dedicated computer manages all the door locks in a building. Each door lock is a number pad beside the door. A user must enter the correct sequence of digits and then turn the handle to open the door. An alarm goes off if the user enters the wrong combination too many times in a row. Write a complete analysis and design for this problem. Then completely implement a DoorLock class of objects that would appropriately represent a single 3-digit door lock. It should include an instance method void `lock()` which cancels a partial sequence of digits entered for the combination.

4.12 **ScissorsPaperRock:** Write a subclass of BasicGame that plays the Scissors-Paper-Rock game (ask a friend to demonstrate if you do not know the rules). On each turn, the computer chooses one of these three at random and then asks the user for one of "S", "P", or "R". Assume the choice is "R" if the user gives bad input. One game is 15 such turns. After each turn, the computer announces how many it has won, how many it has lost, and how many it has tied (typically 5 of each possible outcome). Alternatively, play until one player has won 10 games, or play until one player is 3 ahead of the other.

4.13 **Wallets:** Write a Wallets class, where one Wallet object keeps track of the number of pennies, nickels, dimes, quarters, and dollars in it (only the five categories; treat $5 and $20 bills as if they were 5 1-dollar and 20 1-dollar entries). Include methods such as `x.add(y)` to add the contents of y to x, `x.subtract(y)` to subtract them, `x.canMakeChange(n)` if x has money worth exactly n cents, `x.subtract(n)` to subtract exactly n cents worth of value from the contents of x (n is an int value here), `x.totalValue()`, and several methods along the lines of `x.getNumQuarters()`.

5.1 **MilitaryTime:** Write a class containing only class methods to do computations involving military and civilian time. Consider that 0900 is "9:00 AM" and 1420 is "2:20 PM". The military times are int values ranging from 0000 up to 2359. The class should include methods such as `toCivilian(int)` returning a String, `toMilitary(String)` returning an int, and `isMilitary(int)` returning a boolean. Also, `getHoursLater(int,int)` should tell how many whole hours the second parameter is later than the first, and `getMinsLater(int,int)` should return an answer 0 to 59 telling how many minutes the second parameter is later than the first, ignoring whole hours. You should also have the same two methods overloaded with two String parameters for civilian time. That way, you can print the following:

```
t1 + " is " + getHoursLater (t1, t2)  " hours and "
      + getMinsLater (t1, t2) " minutes after " + t2
```

5.2 **MathOp:** Add the following class methods to the MathOp class (Listing 5.1), then write a simple application program to test them all:

1) `abs` returns the absolute value of an int parameter.
2) `sqrt` returns the largest int value whose square is not greater than the int parameter. Return -1 if the parameter is negative.
3) `max` is overloaded: three methods that return the largest of two, three, or four int parameters.
4) `min` is overloaded: three methods that return the smallest of two, three, or four int parameters.
5) `sum` returns the sum of all int values that are equal to or between two int parameters.
6) `gcd` returns the greatest common divisor of two int parameters (as described in an exercise at the end of Section 5.1).
7) `log2` returns the highest int value such that 2 to that power is not more than the int parameter.

5.3 **TwoCharacterCDs:** Revise the given Vic implementation so that the name of each CD is two characters, not one. Use "00" for the absence of a CD. For instance, `"  a100itis00"` is the way you would record a five-slot sequence consisting of "a1" in the first slot, "it" in the third slot, "is" in the fourth slot, and no CD in slots two and five. A String received as input in `reset` is to have the digit for its sequence appended to each character representing a CD, e.g., if the second value `args[1]` is `"a0bc"`, you are to represent the sequence as `"  a200b2c2"`.

5.4 **StringBufferVics:** Read the description of the StringBuffer class in Section 7.12. Then revise the given Vic implementation to store the stack and all sequences as StringBuffer objects, but with no change in effect.

5.5 **TicTacToe:** Write TicTacToe as a subclass of BasicGame. Have the computer pick the first available open spot each time it is its turn. Display the board as a String value with two of `'\n'` in it, to get three lines displayed. Extra credit: Improve the computer's simple-minded strategy so that the computer never loses.

5.6 **Hangman &:** Write Hangman as a subclass of BasicGame. Each word to be guessed should have five letters. Have a 250-letter String class variable that stores 50 possible words to choose at random.

5.7 **GPA:** Write an application program that accepts from the user a number of String inputs representing grades in various college courses. When the user responds to `showInputDialog` with a value that does not have exactly either 1 or 2 characters, print the grade-point average and terminate the program. The single letters A, B, C, D, F have the values 4, 3, 2, 1, and 0. If they have a "+" or "-", add or subtract 0.3 respectively. Note: You do not need to know how to calculate using numbers with decimal points in Java to do this program, but it would make the program easier to do.

5.8 **SmarterNet:** Write a subclass of the SmartNet class (Listing 5.8) having the following instance methods:

1) `averageEdgesPerNode` returns the average number of edges per node.
2) `numberOfEdges` returns the total number of edges (do not forget to divide by 2).
3) `averageForConnections` returns the average number of edges per node connected to a given node parameter.
4) `hasTriangle` (Extra credit) tells whether there is any node which connects to a different node which connects to a third node (different from both of those) which connects back to the first node.

5.9 **Tums:** Implement the Tum class described in Section 3.9 similar to the String implementation of the Vic class. The following is the constructor done for you. Extra credit: Modify the constructor to replace any non-digit character by a blank.

```
public Tum (String given)
{  super();
   if (given == null)
      itsSequence = " ";
   else
      itsSequence = " " + given;
   itsPos = 1;
}
```

6.1 **BorrowMoney:** Write a program that repeatedly accepts an amount of money from the user, stopping when the user says to. After each positive input, which represents an amount borrowed, ask the user for the annual interest rate (e.g., 7.2 represents 7.2% annual rate) and the number of months over which the loan is to be paid off. Calculate and print each monthly payment, but print an error message if either input is non-positive.

Hint: First convert the interest rate to a monthly rate R (e.g., 7.2% corresponds to R = 0.006). The "infinity" monthly payment, what you would have to pay monthly if the loan ran on forever, is R times the amount borrowed. You obviously have to pay more than that monthly to pay off the loan. The actual amount is calculated by dividing the "infinity" monthly payment by 1-(1/F), where F is the result of taking 1+R to the power equal to the number of months (use a while-loop to get the power).

6.2 **UlamValues:** Write a program that asks the user for a positive integer and then calculates the Ulam value for each of the 100 integers from that point on up (for instance, if the input is 3200, calculate the Ulam value for each number 3200 up to but not including 3300). Report the largest Ulam value seen and the number that has that Ulam value. The Ulam value for a number N is defined as the number of iterations of the following process required to get to the number 1, starting with N: Divide N in half if N is even, otherwise multiply N by 3 and then add 1.

6.3 **Calendar &:** Write a program that reads in a date as three integers Month/Day/Year. It then tells the user what day of the week that was. Have it work correctly for every date from 1/1/1753 on. It is useful to know that January 1, 1753 was a Monday, and that every year divisible by 4 except 1800, 1900, 2100, etc. is a leap year (i.e., not centesimal years except those divisible by 400). The reason for starting with 1753 is that we switched to the Gregorian calendar in 1752, losing 11 days. Extra credit: Print out a calendar for the entire month of the date requested.

6.4 **RepairQueue:** Revise the RepairShop class (Listing 6.10) to use appropriately a RepairQueue class of objects that extends the Queue class. A RepairQueue has two instance variables, `itsTotalTime` and `itsCount` (remove `totalTime` from the RepairShop class). It has three instance methods: `remove()` removes and returns a RepairOrder object and updates its instance variables, `add(RepairOrder)` adds one RepairOrder and updates its instance variables, and `jobDescription()` returns a two-line string giving the job most recently returned, the total hours for the remaining jobs, and number of jobs remaining on the queue.

6.5 **JustifiedOutput:** Add six independent class methods to your own library classes: `rightJustify (x,10,4)` returns the string representation of the decimal value in x, with exactly four digits after the decimal point and a total of ten characters altogether (including the decimal point and leading blanks). Similarly, `rightJustify (n,10)` returns the string representation of the integer value in n, with enough leading blanks added to make a total of ten characters altogether. Add two more for `leftJustify` and two more for `centered`. Note: This meaning and order of the parameters is almost the same as what is used in the output statements that are built-in with Pascal and Fortran.

6.6 **BankAccount:** Write a BankAccount class with these instance variables: `itsCurrentBalance`, `itsOwner` (a String), `itsNumDepositsMadeThisMonth`, and `itsMinimumBalanceForThisMonth`. Have methods that get these values and methods that change the owner's name. Also have methods that accept an amount for deposit or withdrawal (which must be positive numbers, otherwise the method call has no effect). Any check that brings the balance below zero is to be bounced with a $25 penalty. Add a `printMonthlyStatement()` method that prints the balance as of the end of the current month, together with fees, interest, and the values of all instance variables. The interest is 3% annually compounded monthly, paid on `itsMinimumBalanceForThisMonth`. The fees are (a) 15 cents per check that was cashed when the balance was below $1000, (b) 10 cents per deposit, and (c) $3 per month. However, waive all fees if the balance was never below $3000 at any point during the month.

Write an application program that creates three BankAccount objects numbered 1, 2, 3. It makes #1 the current account, then accepts user requests to credit or debit the current account, print a monthly statement for it, change the owner's name for it, or switch over to an account with a different number.

6.7 **NumbersWithCommas:** Add two independent class methods to your own library classes: One accepts a long value and returns the String equivalent with commas every three digits. The other accepts a String value that should represent an integer with commas every three digits and returns the long value corresponding to it. Make sure you do not put in any unnecessary commas or leading zeros.

6.8 **Quadratics:** Develop a Quadratic object class whose instances represent quadratic equations. A Quadratic object should be able to return its value for a given x-value, tell whether it has any zeros, return one of those zeros (which one depends on a boolean parameter), modify itself so that its graph translates horizontally or vertically (have two parameters to specify the translation), and say where its inflection point is. Have three instance variables for its three coefficients plus one for its discriminant (b*b-4*a*c). Have two constructors, one with three coefficients as parameters and one with a String parameter that is supposed to contain three numerals. Write a main method that tests out all of the Quadratic methods.

6.9 **Shorthand:** Write a program that repeatedly accepts a line of input and produces the shorthand form of that line, defined as follows: (a) remove all vowels ('a', 'e', 'i', 'o', 'u', whether lowercase or uppercase), except (b) replace these five words: "and" by "&", "to" and "too" by "2", "you" by "U", and "for" by "4".

6.10 **LunarLander:** Design a Lander class of objects with (a) a 3-parameter constructor: its initial height in feet above the lunar surface, its initial rate of fall in feet/sec, and its gravitational pull (which for the moon is about 5 feet/sec change in velocity for each second of fall, but allow for the possibility that this program will be used for various planets too); (b) an action method with double parameter x that updates the current height and rate of fall assuming x is the number of G's of thrust applied over a 1-second period since the last update (1G is a 32 feet/sec decrease in the rate of fall relative to what it would otherwise be due to gravity alone); (c) two query methods that return the current height and rate of fall.

Next, write a main method that creates a Lander with random initial values and repeatedly gets thrust requests from the user (pilot) once for each 1-second period, making the appropriate changes and reporting the status, until the Lander hits the ground. Report the speed at which it hit. Allow many re-plays with the same initial values until the user wants to change to different initial values. After you learn to use graphics in Chapter Eight, add a graphical display of the current position of the lander.

Extra credit: Write a different main program that finds the best solution by repeated simulation, as follows: It applies no thrust for T seconds, then applies exactly 1G of thrust until the the Lander hits the ground or its velocity is reduced to zero. Have it try integer values of T from 1 on up until it finds the first value that has the Lander hit the ground, then repeatedly subtract 0.1 from T until it finds the first value that reduces the velocity to zero just before it hits. Print this best solution for any given inputs.

7.1 **HighLowPay:** Write a program that reads in a list of Worker data from a file. It then prints out the name and pay of the worker who has the highest pay, the second highest, the lowest, and the second lowest. In the case of ties for pay, it prints the one whose name comes alphabetically earlier. The program also prints the average pay per worker.

7.2 **FilledArrayOp**: Revise the WorkerList methods in Listing 7.9 and in Exercises 7.25-7.28 to be the filled-array-parameter analog as described in Section 7.6.

7.3 **FederalTax &:** Write a program that reads in a list of Worker data from a file into a WorkerList. Then it prints a report giving the name, gross pay, federal income tax, and net pay for each Worker. Calculate the federal income tax using the following approximation of the tax rate on singles: Multiply the number of exemptions by $2,800 and add $4,400 for the standard deduction. This amount is tax free. The next $27,000 that the worker earns is taxed at 14%, and all the rest is taxed at 27%.

Add an instance variable to each Worker object to record the number of exemptions he/she is entitled to. Also, the dollar amounts above are per year, so divide them by 52.2, the number of weeks in a year. Extra credit: Research and use the correct dollar amounts for the current calendar year. Also allow for married tax rates.

7.4 **Mastermind:** Revise the Mastermind methods in Listing 4.9 and 4.10 to allow the user to choose a game with two to seven digits. Have the constructor get the number 2 to 7. Then create an array for the secret digits and an array for the user's digits. Do not make any assumptions about the input actually being digits.

7.5 **Mancala:** Search the Internet to find the rules for the game of Mancala. Then write the program as a subclass of BasicGame, using one or more arrays.

7.6 **MorseCode:** Write an object class with two instance methods: One has a String parameter that should be an ordinary word and returns the String in which each letter has been replaced by its Morse code equivalent. The other method has a String parameter in which it is to replace each Morse code sequence (hypens and dots ending in a blank) by the corresponding capital letter. Use an array of size 26 to store the codes. Write an application program that accepts a line of input and converts it to Morse code.

7.7 **Cypher &:** Create a Code class of objects and a program that maintains an array of several Code objects. The program accepts as input several lines, each containing a single integer followed by a string of letters and blanks. The first character of the String is either E (for encode) or D (for decode). For an E, the program is to output the encoded form of the rest of the String. For a D, the program is to output the decoded form of the rest of the String. Each Code object represents a simple substitution cypher, in which each of the 26 letters is replaced by the corresponding character specified by the code. Each Code object has an ID number; the integer input tells the ID number of the Code which is to be used.

Example: If Code #1 is to add 2 to the Unicode value of each character, then "1E the quick brown fox" causes an output of "vjg swkem dtqyp hqz", and "1D uewbbz ftkxg" causes an output of "scuzzy drive" (x, y, and z encode into z, a, and b respectively -- you subtract 26 if adding the number takes it past z or Z). The object representing Code #1 should have a String instance variable with the value "cdefghijklmnopqrstuvwxyzab", giving the encoded form of the alphabet.

7.8 **FormLetter &:** Write a program that reads in a plain text file containing what should be a form letter. Any use of `\Name` in the letter should be replaced by a user-supplied value, and similarly for `\Address` and `\Date`. If the form letter is supposed to contain the backslash character itself, it uses `\\` to indicate one instance of the backslash to be printed. The form letter with the substitutions should be printed.

Extra credit: First read another file that supplies a list of substitutions to be made. For instance, if the second file contained the following, the program would produce two corresponding form letters:

```
\Name       Ralph Kramden
\Address    5 Main Street
\Date       January 5, 2003
\Name       Walter Mitty
\Address    27 Milquetoast Avenue
\Date       January 7, 2003
```

7.9 **Networks:** Write the Node and Position classes to go with the Network class described in Section 7.8. Then add a Network constructor to read data from a file named by a String parameter, with each Node's name and connections given on one line.

Extra credit: Add an Edge class with an instance variable of Node type, which stores the Node that it connects to. The two-dimensional array should be an array of Edges, not Nodes. An Edge should have three instance methods: `getNode`, `getMark`, and `setMark` (the latter two analogous to those for Nodes). The `getNext` method for Positions should contain one statement: return `itsList[itsPos].getNode()`. A Position should have an additional method `getEdge` that returns `itsList[itsPos]`.

7.10 **NongraphicalVics**: Complete the development of a non-graphical simulator for Vics begun in Section 7.9. So that a user can see what the status is at any point in a program, have the constructor and the `putCD` and `takeCD` methods print to System.out a description of the complete current status of the executor, including the status of the stack.

7.11 **Sets**: Implement a class for which each instance models a mathematical set of objects. This class has some similarities to the Queue class. The Set class should have the following instance methods:

```
public boolean add (Object ob); // add ob only if no Object that
     // equals it is already in the Set; tell whether the Set changed.
public boolean remove (Object ob); // remove the Object that equals
     // ob, but only if it is there; tell whether the Set changed.
public boolean contains (Object ob);   // tell whether some Object in
     // the Set equals ob
public Set union (Set par);   // return the union of the two sets
public Set intersection (Set par);   // return their intersection
public Set difference (Set par);   // return the Set of all Objects
     // that are in the executor but not in par
```

Also implement `size()`, `isEmpty()`, `equals(Set)`, and `toString()` with the normal meanings, plus a constructor to make an empty Set and a constructor to make a copy of a Set parameter.

7.12 **WeatherChannel:** The Weather Channel has hired your software development group to create software that accepts from the keyboard a sequence of temperature readings. These readings are taken at noon, 6pm, midnight, and 6am each day. You may have thousands of temperature readings, but you will not have more than 200 different temperature values. The software must, at any time during data entry when requested, (a) display the different temperatures in ascending order, with the number of occurrences of each, (b) tell the largest temperature seen so far, (c) tell whether a specified temperature has occurred before, or (d) tell the temperature value that has occurred most often (in case of ties, tell the largest such value).

Write this program. Use a DataSet class with two instance variables: `private int itsSize; private Info[ ] itsItem` for a partially-filled array. Have a separate DataSet instance method to perform each of the operations described in (a)-(d). Use a nested class of objects, defined inside the DataSet class as follows:

```
private static class Info
{  public int itsTemp;
   public int itsCount = 1;
   public Info (int par)
   {  itsTemp = par;
   }
}
```

Restriction: Design the DataSet class as a partially-filled array so that, if the following method were part of that DataSet class, it would remove all temperature values less than the given temperature value:

```
public void erase (int cutoff)
{  int keep = 0;
   while (keep < itsSize && itsItem[keep].itsTemp < cutoff)
      keep++;
   if (keep > 0)
   {  for (int k = 0;  k < itsSize - keep;  k++)
         itsItem[k] = itsItem[k + keep];
   }
   itsSize -= keep;
}
```

Extra credit: Revise the main logic so that the user can handle ten DataSet objects instead of just one. The user can specify a number 1..10 to indicate which DataSet is being handled by all future requests to add, display, etc., until the user specifies an new number 1..10. Use an array of 10 DataSets.

7.13 **OneDimensionalRandomWalk:** (Read the earlier RandomWalk project description for Chapter Four to get the idea) One trial consists of a person walking 200 steps. Each step is either east or west, with equal probability. Record the final location of the person in an array. Repeat for 1000 trials. Print out the number of persons at each possible location, except ignore those more than 25 steps away from the starting point. Discuss the relation of the numbers you get to Pascal's Triangle.

7.14 **TwoDimensionalRandomWalk:** Same as the previous problem, but each step is either north or south or east or west. Record the final location of the person in a 2-dimensional array, ignoring those people who are more than 25 steps away from the starting point. Instead of printing out the whole 51-by-51 array, print the number of people N total steps away from the starting point for each value of N from 0 to 25. Extra credit: Do it for a 3-dimensional array (think of it as representing fish in a fish tank).

7.15 **FlashCards:** Write a program that reads in a text file with two words on each line, the first in English and the second in Spanish (or whatever other language you prefer). This program allows the user to practice vocabulary items in a foreign language. As the pairs of words are read in, store each pair in a Vocabulary object, which you then put in a WordList object, a partially-filled array of Vocabulary items that you always keep in alphabetical order of English words, regardless of their order in the file. However, whenever one pair has the same English word as an earlier pair, replace the earlier pair with the later one. Restriction: Vocabulary objects should be mentioned only within the WordList class.

After reading to the end of the file, the main method lets the user repeatedly indicate one of the following choices until EXIT is chosen (you may use either letters or digits to indicate the choices):
(a) EXIT means quit the program.
(b) TO_ENGLISH means list the English words one at a time, waiting after each one until the user presses ENTER, then display the corresponding Spanish word. However, if the user presses CANCEL in response to a display of the Spanish word, remove that Vocabulary item permanently from the list.
(c) FROM_ENGLISH is the opposite of (b): list Spanish words, then display the corresponding English.
(d) TRANSLATE means accept one English word from the user and display its translation.

If you have studied buttons and textfields, use a button for the three options (a), (b), (c) and a textfield to enter the English word for option (d).

7.16 **CourseSections:** Write a CourseSection class as follows: It has only these instance methods: `getMales()`, `getFemales()`, `getName()`, `toString()`, `changeMales(int)`, and `changeFemales(int)`; all instance variables are private. The first four methods listed simply return a value. The two change methods do nothing if the parameter is not in the range -2 to +2 inclusive; otherwise they change the number of people registered by the amount specified. The one CourseSection constructor should have one parameter, which is one line from the file.

The text file `data.txt` has a group of 1 to 20 lines of the form <males females sectionName> each representing one CourseSection object (all sectionNames different; males and females are ints; sectionName is a String with no blanks; one space separation between parts). The text file `changes.txt` has a group of an unknown number of lines of the form <change sectionName> (change is one of "+m", "-m", "+f", or "-f", meaning "add 1 male", "remove 1 male", "add 1 female", "remove 1 female"; sectionName is one of those that appeared in `data.txt`).

Write an application that reads in the `data.txt` file, stores the information in an array, makes all the changes specified by the `changes.txt` file to the CourseSection objects, then prints representations of the revised CourseSection objects.

Sample data.txt file	Sample changes.txt file	Output for these files
17 4 CS152-71	+f CS151-02	17 4 CS152-71
12 13 CS151-02	-m CS253-01	13 14 CS151-02
5 15 CS253-01	-f CS253-01	4 14 CS253-01
	+m CS151-02	

Avoid all possible crashes, for bad input or otherwise, treating contrary-to-spec input lines as if those lines were not there. Use a second class of objects CourseSectionList for all list operations.

7.17 **PayWorkers &:** Write a program that reads in a file describing workers and also a file giving the five numbers for each worker that tell the number of hours worked each day. Accept keyboard input that updates these work hours for the next day. Produce a new file giving the updated five numbers (use redirection to create the new file).

7.18 **WorkersNames:** Some workers may have the same name. Write a program that reads in a file describing workers and then prints out each name that occurs more than once. Make sure you only print a name once, even if it occurs three or four times.

7.19 **EvilCarRentals &:** A car rental company charges its customers a $150 penalty if they go over 80 mph in the rental car and $100 if they go outside the designated driving area. Write a program that takes input from a text file (which is actually a remote terminal connected to GPS) for which each line gives the time and <x,y> position of one of up to ten rental cars (three double values, with positional coordinates measured in feet from an arbitrary reference point) as well as its VIN (a String). Each time a penalty is incurred, write out the time and nature of the offense and its VIN. Design an appropriate Car class of objects. The designated driving area is specified as two opposite corners of a rectangular area. Extra credit: The designated driving area is a quadrilateral.

7.20 **SieveOfErosthenes:** Create a class of objects as follows: The constructor has a positive int parameter that tells it how large an array to construct; initially each component `a[k]` contains the value k (you will ignore the first two components throughout). One method finds the lowest value not previously established to be a prime, notes that it is now known to be a prime, and overwrites in the array each multiple of that prime with zero. Another method tells whether all non-zero values in the array are known to be prime. Another method removes all zeros from the array by moving non-zeros towards the front of the array. And another method reports the number of primes found so far.

Then write a main method that calculates the percentage of primes in the first N integers for N having the values 1000, 2000, 3000, 4000, and 5000.

7.21 **Supermarket &:** A supermarket executive asks you to decide whether changing one or more checkout lanes to 10-item-or-less lanes will improve customer satisfaction. You have data describing a typical store day's customer demand, in the form of a text file with 2 numbers per line. Each pair represents one customer: the arrival time at a checkout lane (measured in seconds since opening time) and the number of items purchased. The file ends with the pair 0 0 as a sentinel. History indicates that checkout processing requires an average of 5 seconds per item plus 30 seconds of overhead (for collecting the money, etc.). The executive specifies that the "dissatisfaction rating" of a particular choice of express lines is to be the amount of time spent waiting in line, averaged over all customers. [problem taken from Pascal: Problem-Solving and Programming With Style, by Dr. William C. Jones, Jr., copyright 1986 by Harper & Row.]

7.22 **SalesReport &:** A telemarketing firm wants us to create a sales-report program to process the monthly sales figures of each salesperson. The data available is a list of salesperson names, sales figures for the month, and hours spent on the phone in that month. The program is to compute the median sales figure and list the name of the salesperson with the highest sales per hour (so he/she can be given a bonus) and the name of the sales-person with the lowest sales per hour (so he/she can be fired). However, the bonus requires being more than 20% above the median and working at least 100 hours in the month. And a salesperson is fired only if he/she was at least (a) 20% below the median if working at most 50 hours in the month, (b) 25% below the median if working at least 150 hours in the month, (c) on a sliding scale between 20% and 25% for working between 50 and 150 hours in the month.

7.23 **Riffle:** Define a class of CardDeck objects, each representing a single deck of n cards for some positive int value n. Include a constructor with n as a parameter, creating a deck with the n cards in increasing order (i.e., card #0 at index 0, card #1 at index 1, etc.). Add the following methods: `deck.riffle()` performs a perfect riffle shuffle (the deck is divided in half, with the top half larger if it has an odd number of cards, and then put in the order first-card-of-top-half, first-card-of-bottom-half, second-of-top, second-of-bottom, third-of-top, etc.). `deck.riffleCount()` calls `deck.riffle()` as many times as necessary to get the deck back to its initial order and returns that number of times. `deck.howMixedUp()` tells what percentage of "unsortedness" the deck has (calculate for each card the distance from where that card would be if the deck were in ascending order and add up these numbers; if M is the (maximum) value of this total (for descending order), then a deck with a total of M or 0 is 0% unsorted and a deck with a total of M/2 is 100% unsorted). Add two more similar interesting CardDeck methods.

7.24 **PokerHands:** Write a method that accepts an array of five Card objects and then returns the kind of Poker hand it is: straight flush, four of a kind, full house, flush, straight, three of a kind, two pairs, a pair, or no-combination. A Card should be a pair of int values, one for the rank and one for the suit from a standard 52-card deck. Your method should use an array of 13 components that count the number of cards in each rank. So a full house is a hand where one count is 3 and another is 2. Finally, write a program that uses this method in some realistic way.

7.25 **ArrayLists:** Rewrite all the programs in Chapter Seven using ArrayLists as described at the end of Chapter Seven (never use array brackets).

7.26 **ChristmasBeer:** Write a program that prints the words to "The 12 days of Christmas" or to "99 bottles of beer on the wall", whichever the user chooses. Use the obvious array of twelve String values for the Christmas song.

8.1 **GradeHistogram**: Develop a class with the following methods:

```
   public GradeHistogram ();  // constructor; set all values to 0
   public void setGrades (String grades);
      // count how many of each of A,B,C,D,F appears in grades
   public char mode();     // query which of the capital letters
                  // A,B,C,D,F has the highest frequency
   public void showHistogram (Graphic page, int xcor, int ycor);
      // display histogram for the counts of the five characters
      // A,B,C,D,F with (xcor, ycor) as its bottom-left corner
```

Test with an applet or frame having this coding for the `paint()` method:

```
   public void paint (Graphics page)
   {  GradeHistogram cs1 = new GradeHistogram();
      cs1.setGrades ("ABC ABCD ABCBc  eABCF");
      System.out.println ("The mode is " + cs1.mode());
      cs1.showHistogram (page, 50, 110);
      cs1.showHistogram (page, 180, 120);
      cs1.setGrades ("AAAFFFFCCBBB");
      cs1.showHistogram (page, 50, 300);
   }
```

The histogram itself lists the five capital letters horizontally. Above each letter is a column of N rectangles, where N is the number of times that letter appeared in the most recent grades parameter. Each rectangle is 5 pixels high and 10 pixels wide.

8.2 **SlotMachine &:** Write an applet or frame that simulates a slot machine with its spinning wheels. It should have three windows with ten possible symbols for each window. Pay off for getting all three the same, paying the most if all three are stars. Choose minor payoffs for matching two symbols. Give the house an overall profit of between 2% and 5% (pay by credit card). Use animations and random numbers.

8.3 **DrawClock:** Write an applet or frame that reads in a time such as 3:15 or 10:45 using `showInputDialog`, then draws a clock with a big hand and little hand pointing to the right time. A minimal drawing is a circle plus two lines of different length extending from the center of the circle. Alternative problem, for teaching young children to tell time: Repeatedly choose a clock time at random and draw the clock that shows that time. Ask the young user to say what time it shows, then say whether it is right (within three minutes of the time that is indicated).

8.4 **ScissorsPaperRock:** Complete the applet for colliding balls described in Section 8.8: Define three subclasses of the Ball class with different `drawImage` methods, different `hitsOther` methods, and running counts of each kind of object. Extra credit: Have a dying Ball slowly drift down to the bottom as it deteriorates, and be reborn when it gets there. Or add a button or slider to affect the movements.

8.5 **MazeDrawer:** Write a program that draws a solvable maze. If e.g. you choose to create a 40-by-30 maze, you could proceed as follows: Think of a rectangular area as being divided into 30 rows of 40 small squares. Note that each of the 1200 small squares has 4 walls, most of which are shared between two squares, so there are somewhat over 2400 walls that could be drawn in. First draw a meandering path from the lower-left corner to the upper-right corner. Then repeatedly choose a wall to fill in that does not block the path you chose. Fill in about 40% of the possible walls. Then print the maze, with a special marker on the starting (lower-left) and ending (upper-right) corners. Allow the user to print the maze both without the path marked (the challenge) and with the path marked (the solution).

8.6 **Yahtzee &:** Design and implement a program that supervises 2 to 4 human players of Yahtzee, keeping records as needed.

8.7 **Elevators &:** Design an implement a program that simulates the movements of 3 elevators in a 20-story building. Be sure to have Elevator objects (each knows the floor it is on and the direction it is moving) and Person objects (each knows the floor it is on and the floor it wants to go to). Assume that one time unit is the time to move an elevator up or down one floor or half the time an elevator spends loading and/or unloading at one floor. Assume for simplicity that elevators never become full. Include a description of a reasonably efficient processing for deciding which elevators go where, in such a way that no person waits for more than 100 time units to get an elevator. Have Person objects appear at the elevator landings at random, on average once each 5 time units.

9.1 **Speculator &:** Modify the Investor software to have assets follow trends. One day's multiplier is to be 70% random and 30% calculated from the trend for the previous few days. Research how to calculate a 9-day Exponential Moving Average and use that for the trend. The description of an asset should give the current trend number on a scale from 0 to 10, 10 being rapidly rising and 0 being rapidly falling.

9.2 **StockRebalancer:** Modify the Investor software as follows: Write a program that accepts from the user five percentages -- non-negative integers that add up to 100. Create a Portfolio that has these percentages in each of the five asset classes. Also ask for an initial balance, a monthly deposit, and a number of years. Example: If the input is 10, 20, 40, 15, 15, the user will have 10% in the money market, 20% in 2-year bonds, 40% in large-cap stock, and 15% in each of small-cap stock and emerging markets. If the user also specifies $80,000, -300, and 20, that means the user starts with a balance of $80,000, spends $300 a month from the account, and this continues for 20 years.

Run a simulation for that many years with that monthly deposit, rebalancing the total assets among the five asset classes to the original proportions once each month. In the example, you would start with $80,000 and, at the end of each month for 240 months, subtract $300 and then shift assets so that you have 10% in the money market, 20% in 2-year bonds, etc. Repeat the simulation 99 times and list (a) the highest outcome, (b) the lowest outcome, and (c) the average outcome (geometric average). The purpose of the exercise is to show the user what the range of outcomes would be if the user were to use this "constant-ratio asset allocation" method, ignoring any trending in the markets.

Extra credit: Keep track of all 99 results in an array of values. Each time you add a value, insert it in increasing order. After all 99 repetitions have been made, list the first, tenth, fiftieth, ninetieth, and ninety-ninth values. This gives the user an even better idea of the possible outcomes.

10.1 **Scrolling Marquee:** Write a program to create a full-screen frame with a scrolling marquee. Initially it says "Hello World!". Have a textfield in the bottom-right corner where the user can enter a new phrase to be scrolled. Alternative: Make an applet.

First display the given String of characters at the far right of the screen at a randomly chosen height. Most of the characters will not show because they are off the screen to the right. Then move the String a few pixels to the left and repaint. Repeat until the String disappears off to the left. Then choose another random height and repeat. Extra credit: Keep changing the font shape and size.

10.2 **GraphicalGames:** Write a game-playing program with graphical user interface: Display a main frame that allows the user to choose one of the three games GuessNumber, Nim, or Mastermind described in Chapter Four. Revise each of these games to have input from a textfield or a slider that stays on the screen through the game rather than using JOptionPane.

10.3 **AddressBook &:** Write a program with a graphical user interface that allows the user to create, update, and search a personal address book (names, phone numbers, email, addresses, birthdays), including the option of listing certain subsets in order.

10.4 **ShoppingCart &:** Write an applet or application program that behaves much as the shopping cart applets for online shopping. Read a file to get the information about products. The main screen should have at least three categories of products to click on and display the total money due so far. Each category should have from 3 to 10 items in it that a buyer can choose.

10.5 **Solitaire:** Use a frame and graphical components for this program. The program shuffles a standard 52-card deck and deals it in 4 rows of 13 cards. The player repeatedly clicks on two adjacent cards of the same suit or of the same rank (the last card on one row is considered adjacent to the first card on the next row below). If the two cards clicked meet this criterion, the program removes those two and moves the rest closer together to fill up the gap. Play stops when no legal move is left. The score is the number of cards removed, so the highest score possible is 52.

Card objects should have two characters for instance variables: the rank from the 13 char values {A,2,3,4,5,6,7,8,9,T,J,K,Q} and the suit from the four char values {C,D,H,S}. A random shuffle of the cards can be obtained as follows:
for each int value k from 0 to 51 inclusive do...
{ r = a random int from k through 51; swap the card at index r with the card at index k;}.

10.6 **PatiencePoker:** This is similar to the preceding solitaire game but more challenging and interesting. The program shuffles the standard 52-card deck and shows 25 cards, one card at a time. After each card is shown, the player must put it in a 5-by-5 grid before seeing the next card. When all 25 cards have been placed (one card in each of the 25 parts of the grid), the program announces the player's score, which is the sum of the scores for the ten poker hands that show (5 in the horizontal rows and 5 in the vertical columns). The score is 2 for a pair, 5 for two pairs, and 10/15/20/25/50/75 for three-of-a-kind, straight, flush, full house, four-of-a-kind, and straight-flush, respectively. Hint: You can most easily tell what a 5-card hand counts as if you first count the number of cards that have the same rank as a card later in the hand: 1 means a pair, 2 means two-pair, 3 means three-of-a-kind, etc.

10.7 **VendingMachine &:** Design software to control a candy vending machine. It will need a Selection object (responds to the customer pressing a button to choose one particular product bin), a CoinSlot object (responds to the customer entering a particular denomination of coin or bill), a ReturnMoney object (responds to the customer pressing the `returnMyMoney` button), a MoneyCounter object (tracks the amount entered and the coins available to make change), and a list of Bin objects (one per kind of product; tracks the price and availability of the product). Allow for refills and service by the manager.

10.8 **Bingo &:** Write a program that lets two players play Bingo. Each gets two different cards, chosen from a store of ten cards. Call letter/number pairs (a different random letter/number pair each time) until one of the cards wins (5 in a line, any of 12 possible). Keep track of total winnings. Include Card, LetterNumber, and Player objects.

Additional problems for Chapter Ten: Most of the problems specified for Chapter Seven can be written using a Graphical User Interface instead of what they describe.

11.1 **UltraLong**: Write an UltraLong subclass of Numeric, like VeryLong except `itsItem` is an array that can have any positive number of components. Adding two values requires allowing for one having a longer array than the other and never produces the wrong answer just because the result is too large. Store the most significant part of the number in the highest-indexed component, not in component zero. That highest-indexed component is to have a positive int value when the array has more than one component.

The `valueOf` method should accept any String of digits and commas that begins with a digit. Have a private constructor with one int parameter giving the size of array to make, and a private method: `result.trim()` replaces the executor's array by a shorter one if it has leading zeros. You may disable the `multiply` method. Extra credit: Include the `multiply` method. More extra credit: Add a method to divide by a given int value. Use the following two constructors for this class. The first gives the UltraLong equivalent of `digits * 10`^{expo} and the second is for private use in constructing return values:

```
public UltraLong (int digits, int expo)
{  super();
   if (digits <= 0 || expo < 0)
      itsItem = new int [1];  // contains zero
   else
   {  long num = digits;
      for (int i = expo % 9;  i > 0;  i--)
         num *= 10;

      if (num < LONGBILL)
      {  itsItem = new int [expo / 9 + 1];  // all zeros
         itsItem[itsItem.length - 1] = (int) num;
      }
      else
      {  itsItem = new int [expo / 9 + 2];  // all zeros
         itsItem[itsItem.length -1] = (int) (num / LONGBILL);
         itsItem[itsItem.length -2] = (int) (num % LONGBILL);
      }
   }
}
private UltraLong (int arraySize)
{  super();
   itsItem = new int [arraySize];
}
```

11.2 **Multiplication:** Write an application program that tests out all the services offered by the VeryLong class. Complete all methods in the VeryLong class (the hard one is the multiply method). Extra credit: Add a method to find the square root of a VeryLong value. More extra credit: Allow negative numbers (have an extra instance variable that tells whether the value is negative).

11.3 **PolarCoordinates:** Add three methods to the Complex class: a class method `fromPolar(r,t)` returns the Complex object whose polar coordinates are radius r and angle t; instance methods `getRadius()` and `getAngle()` return those two attributes.

11.4 **AlgebraicIntegers:** Implement a new subclass of Numeric: An AlgInt object represents a number of the form a + b*s where a and b are integers and s is the square root of 3. Extra credit: Add a method that tells whether a given AlgInt value is a prime number in the sense of having exactly two positive AlgInt divisors. Note that 13 is not an AlgInt prime because 13 = (4 + s) * (4 - s).

11.5 **Polynomials:** Implement a new subclass of Numeric: A Polynomial object represents a polynomial with double coefficients and no term of higher degree than 9. Have an array of length ten as an instance variable. A polynomial is to be considered positive when its leading coefficient is positive. Extra credit: add methods to evaluate the polynomial for a given x-value and to find the derivative of the polynomial.

11.6 **BooleanAlgebra:** Implement a new subclass of Numeric: A Group object represents a finite set of positive integers. The sum, difference, and product of two Groups corresponds to set union, set subtraction, and set intersection. Hint: Store the finite set of positive integers as an array of positive ints in increasing order.

12.1 **ExamGrades &:** Write a program to handle exam grades for one group of students. Maintain a permanent file of Student data where each Student object includes five exam grades (integers). Also maintain a permanent file of Exam data where each of five Exam objects includes the correct answers to the exam, the number of people who got each question right (an array of non-negative integer values), and the highest and lowest grades on the exam.

From time to time, read in a file containing student answers to a single exam, one line per student. Use this data to update the Student and Exam files. Use appropriate methods in the Student and Exam classes. Write an application program that would be useful to a teacher keeping track of student grades for the semester.

12.2 **RandomAccess:** The company that wants email tracked as described in Chapter Twelve has a thousand employees. For each possibility of one employee sending email to another, the company wants you to store the subject lines of the five most recent emails for analysis (up to 80 characters per subject line). Implement the EmailSet class using a random-access file, since this amount of data is beyond the RAM capacity of the computer being used.

12.3 **RemoveComments:** Write a method that reads in a text file that is contains a compilable Java class and prints a new file consisting of everything but the comments in that class. That is, strip out // and everything after it on the same line, as well as /* and everything after it up through the next */. However, ignore //, /*, and */ that are inside of quote marks.

12.4 **Concordance:** Write a program that reads in a text file and produces a concordance. This is a listing of many lines of text, one for each occurrence of a principal word in the original text file. Each line contains the principal word plus the three words that come before it and the three words that come after it in the original text. The listing is in alphabetical order of principal words. The listings for any one principal word are in the order they occur in the original text. For example, if you define a principal word to be any word of six or more letters, a concordance of the first three sentences of this paragraph is as follows:

```
words that come before it and the
and produces a concordance.  This is a
file.  Each line contains the principal word
This is a listing of many lines
one for each occurrence of a principal
word in the original text file.  Each
it in the   original text.  The listing
occurrence of a principal word in the
line contains the principal word plus the
text file and produces a concordance.  This
Write a program that reads in
```

Define a principal word to be any word of N or more letters, where N is obtained from the command line (with a default of six). For a text file of any substantial size, this concordance can become quite large. It would be quite inefficient to keep inserting new lines in an array of tens of thousands of lines, keeping them all in order. So you are to use an array of 26 LineList objects, one LineList object for each letter of the alphabet. The component numbered 0 will contain the principal words beginning with a, the component numbered 1 will contain the b's (the first line in the example above), the component numbered 2 will contain the c's (the second and third lines in the example above), etc. Each LineList object is to be an array of objects that each have two instance variables: one is to be the String of three words that come before the principal word, and the other is to be the String of the remaining four words beginning with the principal word. This makes it easy to test the ordering and keep the array in increasing order.

Extra credit: Allow the user to specify that a principal word is any word of N or more letters, plus any word on a given list of words, minus any word on a second given list of words (all of these specifications are obtained from a separate input file). Also, allow the user to specify that the principal word be boldfaced or that each listing be in inverted order, illustrated by the following:

```
before it and the             -- words that come
concordance.  This is a       -- and produces a
contains the principal word   -- file.  Each line
```

Extra credit #2: Do not include any words from another sentence (a sentence ends with a period or exclamation mark or question mark or colon). If that shortens the phrase on one end, use four words from the other end (unless that would go outside the current sentence).

12.5 **GameOfLife:** This software simulates a number of cycles over which various kinds of beings are born, live, and die. The most elementary kind is John Conway's Game Of Life with two kinds of Beings, x's and empty spots. It is convenient to have an empty spot considered to have an empty Being there named Being.EMPTY_ONE. More advanced simulations have tigers eating deer which eat grass, three kinds of Beings besides empty spots, so Being.NUM_CATEGORIES is 4 instead of 2. Each kind of Being has a different ID number, from 0 up to Being.NUM_CATEGORIES, which can be found by calling `someBeing.getID()`.

The general idea is that we have a "world" consisting of a rectangular array of Being objects. Initially, each cell of the array is populated by a random Being, except that all cells along the border are empty. At the beginning of each small time period, each cell in the rectangular array changes to a new kind of Being or stays the same, depending on the kind and number of various beings in the eight cells adjacent to it at the end of the previous time period. You then display the state of the world in a sequence of time cycles. First you need a class of World objects. The following coding is complete except that you have to fill in the part marked with a comment. Your assignment is to write the Being class and its two subclasses (these are the basic Game Of Life rules): Empty beings become Cross beings if they have exactly 3 Cross beings as neighbors, otherwise they remain Empty. Cross beings become Empty beings if they do not have either 2 or 3 neighbors, otherwise they remain Cross. Extra credit: Invent more interesting rules for three or four categories of Beings. Search the web for Thomas B. Schelling's work.

```
public class World
{
   private Being[][] itsCurrentState;
   private Being[][] itsFutureState;
   private final int[] itsCount;
   private final int itsSize;

   public World (int dimension)
   {  super();
      itsSize = dimension > 2  ?  dimension  :  20;
      itsCurrentState = new Being [itsSize + 2][itsSize + 2];
      itsFutureState = new Being [itsSize + 2][itsSize + 2];
      // add coding to put Being.EMPTY_ONE in each border cell
      for (int row = 1;  row <= itsSize;  row++)
      {  for (int col = 1;  col <= itsSize;  col++)
            itsCurrentState[row][col] = Being.random();
      }
      itsCount = new int [Being.NUM_CATEGORIES];
   }

   public void liveOneCycle()
   {  for (int row = 1;  row <= itsSize;  row++)
      {  for (int col = 1;  col <= itsSize;  col++)
         {  countNeighbors (row, col);
            itsFutureState[row][col]
                  = itsCurrentState[row][col].next (itsCount);
         }
      }
      Being[][] save = itsFutureState;
      itsFutureState = itsCurrentState;
      itsCurrentState = save;
   }

   private void countNeighbors (int row, int col)
   {  for (int k = 0;  k < itsCount.length;  k++)
         itsCount[k] = 0;
      for (int i = row - 1;  i <= row + 1;  i++)
      {  for (int j = col - 1;  j <= col + 1;  j++)
            itsCount[itsCurrentState[i][j].getID()]++;
      }
   }
}
```

12.6 **VignereCypher:** Write a class with a constructor that has a String parameter plus two instance methods to encode and decode a text file using the String parameter as the key. Encoding consists of converting each letter in the input to an uppercase letter and each non-letter to the value `(char)('A'-1)`. This gives one of the 27 integers from 64 to 90 inclusive. If `itsCode` denotes the key, you add the first character of `itsCode` to the first converted character of the input file to get the first character of the output file. In general, you add the nth character of `itsCode` to the nth converted character of the input file to get the nth character of the output file, where (a) if the sum is at least 27, subtract 27, and (b) when you come to the end of `itsCode`, you start over with the first character of `itsCode`. This is called the Vignere Cypher method; it is quite hard to decode if `itsCode` is a long word or phrase.

12.7 **Matrices:** Write a class where each object represents a single matrix (a positive number of rows and a possibly different positive number of columns). Have instance methods `x.add(y)` to return the sum (where possible) and `x.multiply(y)` to return the product (where possible), as well as at least three other interesting methods of your choosing.

13.1 **TimingSorts:** Write a program that chooses 3000 Double values at random, sorts them by each of the `selectionSort` and the `insertionSort` methods, and reports the elapsed time. Call a method to do this 20 times and give the average of the 20 for each sorting method. Then call the method again for 6,000 values, then again for 12,000 values. Write an explanation of how your results confirm or contradict the expected big-oh behavior of these sorting algorithms. Extra credit: Include the Merge Sort algorithm.

13.2 **DoubleSelectionSort:** Develop logic for finding both the maximum and minimum of 2*m values in an array in only 3*m-2 comparisons: Each pair of values is compared with each other; then the smaller of the two is compared with the `smallestSoFar` and the larger of the two compared with the `largestSoFar`; then both swaps are made. Then use this logic as a basis for a method of sorting N values that makes less than $3N^2/8$ comparisons.

13.3 **QuickSelect:** Add to Listing 13.4 a method `public Comparable atIndex (int index)` to find the value that would be at a given index if the array of values were sorted, using an algorithm that is big-oh of N on average.

13.4 **RiseAndFall:** Implement the "rise-and-fall" logic for the MergeSort described at the end of the section on the MergeSort.

13.5 **MostlySorted:** Modify the MergeSort logic so that it executes much faster when the elements in the array are nearly in order already. Specifically, you always check before merging whether the elements are already in order. If so (the highest element in the lower group is less than or equal to the lowest element in the upper group), then do not do any merging. Have the private sorting methods return a boolean value: true if they actually merged, false if they did not move the groups as they were supposed to (as thus the group is left in the wrong array). Adjust the logic for this returned value.

13.6 **Quicksplit:** Rewrite the coding for the `split` method in `quickSort` as follows: After removing the pivot from `item[lo]`, keep track of two index positions `small` and `big` such that `item[lo]...item[small]` are all less-equal the pivot and `item[small+2]...item[big]` are all larger-equal the pivot, with `item[small+1]` remaining empty (so you can put the pivot there when done). Keep incrementing `big` until it becomes `hi`, at which point you can move the pivot to where it goes and have the desired split. Discuss whether this executes slower or faster than the given coding.

13.7 **QuickSortWithSpare**: Rewrite the QuickSort logic to execute faster by using a second spare array (similar to MergeSort's use of a spare array).

14.1 **StockTransactions &:** Write out the full program for the buying and selling of company stock described in Section 14.1.

14.2 **MatchingParentheses:** Write a program that reads in several lines of input and for each one, tells whether its grouping symbols are nested properly, as described in Section 14.1. Ignore everything on a line except for `( ) [ ] { }`. Store these symbols on a stack until matched. Extra credit: Ignore everything that is inside a pair of quotes and everything that comes after `//` on the line.

14.3 **LazyQueue:** Write a complete implementation of QueueADT using the "avoid-all-moves" approach described in Section 14.2.

14.4 **DoubleEndedQueue:** Write a complete array implementation of a class that allows the queue operations plus adding to the front of the list and deleting from the rear of the list. Use the concept shown in Listing 14.3. However, start with `itsFront` initially `itsItem.length / 2` so you can add to the array in either direction. When you get to one end, shift all the values back to the middle of the array (unless the array is full).

14.5 **DMV &:** The Department of Motor Vehicles in a large city has 9 clerks at 9 windows, one for each three letters in the alphabet ('A'-'C', 'D'-'F', 'G'-'I', etc.). Each customer who comes in signs up by last name at a computer terminal. Each time a clerk finishes with a customer, the clerk presses a button, which causes this computer to display the next customer's last name on a screen above the clerk (that is, the next customer whose first letter of the last name matches the clerk). Write the software to control the computer with a GUI interface having three panels: one for sign-in, one for 9 buttons, and one for 9 displays. Your JFrame subclass should have an array of Clerk objects, each Clerk with 3 instance variables: itsJButton, itsJLabel, and itsNodeQueue. Each JButton should have the Clerk's letters on it: "ABC", "DEF", "GHI", etc.

14.6 **QueueWithManyArrays:** Implement QueueADT analogously to the old/new linked-array implementation of StackADT described at the end of Section 14.2, in which one array contains a reference to another array. Hint: Store a link to the new array in the old array instead of vice versa, and keep track of both the front and the rear arrays.

14.7 **PostfixCalculator &:** Write a program that acts as a calculator using postfix notation. The JFrame should have 10 buttons with the 10 digits on them, a button with a decimal point on it, a button with ~ on it, a button with PUSH on it, and four binary operator buttons with +, -, *, and / on them. A label displays the digits in the currently-entered number. Pressing the PUSH key pushes the number currently on display onto a stack of numeric values. Pressing a digit button appends it to the displayed number, unless it has been pushed, in which case pressing a digit button makes that digit the first and only digit in the displayed number. Pressing the decimal point appends it to the displayed number if it does not already have a decimal point. The ~ button only has an effect when pressed as the first entry for a number; it represents a negative sign (as opposed to the - button which represents subtraction of two numbers).

Pressing a binary operator button has the push effect if the displayed number has not yet been pushed. It then pops the two top values off the stack and displays the result (the sum, difference, product, or quotient depending on the operator symbol). Finally, it pushes the result. In addition, a JTextArea shows all the values currently on the stack and a CLEAR button clears the display.

14.8 **Registration &:** Write a Register object class: A Register object keeps track of courses offered at a school and the students who sign up for them. `signUp (aCourse, aStudent)` adds `aStudent` to the NodeList of those signed up for `aCourse` and adds `aCourse` to the NodeList of the courses `aStudent` is signed up for, if room (a limit of 25 students per course and 5 courses per students). Add several other methods for maintaining registration records. Extra credit: Check time conflicts.

14.9 **Anagrams:** Write a program that accepts as input two positive integers N and n and then prints out 24 anagrams of the first N letters of the alphabet starting with the nth anagram. Use the logic illustrated in Section 14.1, in which you keep the letters of the current anagram in a stack in reverse order and move from one anagram to the next with the help of just one queue. The anagrams are numbered in alphabetical order, so for N = 5, the first is abcde, the second is abced, the third is abdce, etc. The reason for the restriction to 24 is that printing all of them would print N-factorial lines, which is 5040 when N is just 7. Add to your stack class a method that returns the `toString()` value of all the elements on the stack in reverse order. Extra credit: Write an alternative version of this program in which you keep the current anagram in a queue in reverse order and move from one anagram to the next with the help of just one stack.

14.10 **RadixSort:** Write a program that reads in a file containing one word per line and prints them out in alphabetical order, as follows: First add to NodeQueue a method `public void append (NodeQueue queue)` for which the executor adds the contents of queue after its own elements, keeping the same relative order. Read the file into a single NodeQueue. Apply the logic of the radix sort described at the end of Chapter Fourteen, one pass per character, with an array of 26 NodeQueues. Only use the first ten characters, so make ten passes that put values in the array of NodeQueues and then back into a single NodeQueue. Finally, apply an insertion sort to take care of the very few cases of two words that have the same first ten characters.

14.11 **QueueFromStacks:** Write an implementation of QueueADT where each queue object has two instance variables, each of which is a stack. To add a value, put it on the second stack. To remove a value, take it from the first stack, unless the first stack is empty, in which case shift all the values from the second stack to the first stack and then remove the value. Essay question: Can you implement StackADT using two queues?

14.12 **BankTellers:** A bank has one line that customers stand in until called. It has 3 teller windows. A new customer arrives every 1 to 3 minutes, uniformly randomly distributed. A customer requires 2 to 5 minutes for service, also uniformly randomly distributed. Estimate the average waiting time per customer, the average length of the queue where they stand, and the longest waiting time by using 1000 sample runs and averaging their results. Repeat the experiment for 1, 2, 3, 4, and 5 teller windows and put the results in a table nicely laid out so that the bank manager can decide how many teller windows are best.

15.1 **ArraySequence:** Write out the full ArraySequence and ArraySequenceIterator classes. Include a constructor with a Collection parameter. Write an application program that tests almost all of the methods.

15.2 **PolynomialLinks:** Write a Polynomial class that extends the Numeric class of Listing 11.3. Use a linked list that contains exactly one Node for each non-zero term in the polynomial. The coding on the next page gives a partially done implementation for you to complete; you have left to do `toString` (which should return something like 3.4*x^4 - 7.0*x^2 + 5.2*x -8.1), `doubleValue` (which should just return the constant coefficient), `equals`, `compareTo` (which should go by the coefficients of the highest power at which two polynomials differ), and `subtract`; omit the `multiply` method. Use recursion wherever feasible. Put in the following five additional methods:
`public double getCoef (int x)` returns the coefficient for the given exponent.
`public void deleteTerm (int x)` deletes the term with the given exponent.
`public Polynomial addTerm (double e, int x)` adds the one term (do this without calling the `add` method).
`public double evaluate (double argument)` returns the value of the Polynomial when you replace x by the parameter.
`public Polynomial derivative()` returns the derivative of the executor.

```
public class Polynomial extends Numeric // partial listing
{
private Node itsFirst;

public Polynomial()
{  itsFirst = null;
} //================================================

public Numeric valueOf (String par)
{  Polynomial answer = new Polynomial();
   answer.itsFirst = convert (new StringTokenizer (par));
   return answer;
} //================================================

private static Node convert (StringTokenizer strit)
{  try
   {  double co = Double.parseDouble (strit.getNextToken());
      int expo = Integer.parseInt (strit.getNextToken());
      return new Node (co, expo, convert (strit));
   }catch (RuntimeException e)
   {  return null;
   }
} //================================================

public Numeric add (Numeric par)
{  Polynomial sum = new Polynomial();
   sum.itsFirst = add (itsFirst, ((Polynomial) par).itsFirst);
   return sum;
} //================================================

private static Node add (Node p, Node q)
{  if (p == null && q == null)
      return null;
   else if (p == null)  // so q != null
      return new Node (q.itsCoef, q.itsExpo, add (p, q.itsNext));
   else if (q == null || p.itsExpo > q.itsExpo)
      return new Node (p.itsCoef, p.itsExpo, add (q, p.itsNext));
   else if (q.itsExpo > p.itsExpo)
      return new Node (q.itsCoef, q.itsExpo, add (p, q.itsNext));
   else if (p.itsCoef + q.itsCoef == 0)   // same exponent
      return add (p.itsNext, q.itsNext);
   else
      return new Node (p.itsCoef + q.itsCoef, p.itsExpo,
                       add (p.itsNext, q.itsNext));
} //================================================

public void timesConstant (double multiplier)
{  if (itsFirst != null)   // an executor must always be non-null
      itsFirst.timesConstant (multiplier);
} //================================================

private static class Node  // inside Polynomial; partial listing
{
   public void timesConstant (double given)
   {  itsCoef *= given;
      if (itsNext != null)
         itsNext.timesConstant (given);
   } //================================================
}
}
```

15.3 **LinkedWorkerList:** Rewrite the entire WorkerList class (Listings 7.8 and 7.9) to use a linked list of Nodes rather than an array.

15.4 **TrackRear:** Rewrite the NodeSequence and NodeSequenceIterator classes so that a NodeSequence object has another instance variable `itsLast`, which is always the last node in the sequence (or null if the sequence is empty). Use it to make the `add` method much more efficient.

15.5 **Positions:** Write a Position interface that extends Iterator (with `remove` as an unsupported operation) by adding `getNext()` to return the element that `next()` would return, but without moving further in the sequence. Define them so that the standard loop for arrays translates as shown:

```
   for (k = 0;  k < itsSize;  k++)
      {x = itsItem[k];...}
   for (p = position(); p.hasNext(); p.next())
      {x = p.getNext();... }
```

Position is to also have the following three instance methods:

- `setNext(anObject)` to replace the value that `getNext()` returns by `anObject`;
- `addNext(anObject)` to insert `anObject` before what `getNext()` returns and make it the value that `getNext()` returns (insert at the end if `hasNext()` is `false`);
- `removeNext()` to remove the value that `getNext()` returns.

Define `next()` to be the equivalent of `getNext(); moveOn()`. Write out a full implementation of Position as a inner class of NodeSequence, possibly with the modifications for the next problem. Allow a call of any method whenever `hasNext()` is true and allow `addNext` anytime. Write an essay comparing this approach with having `add`, `remove`, and `set` added to Iterator.

15.6 **OrderedSequence:** Write a Collection implementation that throws an Exception if some object is added that is not Comparable. Store all objects in a linked list in ascending order as determined by `compareTo`. Include a constructor that makes an ordered sequence from a Collection parameter with unordered Comparable values. The `add` method should add the value in the list to keep the ascending order.

15.7 **HeaderNode:** Rewrite the NodeSequence and NodeSequenceIterator classes so that a NodeSequence's `itsFirst` value acts as a dummy header node with no data, as follows: Rename `itsFirst` as `itsNext` throughout. Delete the `itsNext` field from the nested Node class and make Node a subclass of NodeSequence (as well as being nested in it). Then the first element of the abstract sequence is in `itsNext.itsData`, except a NodeSequence's `itsNext` is null if the sequence has no elements. Then the `add` method is greatly simplified by replacing its call of `addLater` by a recursive call of `add` and eliminating the `addLater` method entirely.

The real improvement is in the ListIterator methods. The iterator does not create its own dummy header node; instead, it initializes `itsPos = NodeSequence.this` (which requires that `itsPos` be declared as a NodeSequence). Simplify the coding of the rest of the ListIterator methods where possible. Extra credit: (a) Add `push`, `pop`, and `peekTop` methods so NodeSequence implements StackADT; do not call on iterator methods or other methods in the NodeSequence class. (b) Use recursion in place of every loop.

15.8 **InternalSorts:** Add two methods to the NodeSequence class, one to do a SelectionSort and one to do an InsertionSort. Each method should be an instance method with no parameters. Revise the TimingSorts project for Chapter Thirteen to compare these two linked-list sorting methods on execution times.

Hint for the InsertionSort: Denote the first node as `lastGood`, since it is the last node in the initial sorted list of one element. As you insert nodes into the sorted list, keep `lastGood` always indicating the last node in the sorted list. So the `bad` node you are to insert next is always the one after `lastGood` and there are three cases to code: Either `bad` is large enough to remain after `lastGood` (that is easy) or `bad` is removed from where it is and then either inserted as the first node on the list or else inserted later in the list. Call a private method to handle the last two cases.

15.9 **LinkedQuickSort:** Add `public void quickSort()` to the NodeSequence class along with the two methods described in the previous problem, and compare its execution time with the other two. Hint: Call a recursive method `private Node sorted (Node pivot, Node end)` with the precondition that `pivot` is an unsorted list with at least two nodes and the postcondition that the node returned is the first node in the sorted list, with the last node in the sorted list linked to `end`.

15.10 **DoublyLinkedList:** Write a List implementation, together with a full implementation of ListIterator, using a doubly-linked list as for TwoWaySequence and TwoWaySequenceIterator, except do not have a header node. Hint: Include a private method `find(indexInt)` to return the Node corresponding to `indexInt`.

15.11 **Word Chains:** Your program should read in a list of four-letter words, one per line, from a text file, and store it in a Collection object. Then repeat the following as long as the user wants to continue: Ask the user for two four-letter words (which may or may not be in the file) and either (a) print some chain starting from the first word and ending with the second word, or (b) tell the user that no chain exists. A chain is defined to be a sequence of words, all from the file except possibly the two inputs, such that each differs in only one letter from the one before it in the sequence. Example: If the user's input is "mail" and "down", you could produce mail -> main -> fain -> fawn -> dawn -> down if the four intermediate words are in the file.

Hint: For each pair of inputs, first make a copy x of the file's collection. Then there is a chain from `firstInput` to `secondInput` only if there is `someWord` that is one letter different from `firstInput` for which there is a chain from `someWord` to `secondInput`. Each time you find a value for `someWord`, remove it from your copy before seeing whether you can make a chain to `secondInput`. Use recursion.

15.12 **ArraysForNodes:** You can save about a third of the storage space that Nodes take up by using two "parallel arrays" instead of Node objects for NodeSequence. Each conceptual Node is represented by an index in the arrays named `data` and `next`, whose lengths are at least as large as the number of Nodes that would ever be used at any one time by the software. Node #5, for instance, stores its data in `data[5]` and the index of the next Node in `next[5]`, so Node #5 need never really exist as an Object. So the `contains` method in Listing 15.5 would be replaced by this coding:

```
for (int k = itsFirst;  k >= 0;  k = next[k])
   if (data[k].equals (ob))
      return true;
return false;
```

Rewrite the entire NodeMap class with this logic. Note that you have to keep track of "Nodes" not currently in use; have `private static int itsTop`, where `itsTop` is the next available "Node" index for use for any NodeSequence that needs it.

16.1 **Parser:** Write the Parser class with the `parseInput` method described in Chapter Sixteen, capable of processing the subset of the Scheme language described in that chapter. Add a main method you can use to test it: Repeatedly get a line of input and say whether it is acceptable.

16.2 **OrderedArrayMap:** Write an array implementation of Mapping in which the Entry objects are kept in ascending order of keys at all times. Have `itsKey` be of a class that implements the Comparable interface (i.e., has the usual `compareTo` method). Use binary search for the `containsKey` method. When the array becomes full, expand it by 50%. Add another map method named `display` that prints to `System.out` all the values in order (this is useful for debugging purposes). Alternative problem: Keep keys in order of their `hashCode` values (`hashCode` is described in Section 16.8).

Also write a main method that creates two maps, named `oneTime` and `manyTimes`, that store word/definition pairs, both objects being Strings. The main method should then read in a sequence of word/definition pairs and store in `oneTime` those words that appear once in the input, but in `manyTimes` those words that appear more than once in the input. When input is complete, print all the word/definition pairs in both maps in order with appropriate titles. Hint: The first time a word appears in the input, put it in `oneTime`; the second time it appears, take it out of `oneTime` and put it in `manyTimes`.

16.3 **TimingLinkedSorts:** Same as Problem 13.1, TimingSorts, except implement all three sorting methods on linked lists with trailer nodes rather than arrays. Hint for Insertion Sort: Start a new list containing only the first data value. Go through the rest of the original list one value at a time and insert it into the new list where it goes.

16.4 **HeaderNodeMap:** The NodeMap class has N+1 nodes whenever it contains N entries, with the last node (the empty list) having null for its entry. That is a trailer node. Write a HeaderNodeMap class that implements a Mapping as a linked list with N+1 nodes whenever it contains N entries, except that it is the first node (the "header node") that has `null` for its entry. Hint: `containsKey` should set `thePosition` to the node before the one that contains the entry that was found, so `put` can insert a node after `thePosition's` node containing the entry to be added.

16.5 **SimpleListMap:** Rewrite the entire NodeMap class to use a simple linked list (no header or trailer) as described at the end of Section 16.5.

16.6 **ListMapIt:** Write the `remove` method for the MapIt class for NodeMap. Add in fail-fast protection: If the program ever calls directly on a NodeMap object X's `remove` or `put` operation after a particular MapIt object is constructed for X, and thereafter calls some MapIt method for that particular MapIt object, the method throws an IllegalStateException. This prevents something like getting the next value using the Iterator and removing it using the NodeMap `remove` method and then calling `next` again, thereby confusing the Iterator object. Hint: Add an instance variable to the NodeMap class that counts the total number of puts and removes that have been executed on a given NodeMap object.

16.7 **ArraysForMapNodes:** Same problem as the ArraysForNodes problem for Chapter Fifteen, but give a full implementation of NodeMap instead.

16.8 **BinarySearchListMap:** Complete the BinarySearchListMap class (using an array for binary search for Comparable keys) that is begun in Listing 16.12. Include an Iterator with a working `remove`. It is okay to allow removals to possibly leave the Node stored in one array component be the same as the Node stored in the next array component. But when the number of values drops to less than twice the number of array components, you should discard the current array and create a smaller one.

16.9 **BinarySearchListCollection:** Write an implementation of the Collection interface that uses the concept of BinarySearchListMap (in the preceding problem), but put data values in order of hashCodes. That makes the set operations of `addAll`, `retainAll`, and `removeAll` execute in big-oh of N time, where N is the larger of the two sets.

16.10 **IndexedMap:** Complete the IndexedMap class (using a hash table) that is begun in Listing 16.13, where each ID is an index. Include an Iterator with a working `remove`.

16.11 **ArrayHashMap:** Complete the ArrayHashMap (using a hash table with open addressing and double-hashing) as described near the end of Section 16.7. Include an Iterator, but do not implement its `remove` method.

16.12 **OpenHashMap:** Complete the OpenHashMap (using a hash table with open addressing and double-hashing) that is begun in Listing 16.15. Include an Iterator, but do not implement its `remove` method.

16.13 **ItemCounter:** Write a class that keeps track of the number of times each word or other object is added to a data structure. You can add a word, remove a word, ask how many times a given word has occurred, ask for the word with the highest frequency, ask for how many words are currently in the data structure, and ask for the sum of the frequencies of words in the data structure. You can get an iterator that iterates through all the different words. However, store arbitrary objects, not just words. Have an instance variable that is a NodeMap or other Map implementation. Make it so that, for any word `w` stored in the data structure, a request for the number of times `w` appears can be answered by `((Integer) itsMap.get (w)).intValue()`.

17.1 **TreeMapIts:** Write the `remove` method for each of the three Iterator implementations coded in the Listings of Chapter Seventeen (Sections 17.5 and 17.6). Call on the `removeData` method in TreeNode as needed. Remember that you must throw an Exception if you cannot remove what the most recent call of `next` has returned, i.e., if `remove` is attempted when `next` has not yet been called or if `remove` has already been called since the most recent call of `next`.

17.2 **BintMapRemove:** Rewrite the `remove` method for BintMap so that, in cases where the data to be removed is in a leaf node, it finds the parent node of the node containing that data to be removed and removes the leaf node.

17.3 **TreeMapUsingHashCodes:** Revise the entire BintMap class to store objects in order of their `hashCode` values (`hashCode` is described in Section 16.8). Then you do not have to restrict it to Comparable keys. The data stored in a node should be from a new class with two instance variables, the `hashCode` of the key plus the MapEntry.

17.4 **ArraysForTreeMappings:** For an implementation of Mapping using binary trees, you can save half or more of the storage space that TreeNodes take up by using an array of Objects to store the data values but not store any child information. Instead, for a given "TreeNode" at index n, store the data value for its left child at index 2*n+1 and store the data value for its right child at index 2*n+2. This works as long as you know the maximum index any "TreeNode" can have. For instance, if the tree is to be restricted to at most 10 levels, then you create an array with 1023 components (since 2 to the 10th power is 1024). This is space-efficient if the Mapping contains at least 256 data values, since it uses one 32-bit word for storage per data value rather than four.

17.5 **RedBlackRemove:** Complete the BintMap implementation of Mapping to use the red-black tree logic. The hardest part is the `remove` method. Then write a main logic that adds 1023 randomly-chosen Integer values to the Mapping, one at a time, and counts the number of times a rotation is made and the number of times the color of an existing node is changed. Divide by 1023 to get two averages. Repeat ten times for an estimate of the overhead of red-black trees.

17.6 **AVL:** Revise the entire BintMap implementation of Mapping to use the AVL tree logic. Then write a main logic that adds 1023 randomly-chosen Integer values to the Mapping, one at a time, and calculates the statistics mentioned in the preceding problem (except count the number of times the `itsLeftMinusRight` value is changed). If someone else in class does the preceding problem, compare the two sets of statistics.

17.7 **2-3-4Trees**: Complete the Mapping implementation using 2-3-4 trees described in Section 17.8 as the BigCircle class.

17.8 **Btree**: Write a Btree class to fully implement the Mapping interface using B-trees. Assume the existence of two random-access files, one an index file organized as a B-tree and the other the data file containing the actual data.

17.9 **GuessAnimals:** Write a program that has the user think of an animal and then repeatedly asks yes-no questions to determine what kind of animal it is. When the program thinks it knows what it is, is makes a guess. If it is wrong, it asks the user for an additional question that would distinguish its guess from the correct answer, and adds the question, the correct answer, and its guess to its database. Thus the program learns as it plays the game over and over.

The database of questions and answers should be stored internally as a binary tree, with each question having two children and each answer being a leaf. Have it write its database to a sequential file in preorder traversal so it can remember it for the next time the game is played. The initial database need only consist of one question, "Does it have fur?", and two answers, "a bear" and "a turtle".

17.10 **ArraysForTreeNodes:** Same problem as the ArraysForNodes problem for Chapter Fifteen, but give a full implementation of BintMap instead. However, in place of `next[k]`, have a two-dimensional array where `child[k,0]` is the left child and `child[k,1]` is the right child of TreeNode #k. This can save nearly half the space that TreeNodes require, if `child` is an array of char values (16 bits each).

17.11 **FileDirectories:** Write an object class that maintains the file information on a hard disk or other permanent storage medium in the standard tree-like structure. The children of one FileDir object should be its sub-directories (also known as sub-folders) in alphabetical order, all FileDir objects, followed by the names of its ordinary files in alphabetical order (String objects).

The FileDir class should Include instance methods makeDir (add a new sub-directory), addFile (add a new file to this directory), removeFile (remove an existing file from this directory), and changeDir (return one of its sub-directories to be used for further commands, given the name of the sub-directory). changeDir("..") should return the parent directory of this directory. Include a toString() method that lists each FileDir object followed by its children indented one level beyond it, in preorder traversal.

18.1 **UnorderedPriQue:** Write a complete implementation of PriQue using a partially-filled array of Store objects. The Store class is nested inside the UnorderedPriQue class. A Store object has two instance variables -- the element being stored and an int value that counts the total number of values added to the priority queue so far. The following is the `add` method; you have to write the rest of the class including Store. When you remove a value and there are several with the same highest priority, you can easily find the one that was added earliest. And you can simply replace it with the first value in the list. Alternative problem: Do this with a linked list rather than a partially-filled array.

```
public void add (Object ob)
{  if (itsSize == itsItem.length)
   { } // same as in ArrayStack's push method, Listing 14.2
   itsNumAdded++;
   itsItem[itsSize] = new Store (ob, itsNumAdded);
   itsSize++;
}
```

18.2 **IntPriQue:** Write an implementation of PriQue for which the priority of each object is given by an int value in the range 1 through MAX, where MAX is some positive integer. Use an array of NodeQueues, as described at the end of Section 18.5.

19.1 **ListEdgesTo:** Revise the ListGraph class to have an efficient implementation of a Graph method `public Iterator edgesTo (Vertex v)`: The executor returns an iterator over the Edge objects that have v as their head. Do this by adding `private ArrayList itsOutList = new ArrayList();` to the ListGraph declarations, which stores one list per vertex v of those Edge objects that have v as the head. Use that list appropriately.

19.2 **MatrixEdgesTo:** Revise the MatrixGraph class to have an efficient implementation of a Graph method `public Iterator edgesTo (Vertex v)`: The executor returns an iterator over the Edge objects that have v as their head.

19.3 **BigGraph:** Write a BigGraph class that has both a two-dimensional matrix `edgeAt` (like MatrixGraph) and a one-dimensional array of ArrayLists (like ListGraph). Have `addVertex(Vertex)`, `add(Edge)`, and `remove(Edge)` maintain both structures. Have the query methods access whichever structure provides the fastest response time.

19.4 **UnionFind:** Revise the UnionFind class in Listing 19.10 to have `unite` execute in big-oh of 1 time but `equates` execute in big-oh of log(N) time as follows: itsItem stores a Node that has a counter (initially 1) and a reference to a parent (initially null). To unite two int values x and y with Nodes `itsItem[x]` and `itsItem[y]`, have the Node with the smaller count reference the other Node as its parent. You can then tell if two int values equate by whether tracing the parent pointers to the Node with no parent arrives at the same Node. Explain clearly why equates is a big-oh of log(N) operation.

19.5 **PrimsAlgorithm:** Write coding for Prim's algorithm, analogous to Listing 19.11.

19.6 **JonesAlgorithm:** Write coding for Jones's algorithm, analogous to Listing 19.11.

19.7 **TransitiveClosure:** Write a makeTransitiveClosure method in both MatrixGraph and ListGraph to convert the executor Graph into its transitive closure.

Many More Problems: Hundreds more problems are available from the Association for Computing Machinery website http://acm.uva.es/problemset/. These are contest problems. You may submit solutions to be evaluated by a non-human judge.