# The Art & Science of Java™

*An Introduction to Computer Science*

## Answers to Review Questions

### Chapter 1. Introduction

1. Babbage's Analytical Engine introduced the concept of *programming* to computing.
2. Augusta Ada Byron is generally recognized as the first programmer. The U.S. Department of Defense named the Ada programming language in her honor.
3. The heart of von Neumann architecture is the *stored-programming concept,* in which both data and programming instructions are stored in the same memory system.
4. Hardware is tangible and comprises the physical parts of a computer; software is intangible and consists of the programs the computer executes.
5. The abstract concept that forms the core of computer science is problem-solving.
6. For a solution technique to be an algorithm, it must be
   - *Clearly and unambiguously defined*
   - *Effective,* in the sense that its steps are executable
   - *Finite,* in the sense that it terminates after a bounded number of steps
7. *Algorithmic design* refers to the process of designing a solution strategy to fit a particular problem; *coding* refers to the generally simpler task of representing that solution strategy in a programming language.
8. A higher-level language is a programming language that is designed to be independent of the particular characteristics that differentiate computers and to work instead with general algorithmic concepts that can be implemented on any computer system. The higher-level language used in this text is called Java.
9. Each type of computer has its own machine language, which is different from that used in other computers. The compiler acts as a translator from the higher-level language into the machine language used for a specific machine.
10. A source file contains the actual text of a program and is designed to be edited by people. An object file is created by the compiler and contains a machine-language representation of the program. Most programmers never work directly with object files.
11. A syntax error is a violation of the grammatical rules of the programming language. The compiler catches syntax errors when it translates your program. Bugs are errors of the logic of the program. Programs containing bugs are perfectly legal in the sense that the compiler can find no violation of the rules, but nonetheless they do not behave as the programmer intended.

12. False. Even the best programmers make mistakes. One of the marks of a good programmer is the ability to find and correct those mistakes.
13. False. Between 80 and 90 percent of the cost of a program comes from maintaining that program after it is put into practice.
14. The term *software maintenance* refers to development work on a program that continues after the program has been written and released. Although part of software maintenance involves fixing bugs that were not detected during initial testing, most maintenance consists of adapting a program to meet new requirements.
15. Programs are usually modified many times over their lifetimes. Programs that use good software engineering principles are much easier for other programmers to understand and are therefore easier to maintain.
16. Under the traditional procedural paradigm, programs consist of a collection of procedures and functions that operate on data; the more modern object-oriented paradigm treats programs as a collection of *objects*, for which the data and the associated operations are encapsulated into integrated units.
17. Running an applet in a web browser consists of the following steps:
    - The user enters the URL for the applet page into a web browser.
    - The browser reads and interprets the HTML source for the web page.
    - The appearance of an **`applet`** tag in the HTML source file causes the browser to download the compiled applet over the network.
    - A verifier program in the browser checks the applet intermediate code to ensure that it does not violate the security of the user's system.
    - The Java interpreter in the browser program runs the compiled applet, which generates the desired display on the user's console.

    Executing a program as a Java application runs the interpreter directly without the intercession of the browser.

## Chapter 2. Programming by Example

1. The purpose of the comment at the beginning of each program is to convey information to the human reader of the program about what it does.
2. A library package is a collection of tools written by other programmers that perform useful operations, thereby saving you the trouble of implementing those operations yourself.
3. When you execute a program developed using the **`acm.program`** package, Java invokes the **`run`** method.
4. An argument is information that the caller of a particular method makes available to the method itself. By accepting arguments, methods become much more general tools that provide considerable flexibility to their callers.
5. The **`println`** method is used in the context of a **`ConsoleProgram`** to display information to the user. The suffix **`ln`** is a shorhand for the word *line* and indicates that the output displayed by the **`println`** call should appear on the current line but that any subsequent output should begin on the following line.
6. The **`readInt`** method requests an integer value from the user and then returns that value so that it can be used as input to the program. The standard pattern for using the **`readInt`** method looks like this:

   **`int`** *variable* **`= readInt(`**"*prompt*"**`);`**

7. The **+** operator is used to signify *addition* when it is applied to numeric arguments and *concatenation* when at least one of its operands is a string.
8. *Reductionism* is the philosophical theory that the best way to understand a large system is to understand in detail the parts that compose it. By contrast, *holism* represents the view that the whole is greater than the sum of the parts and must be understood as a separate entity unto itself. As a programmer, you must learn to view your work from both perspectives. Getting the details right is critical to accomplishing the task. At the same time, setting those details aside and focusing on the big picture often makes it easier to manage the complexity involved in programming.
9. A *class* defines a general template for objects that share a common structure and behavior. An *object* is a particular instance of a class. A single class can serve as a template for many different objects, but each object will be created as an instance of a single class.
10. In object-oriented languages, one typically defines a new class by extending an existing one. In Java, the syntax for doing so looks like this:

    **public class** *new class* **extends** *existing class*

    In this case, the new class becomes a *subclass* of the existing class. The *superclass* relation is the inverse; given this definition, the existing class becomes the superclass of the new class. The idea of *inheritance* is simply that any subclass automatically acquires the public behavior of its superclass.
11. Java uses the keyword **new** to create a new object by invoking the constructor for its class.
12. The only difference between a **ConsoleProgram** and a **DialogProgram** is the model used to request information from and report results to the user. In a **ConsoleProgram**, all information in both directions uses the console. A **DialogProgram** uses dialog boxes for this purpose.
13. True. Sending a message to a Java object is in fact always implemented by calling a method in that object. In the usual case, that call appears explicitly in the program, although it can also be triggered by events, as discussed in Chapter 10.
14. In Java, you indicate the *receiver* for a message by naming that object and then appending a period and the name of the method, as follows:

    *receiver***.***method***(***arguments***)**
15. The four **GObject** subclasses introduced in Chapter 2 are **GLabel**, **GRect**, **GOval**, and **GLine**.
16. Both the **GRect** and **GOval** classes respond the **setFilled** method; only the **GLabel** class responds to **setFont**.
17. Yes. All **GObject** subclasses respond to the **setColor** method because it is defined for the class as a whole. Any subclass of **GObject** therefore inherits this behavior.
18. The primary difference between Java's coordinate system and the traditional Cartesian model is that the origin appears in the upper left rather than in the lower left. The units in the Java model are measured in terms of *pixels,* which are the individual dots that cover the face of the display screen.

## Chapter 3. Expressions

1. A data type is defined by a domain and a set of operations.

2. a) legal, integer
   b) legal, integer
   c) illegal (contains an operator)
   d) legal, floating-point
   e) legal, integer
   f) legal, floating-point
   g) illegal (contains commas)
   h) legal, floating-point
   i) legal, integer
   j) legal, floating-point
   k) legal, floating-point
   l) illegal (contains the letter **`X`**)

3. a) **`6.02252E+23`**
   b) **`2.997925E+10`**
   c) **`5.29167E-10`**
   d) **`3.1415926535E0`**

4. a) legal
   b) legal
   c) legal
   d) illegal (contains **`%`**)
   e) illegal (**`short`** is reserved)
   f) legal
   g) illegal (contains a space)
   h) legal
   i) illegal (begins with a digit)
   j) illegal (contains a hyphen)
   k) legal
   l) legal

5. a) **`int`**: 5
   b) **`int`**: 3
   c) **`double`**: 3.8
   d) **`double`**: 18.0
   e) **`int`**: 4
   f) **`int`**: 2

6. The unary minus operator is written before a term, as in **`-x`**; the binary subtraction operator uses the same symbol but is written between two terms, as in **`x - 3`**.

7. a) 4
   b) 2
   c) 42
   d) 42

8. The variable **`k`** would contain the value 3 after the first assignment and 2 after the second. When a floating-point value is cast to an integer, its value is *truncated* by dropping any decimal fraction.

9. To convert between numeric types in Java, you need to apply a *type cast,* which consists of the name of the desired type written in parentheses before the value you wish to convert.

10. **`cellCount *= 2;`**

11. **`x++;`**

12. The two values of type **`boolean`** are **`true`** and **`false`**.

13. Java uses the single equal sign to indicate assignment and not as a check for equality. In most cases, Java will catch the error because the result is not ordinarily a **`boolean`** value.

14. **`0 <= n && n <= 9`**

15. The expression checks to see if the value **`x`** is either (a) not equal to 4 or (b) not equal to 17. No matter what value **`x`** has, one of these two conditions must be true because **`x`** cannot be both 4 and 17 at the same time.

16. *Short-circuit evaluation* means that evaluation of a compound Boolean expression written with the operators **`&&`** and **`||`** stops as soon as the result is known.

17. Named constants are useful for several reasons. For one thing, the symbolic name carries more information, which typically makes the resulting program easier to read. In addition, defining a named constant usually makes it easier to maintain a program by ensuring that a change in the definition of the constant is reflected everywhere that constant is used.

## Chapter 4. Statement Forms

1. The construction `17;` is a legal statement in Java because it is a legal expression followed by a semicolon. It has, however, no useful effect.
2. A *block* is a sequence of statements enclosed in curly braces. Blocks are often called *compound statements* because a block acts syntactically as a statement and can be used in any context in which a statement is required.
3. The two classes of control statements are *conditional* and *iterative*.
4. Control statements are said to be *nested* if one is entirely enclosed by the other.
5. The four formats of the **`if`** statement used in the text are: (1) the single-line **`if`** statement, (2) the multiline **`if`** statement, (3) the **`if-else`** statement, and (4) the cascading **`if`** statement.
6. The **`switch`** statement first evaluates the parenthesized expression, which must be an integer (or integer-like type, as discussed in Chapter 8). The **`switch`** statement then compares the value against each of the constants appearing in **`case`** clauses within the body. If a match exists, Java executes the statements associated with that **`case`** clause. If none of the **`case`** expressions match the value, Java executes the **`default`** clause, if any.
7. A **`while`** loop tests the conditional expression only at the beginning of each loop cycle. If the test becomes **`false`** in the middle of a loop cycle, control continues until that cycle is completed. If the **`while`** test is still **`false`** at that point, the loop exits.
8. The **`DigitSum`** program uses the remainder operator `%` to select individual digits. When `%` is applied to a negative integer, the remainder is defined to be negative, which does not produce the desired digit.
9. The *loop-and-a-half problem* occurs when the test for loop completion comes logically in the middle of a cycle. One approach to solving this problem is to use the **`break`** statement to exit from the loop at the point at which the conditional test occurs. It is also possible to reorder the statements in the loop so that the test appears at the beginning, although this approach usually requires some duplication of code.
10. The three expressions in the control line of a **`for`** statement—*init*, *test*, and *step*,—indicate respectively what operations should be performed to initialize the loop, what conditional expression is used to determine whether the loop should continue, and what operations should be performed at the end of one loop cycle to prepare for the next.
11. a) `for (int i = 1; i <= 100; i++)`
    b) `for (int i = 0; i < 100; i += 7)`
    c) `for (int i = 100; i >= 0; i -= 2)`
12. Floating-point values are only approximations of real numbers in mathematics and may not always behave the same way. In a **`for`** loop that tests for equality between two floating-point numbers, slight errors in the computation may cause the loop to execute a different number of cycles than the programmer might expect.
13. The programs in this chapter that implement animation do so by executing a loop that makes a small change to the graphical objects on the screen and then suspends the program for a short time interval.

14. The purpose of the **pause** call is to delay the operation of the program so that the human eye can follow the changes on the canvas. If you left out the call to the **pause** method, the entire program would finish so quickly that the square would seem to jump instantaneously to its final position.

## Chapter 5. Methods

1. A *method* is a named collection of statements that performs a particular operation, usually on behalf of some other method. A *program* executes a task on behalf of the user. When using the ACM Java Libraries, a program is always a subclass of the **Program** class defined in the package **acm.program**.
2. Whenever you need to invoke the operation performed by a particular method, you *call* that method by specifying its name, followed by a list of values enclosed in parentheses. When Java executes the method call, the calling method is suspended and execution continues beginning with the first statement in the called method. The values in parentheses are called *arguments;* these values are copied into the corresponding formal parameter variables declared by the method. When execution of a method is complete, it *returns* to the calling program, which means that the completed method and its local variables disappear, after which Java continues executing the previously suspended method.
3. Arguments provide a mechanism by which one method can communicate information to another. Input methods like **readInt**, on the other hand, provide a mechanism by which the user can communicate information to a program. If, for example, a program were designed to add a list of integer entered by the user, those values would be read using **readInt**. If, however, you needed to calculate a square root as part of some more complex computation, you would instead pass the appropriate value as an argument to the **Math.sqrt** method.
4. To return a value from a method, you use the keyword **return** followed by an expression specifying the return value.
5. There can be any number of **return** statements in a method.
6. In Java, you indicate the receiver of a message by appending a period and the name of the method, like this:

   *receiver*`.`*method*`(`*arguments*`)`

7. If no receiver is specified for a method, Java applies that method to the current object. In the programs from the earlier chapters, most of the methods are applied to the current object.
8. The **case** clauses in the **monthName** method do not require **break** statements because they end with a **return** statement. The **return** statement causes the program to exit at that point, which means that the program will not continue into the next **case** clause.
9. A *predicate method* is one that returns a Boolean value.
10. You apply the **equals** method, using one string as the receiver and the other as an argument.
11. This text uses the term *arguments* is used to refer to the expressions that are part of a method call; the term *formal parameters* refers to the variable names in the method header into which the argument values are copied.

12. Local variables are only visible inside the method in which they are declared. Outside that method, those variables are undefined, which means that their declaration is "local" to the defining method.

13. The term *return address* signifies the point in the calling method at which execution will continue when a method returns. Java's runtime system must save the return address whenever it makes a method call.

14. The strategy of *stepwise refinement* consists of breaking a program down into methods that solve successively simpler subproblems. To apply the technique, you always begin with the most general statement of the problem. You code the solution to the problem as a method in which you represent any complex operations as method calls, even though you have yet to write the definitions of those methods. Once you finish one level of the decomposition, you can go ahead and apply the same process to the methods that solve the subproblems, which may in turn divide to form new subproblems. The process continues until all operations are simple enough to code without further subdivision.

15. A *brute force* algorithm is one that tries every possibility.

16. The greatest common divisor of 7735 and 4185 is 5. Euclid's algorithm determines this result by computing the following values of **`r`** in successive cycles of the **`while`** loop: 3550, 635, 375, 260, 115, 30, 25, 5, and 0.

17. The code for Euclid's algorithm works even if **`x`** is smaller than **`y`**. When this occurs, the first cycle of the **`while`** loop ends up exchanging the values of **`x`** and **`y`**.

## Chapter 6. Objects and Classes

1. In many programs—most notably games—it is useful to have the computer behave in a way that the computer cannot predict. Nondeterminism is also useful in applications that simulate unpredictable events in the real world.

2. A *pseudorandom* number is a numeric value that appears to be random in a statistical sense even though it is generated by an algorithmic process.

3. `private RandomGenerator rgen = RandomGenerator.getInstance();`

4. a) `rgen.nextInt(0, 9)`
   b) `rgen.nextDouble(-1.0, 1.0)`
   c) `rgen.nextBoolean(1.0 / 3.0)`
   d) `rgen.nextColor()`

5. No. This statement sets both **`d1`** and **`d2`** to the same value.

6. When you are debugging a program, it is often useful to set the random number seed to a fixed value so that the program behaves the same way every time. Doing so makes it easier to reproduce a previous failure.

7. The implementor is concerned with the details of how the methods in a class are implemented. The client does not care about those details but instead needs to know how to use the facilities provided by a class to accomplish a specific task.

8. Clients and implementors must agree on how to use each public method in a class and what effects each of those methods has. This information is typically provided by the **javadoc** entries for the class.

9. The **`javadoc`** documentation system automatically creates web-readable documentation for a Java class or package. The information is taken from the

header-line information for each public entry as well as from specially formatted comments written by the implementor.

10. A *layered abstraction* consists of a set of classes in which each subclass builds on the framework established by its predecessors in the superclass hierarchy.

11. If you omit the **`extends`** clause in a class definition, Java makes the new class a direct subclass of the **`Object`** class, which is the root of Java's entire class hierarchy.

12. The most common entries in a class definition are constructors, methods, named constants, and instance variables.

13. True. Using private instance variables ensures that the details of the internal state remain hidden from clients.

14. The term *overloading* is used to refer to classes that define several methods with the same name but different parameter patterns. Java is able to determine which version to invoke by matching the calling arguments against the various parameter lists. The term *overriding* refers to the process of defining a new version of a method that is already defined in the superclass. The new definition supersedes the preceding one.

15. The keyword **`this`** indicates the current object; the keyword **`super`** refers to the superclass of the current object. This chapter uses each of these keywords in the definition of constructors, using **`this`** to signify calling a different constructor in the same class and **`super`** to indicate calling one of the superclass constructors. The chapter also uses **`this`** in several of the method definitions in the **`Rational`** class to emphasize that the instance variables **`num`** and **`den`** refer to the current object and not some other instance of this class.

16. An *immutable class* is one that does not allow the client to change any of the properties associated with an object once it has been created.

17. The **`toString`** method determins how an object of a particular class is represented as a string. Java automatically invokes the **`toString`** method in contexts where a string value is needed, such as the concatenation operator or the implementation of **`println`**. Defining a new **`toString`** method for each class ensures that the user can interpret the value of an object whenever it is displayed.

## Chapter 7. Objects and Memory

1. A *bit* is the smallest possible unit of information and has exactly two states, usually represented as 0 and 1. A *byte* is a collection of eight bits, and is typically the smallest unit that can be manipulated by the hardware. A *word* is a larger collection of information capable of holding the standard-sized integer value for the hardware.

2. The term *bit* is a contraction of the words *binary digit*.

3. 402,653,184 byes.

4. a) **`11`**
   b) **`100`**
   c) **`6C1`**
   d) **`ACE`**

5. a) 23
   b) 100
   c) 204
   d) 4013

6. An *address* is the numeric memory location at which a particular value is stored.

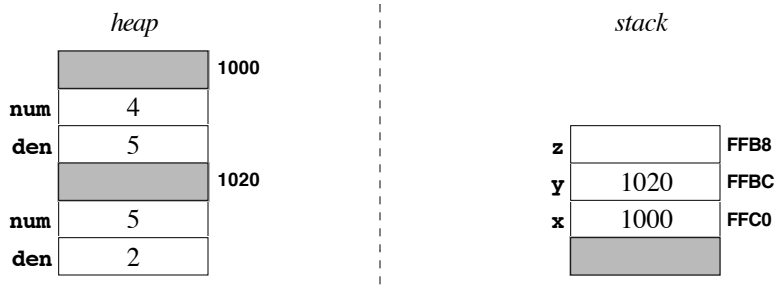
7. Java assigns 4 bytes to an **`int`** and 8 bytes to a **`double`**.

8. The three memory areas are: the *static area* used for constants and other static data, the *heap* used to store new objects, and the *stack* used for local variables.

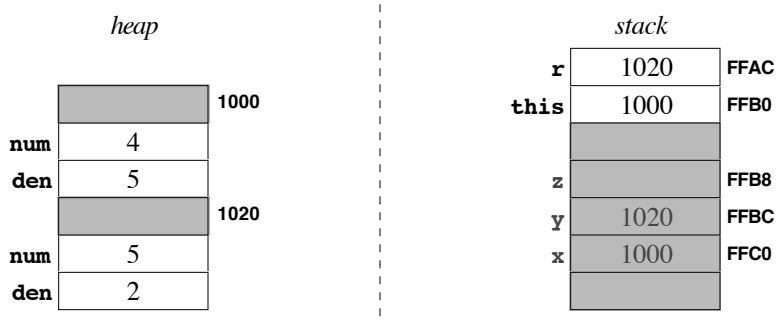
9. The first printing of the book contains an error that prevents this method from compiling: the third line of the **run** method should read as follows:

```
Rational z = x.multiply(y).subtract(x);
```

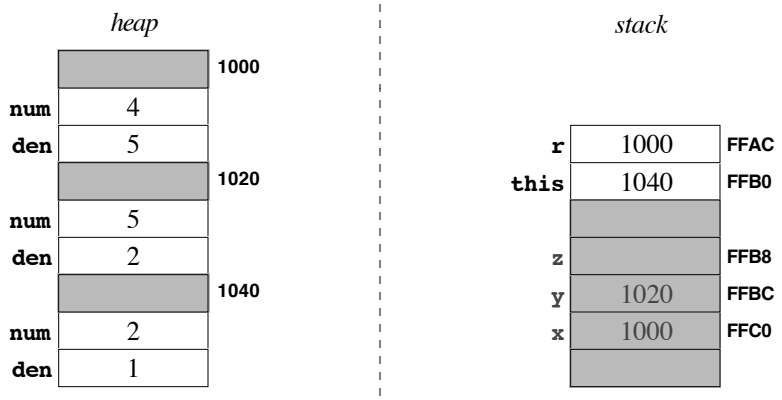
Given this change, the memory image prior to executing this line looks like this:



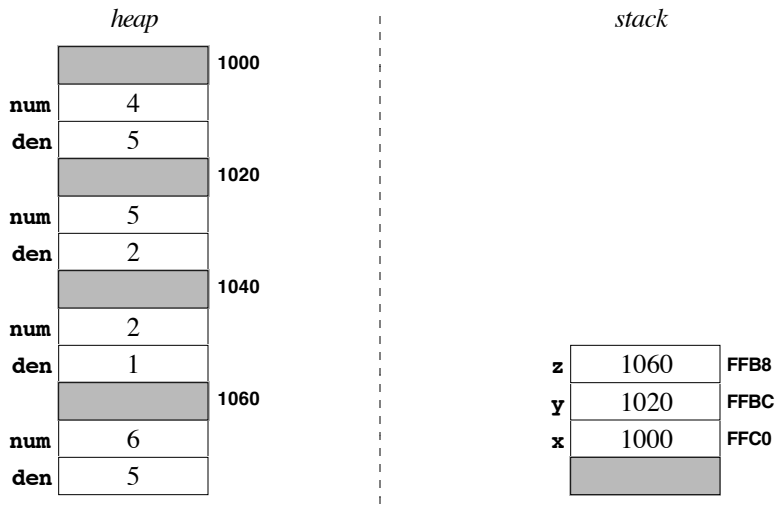

Executing the **multiply** call creates a new stack frame using **x** as the receiver and **r** as the argument, like this:




Executing the body of **multiply** creates a new **Rational** object with a value that reduces so that it has 2 as its numerator and 1 as its denominator. That value is not stored, but simply passed to the **subtract** call, creating the following state:




The **subtract** method computes the result of subtracting four-fifths from two and creates another **Rational** object on the stack to store the result. When **subtract** returns, Java stores the result in the local variable **z**, as follows:




At the end of this process, the temporary result at address 1040 is inaccessible and is therefore garbage.

10. True. Primitive values are always copied when they are used as parameters.

11. False. When you pass an object as a parameter, Java copies the address of that object but not the internal data.

12. A *mark-and-sweep collector* executes the following phases:

    1. Go through every variable reference on the stack and every variable reference in static storage and mark the corresponding object as being "in use" by setting a flag that lives in the object's overhead space. In addition, whenever you set the "in use" flag for an object, determine whether that object contains references to other objects and mark those objects as being "in use" as well.

    2. Go through every object in the heap and delete any objects for which the "in use" flag has not been set, returning the storage used for that object to the heap for subsequent reallocation. It must be safe to delete these objects, because there is no way for the program to gain access to them, given that no references to those objects still exist in the program variables.

13. A *wrapper class* is an actual Java class that corresponds to one of the primitive types. Each wrapper class encapsulates the corresponding primitive object so that the wrapped value can be used in contexts that require objects.

14. The static **`toString`** method in the **`Integer`** class allows the caller to specify the base by providing a second argument. Thus, the call **`Integer.toString(42, 16)`** returns the hexadecimal equivalent of the integer 42.

15. A *linked structure* is one in which the objects contain references to other objects. In most cases, these links define the relationships among the objects in the structure.

16. Although an object cannot contain the data for another object of the same class, it can contain a *reference* to that object, which is simply the address of that object in memory.

17. The value **`null`** is used to indicate a reference to an object that does not exist. In linked structures, the value **`null`** is used to signify the end of a chain.

## Chapter 8. Strings and Characters

1. The principle of enumeration is that finite set of values can be represented by assigning an integer to each element of the set.
2. The traditional approach to defining enumerated types in Java is to define a named constant that assigns an integer value to each element of the set. As of Java 5.0, it is also possible to use the keyword **`enum`** to define an enumerated type.
3. A *scalar type* is any type that is represented internally as an integer and that can therefore be used in contexts like the **`switch`** statement or array indexing, which require integer values.
4. You can include a double quotation mark inside a string by preceding it with a backward slash.
5. The acronym *ASCII* stands for *American Standard Code for Information Interchange*.
6. The older ASCII coding system uses 8 bits to represent a character, which means that it can represent only a small subset of the character sets used in the world. The Unicode system uses 16 bits for a character, which gives it the ability to represent a much larger set of 65,536 characters. The ASCII characters correspond to the first 256 characters in the Unicode set.
7. The character `'$'` has the Unicode value $44_8$ (36), `'@'` is $100_8$ (64), `'\t'` is $11_8$ (9), and `'x'` is $170_8$ (120).
8. The fact that the digit character are consecutive means that you can test to see if a character is a digit by testing whether it is in the appropriate range. It also makes it possible to convert a character to its corresponding value by subtracting the value of the character `'0'`. In practice, each of these operations is ordinarily performed by calling a static method in the **`Character`** class.
9. The four most useful arithmetic operators on characters are:
   - Adding an integer to a character
   - Subtracting an integer from a character
   - Subtracting one character from another
   - Comparing two characters against each other
10. Calling **`Character.isDigit(5)`** returns **`false`** because 5 is not the Unicode representation for a digit. Calling **`Character.isDigit('5')`** returns **`true`**. It is not legal to call **`Character.isDigit("5")`** because the argument cannot be converted to an integer.
11. Calling **`Character.toUpperCase('5')`** returns `'5'` because the **`toUpperCase`** and **`toLowerCase`** methods simply return the argument if it is not a letter.
12. Using the methods from the **`Character`** class has the following advantages:
    - Programs you write will be easier for other programmers to read.
    - It is easier to rely on library methods for correctness than on your own.
    - The code you write yourself will not take advantage of the international applicability of Unicode.
    - The implementations of methods in the library packages are typically more efficient than those you would write yourself.

13. True. Because characters are scalar types, you can use them as **`case`** expressions.

14. None. The **`String`** class is immutable, which means that none of the methods in the class ever changes the string.

15. **`str = str.trim();`**

16. An *immutable class* is one that does not allow the client to change any of the properties associated with an object once it has been created.

17. a) 5
    b) 0
    c) 1
    d) **`'C'`**
    e) **`"ABC"`**
    f) **`"CDE"`**
    g) 2
    h) -1
    i) 3
    j) **`"abcde"`**

18. The most important caution is that the relational operators cannot be used to compare string values. To test whether two strings are equal, use **`equals`**; to determine whether one string comes before another in lexicographic order, use **`compareTo`**.

19. a) **`false`**
    b) **`true`**
    c) 0
    d) some unspecified positive value
    e) some unspecified negative value
    f) **`false`**

20. The optional third argument to the **`StringTokenizer`** constructor is a Boolean flag that indicates whether the tokenizer should return the delimiter characters as part of the token stream. If this argument is **`false`**, delimiter characters serve only to separate tokens and are not returned to the client; if this argument is **`true`**, delimiter characters are returned as single-character strings.

21. The **`nextToken`** method returns the next token from a token stream created by the **`StringTokenizer`** constructor, which requires updating the internal state of the tokenizer to mark that another token has been read. The **`hasMoreTokens`** method returns **`true`** as long as there are more tokens to read and **`false`** when it reaches the end of the token stream.

## Chapter 9. Object-oriented Graphics

1. The **`acm.graphics`** package uses a model in which graphical displays are constructed by adding various **`GObject`**s to a **`GCanvas`** in much the same way that an artist adds objects to a collage.

2. Java's graphics model—which is also used in the **`acm.graphics`** package—measures distances and coordinate values in *pixels*, which are the dots that cover the face of the screen.

3. The origin in Java's graphics model is in the upper left corner of the window.

4. An abstract class is similar to a standard Java class except in that it is illegal to construct an object of an abstract class directly. What clients do instead is construct instances of some concrete subclass of the abstract class. Thus, it is illegal to construct a **`GObject`** *per se*, but legal to construct a specific subclass of **`GObject`**, such as **`GRect`** or **`GOval`**.

5. See Figure 9-1.

6. The **`GFillable`** interface defines the following methods: **`setFilled`**, **`isFilled`**, **`setFillColor`**, and **`getFillColor`**. This interface is implemented by **`GRect`**, **`GOval`**, **`GArc`**, and **`GPolygon`**.

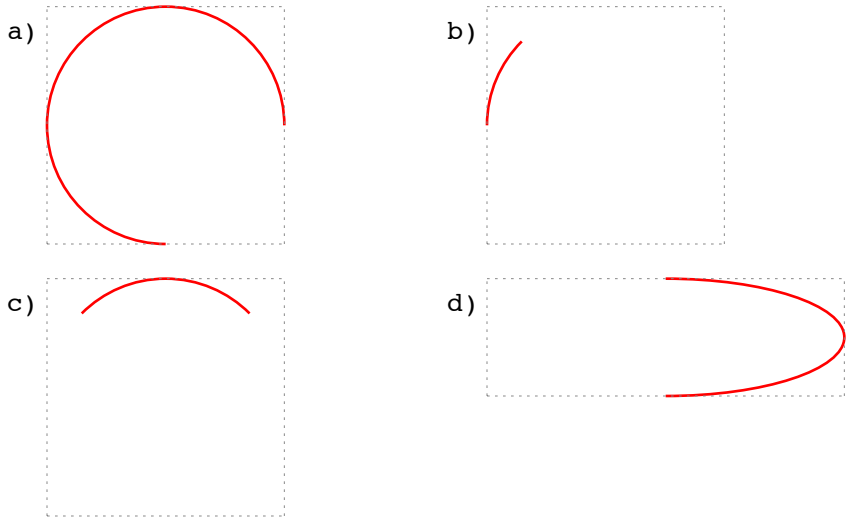
7. The **GResizable** interface specifies a **GObject** class whose size can be reset by calling the **setSize** method. The **GScalable** interface, by contrast, refers to classes that can be stretched in either of both dimensions. The **GArc**, **GCompound**, **GLine**, and **GPolygon** classes are rescalable but not resizable.

8. The **getWidth** method returns the horizontal extent of the **GLabel** in its current font. The **getHeight** method returns the height of the font, which is the vertical distance between successive baselines. The **getAscent** and **getDescent** methods return the maximum distance that characters in the current font rise above or descend below the baseline, respectively.

9. The **start** parameter indicates the angle at which the arc begins, and the **sweep** parameter indicates how far the arc extends around the ellipse. Both angles are measured in degrees, with positive values indicating counterclockwise rotation.

10. If a **GArc** is unfilled, only the arc itself appears; calling **setFilled(true)** on that arc fills in the entire pie-shaped wedge formed by joining the ends of the arc to the center of the ellipse.

11. The first printing of the book specifies sizes for the arcs that makes them too small to see the structure. Here are the drawings at a larger scale, with the dotted rectangle showing the bounding box:




12. A negative **sweep** angle specifies clockwise rotation from the starting point.

13. The **setLocation** method moves the line so that its starting location is at the new position without changing the length and orientation of the line, which means that the end point must move by the same amount. The **setStartPoint** method changes the starting position of the line without changing the end point.

14. The method definitions in Figure 1 create a **GPolygon** whose reference point is the center of the figure and for which the parameter **edge** specifies the length of an edge.

15. The center coordinates can be calculated as follows:

```
double cx = getWidth() / 2;
double cy = getHeight() / 2;
```

16. The **GCompound** class is used to combine various **GObject**s into a single object that you can then manipulate as a unit.

**Figure 1. Methods to create the polygonal forms from review question 16**

```
/* Creates a parallelogram with its reference point at the center */
   private GPolygon createParallelogram(double edge) {
      GPolygon poly = new GPolygon();
      double x0 = -(edge + edge / Math.sqrt(2)) / 2;
      double y0 = edge / Math.sqrt(2) / 2;
      poly.addVertex(x0, y0);
      poly.addPolarEdge(edge, 0);
      poly.addPolarEdge(edge, 45);
      poly.addPolarEdge(edge, 180);
      poly.addPolarEdge(edge, 225);
      return poly;
   }

/* Creates a pentagon with its reference point at the center */
   private GPolygon createPentagon(double edge) {
      GPolygon poly = new GPolygon();
      double x0 = -edge / 2;
      double y0 = edge / 2 / GMath.tanDegrees(36);
      poly.addVertex(x0, y0);
      for (int i = 0; i < 5; i++) {
         poly.addPolarEdge(edge, i * 72);
      }
      return poly;
   }

/* Creates a cross polygon with its reference point at the center */
   private GPolygon createCross(double edge) {
      GPolygon poly = new GPolygon();
      double x0 = -2.5 * edge;
      double y0 = edge / 2;
      poly.addVertex(x0, y0);
      poly.addEdge(edge, 0);
      poly.addEdge(0, edge);
      poly.addEdge(edge, 0);
      poly.addEdge(0, edge);
      poly.addEdge(edge, 0);
      poly.addEdge(0, -edge);
      poly.addEdge(edge, 0);
      poly.addEdge(0, -edge);
      poly.addEdge(edge, 0);
      poly.addEdge(0, -edge);
      poly.addEdge(-edge, 0);
      poly.addEdge(0, -edge);
      poly.addEdge(-edge, 0);
      poly.addEdge(0, -edge);
      poly.addEdge(-edge, 0);
      poly.addEdge(0, edge);
      poly.addEdge(-edge, 0);
      poly.addEdge(0, edge);
      poly.addEdge(-edge, 0);
      poly.addEdge(0, edge);
      return poly;
   }
```

## Chapter 10. Event-driven Programs

1. An *event* is action that occurs asynchronously with respect to the program operation, such as clicking the mouse or typing on the keyboard. Interactive programs that operate by responding to these events are said to be *event-driven*.
2. An *event listener* is an object that has been designated to listen for events. In Java, event listeners come in many different forms depending on the type of event.
3. The standard event classes and the listener interfaces are defined in the package **`java.awt.event`**.
4. True. The **`Program`** class defines empty listener methods for responding to mouse, keyboard, and action events. If a specific program subclass needs to respond to particular events, all that the programmer needs to do is override the methods necessary to implement the appropriate response.
5. The **`init`** method is used to specify initialization code that needs to be executed before the program starts. The **`run`** method specifies what the program does when it runs. Event-driven programs often define only the **`init`** method because all actions after the initial setup occur in the context of listener methods.
6. In Java, the task of responding to a mouse events is broken up into two listener interfaces: **`MouseListener`** and **`MouseMotionListener`**. The former is used for relatively infrequent events, such as pressing the mouse button; the latter is used for events that occur many times in rapid succession, such as those generated when moving or dragging the mouse. Many event-driven programs require only the **`MouseListener`** events; separating mouse events into two categories makes it possible for those programs to operate more efficiently because they no longer have to respond to the much more frequent events in the **`MouseMotionListener`** category.
7. False. A **`mouseClicked`** event is always preceded by both a **`mousePressed`** and a **`mouseReleased`** event.
8. The **`addMouseListeners`** method adds the current **`GraphicsProgram`** to its **`GCanvas`** as both a **`MouseListener`** and a **`MouseMotionListener`**. The **`addKeyListeners`** method adds the program to the canvas as a **`KeyListener`**. The **`addActionListener`** method adds the program as an **`ActionListener`** to every button nested anywhere inside the program window.
9. The interactors described in this chapter are useful in the following contexts:
   - A **`JButton`** allows the user to trigger an action with a single mouse click.
   - A **`JCheckBox`** allows the user to change an option that is either *on* or *off*.
   - A **`JRadioButton`** is similar to a check box in that it defines an option that exists in either an *on* state or an *off* state. Radio buttons, however, are usually designated as part of a **`ButtonGroup`** in which only a single radio button can be on at any time. This facility makes it possible for the user to choose among a small set of mutually exclusive options.
   - A **`JSlider`** allows the user to vary a parameter along a continuous scale.
   - A **`JLabel`** is not really an interactor in the conventional sense but instead makes it possible to create labels that describe the function of other interactors in a graphical user interface.
   - A **`JComboBox`** consists of a popup menu of options from which the user can make a selection. It therefore plays a role similar to that of radio buttons but takes up

less space on the screen. This space savings is particularly important if the number of options is large.

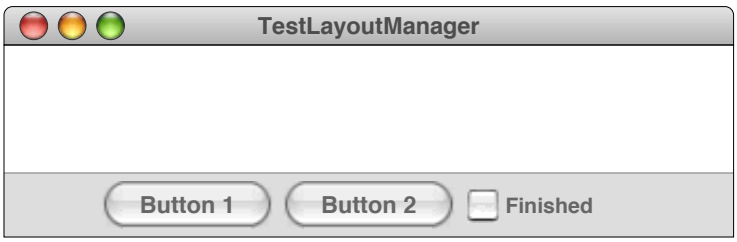
- A **JTextField** allows the user to enter a line of text. It therefore makes sense, for example, if the program needs to request a name from the user.
- An **IntField** makes it possible for the program to request an integer value from the user.
- A **DoubleField** makes it possible for the program to request a floating-point value from the user.

11. The default action command for a **JButton** is the label on the button.
12. If you add more than one interactor to the **SOUTH** border, those interactors are laid out horizontally from left to right in the control bar.
13. The **GObject** and **GCompound** classes stand in the same relation as **Component** and **Container**. In each case, the second class is an subclass of the first can contain instances of the more general class. Thus, a **GCompound** is a **GObject** that can contain instances of other **GObject**s, just as a **Container** is a **Component** that can contain instances of other **Component**s,
14. Java uses a *layout manager* to control the arrangement of components within a container.
15. The three most common layout managers in the **java.awt** package implement the following policies:

    - A **BorderLayout** manager places components in one of five positions—**NORTH**, **SOUTH**, **EAST**, **WEST**, and **CENTER**, any of which may be missing—within the surrounding container. The layout manager first assigns space to the **NORTH** and **SOUTH** components, giving each one its preferred space in the vertical direction but stretching it in the horizontal direction to fill the available space. It then assigns space to the **EAST** and **WEST** components in a similar way, stretching these components along the vertical axis instead. Once space has been assigned to the four borders components, the layout manager assigns the remaining space to the **CENTER** component.
    - A **FlowLayout** manager lays out each component in order from left to right at the top of the container. If it runs out of space in the window, the layout manager moves downward in the window to create a new line of interactors and continues to place interactors from left to right as before. Each line of interactors is then centered horizontally in its container.
    - A **GridLayout** manager lays out its components in a two-dimensional array of equal-sized areas that collectively fill the available space. The number of rows and columns is specified as part of the **GridLayout** constructor. Individual components under the control of a **GridLayout** manager are laid out from left to right in each successive row.

16. **NORTH**, **SOUTH**, **EAST**, **WEST**, and **CENTER**.
17. The top and bottom borders.
18. The **width** option specifies the minimum width in pixels for the current column. The **gridwidth** option specifies the number of columns that the current cell should occupy in the horizontal direction. Thus, the option **width=50** specifies that the current column should have a width of at least 50 pixels; the option **gridwidth=3** indicates that the current cell should stretch horizontally to fill three columns.

19. a)




b)


c)


20. A *package-private class* is a class that is accessible only within the package that defines it. Package-private classes have the same structure as the public classes except that they lack the **`public`** keyword.

## Chapter 11. Arrays and ArrayLists

1. An array is *ordered* and *homogeneous*.
2. An *element* of an array is one of its ordered components. The position number of an element in an array is called its *index*. The data type of each element within an array is the *element type* for the array; because arrays are homogeneous, the element types within a single array must always be the same. The *array length* is the number of elements allocated to an array. *Selection* is the process of identifying a particular array element by its index number.
3. a) `double[] doubleArray = new double[100];`
   b) `boolean[] inUse = new boolean[16];`
   c) `String[] lines = new String[50];`
4. ```
   int[] squares = new int[11];
   for (int i = 0; i < squares.length; i++) {
      squares[i] = i * i;
   }
   ```
5. Select its **`length`** field, as in **`squares.length`** in the preceding exercise.
6. The first approach is to use zero-based indices internally and then add 1 to the internal index whenever the index value is displayed to the user. The second is to allocate an extra element and then ignore element 0 of the array. The first strategy allows you to use methods that work with zero-based arrays directly; the second strategy is often easier to understand because the program and the user both work with the same set of indices.
   ```

7. The parentheses are necessary. Without them, Java would concatenate the value of **i** to the end of the string **"Score for judge "** and then concatenate the strings **"1"** and **": "** to the end of that result.

8. The prefix and postfix forms of the **++** operator have the same effect on the variable to which the operator is applied. In either case, the value of the variable is incremented by 1. The difference lies in the result returned to the surrounding expression. A prefix expression like **++x** returns the value of **x** *after* it is incremented; the suffix expression **x++** returns the value that **x** had before the incrementing occurred.

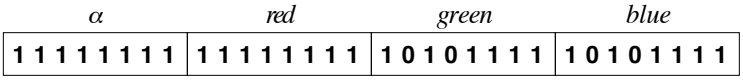
9. False. Arrays are represented in the same manner as objects, and not like primitive values.

10. The variable **temp** in the **swapElements** method holds the original value of one of the array elements being exchanged. If you store a value in an array element without first making sure that you have saved the old value, there is no way to complete the exchange operation because the original contents of that element are lost.

11. **int[] powersOfTwo = { 1, 2, 4, 8, 16, 32, 64, 128 };**

12. In Java, a multidimensional array is simply an array whose elements are other arrays. Conceptually, the elements of a multidimensional array are identified by several index numbers specifying, for example, the row and column of an element.

13. A pixel word has the following format:




14. **p & q** → **0xC00DAC**

    **p | q** → **0xD0FFEF**

    **p ^ q** → **0x10F243**

    **~p** → **0xFF3F0011**

15. Yes.

16. The **>>>** operator performs a *logical shift* in which the new bits that appear on the end are always **0**. The **>>** operator performs what computer scientists call an *arithmetic shift* in which the bit at the extreme left end of the word never changes.

17. **(Color.ORANGE.getRGB() >> 8) & 0xFF**

18. The primary advantage of the **ArrayList** class over the basic array structure is that it allows the programmer to add and delete elements without having to change the size of the array.

19. The **size** method returns the number of elements in an array and therefore corresponds to the **length** field in an array. In the simple form described in this chapter, the **add** method appends a new element to the end of an **ArrayList**, expanding the internal storage as necessary. The **set** and **get** methods make it possible to assign new values to elements or to retrieve existing values for a specific index. The **remove** method deletes an element from the array, shifting every subsequent value by one index position.

20. No. The type designator used in Java 5.0 must specify an object type, not a primitive type.

## Chapter 12. Searching and Sorting

1. *Searching* is the process of finding a particular value in an array. *Sorting* is the process of rearranging the elements of an array so that they appear in some well-defined order.
2. The only changes are in the method header line, where the parameter types need to be changed as follows:

   ```
   private int linearSearch(double key, double[] array)
   ```

   In Java, both versions of the method can coexist, because the Java compiler can tell which method applies by checking the types of the arguments.
3. The linear search algorithm goes through every element of an array in order to check for the key. The binary search algorithm applies only to sorted arrays and begins by checking the element that is at the halfway point of the array. If the key is smaller than that element, the key can occur only in the first half of the array. Similarly, if the key is larger than the middle element, the key must be in the second half. In either case, you can disregard half of the array and repeat the process on the remaining subarray.
4. True.
5. Binary search requires that the array be sorted.
6. The selection sort algorithm consists of two nested loops. The outer loop goes through each element position of the array in order from first to last. The inner loop then finds the smallest element between that position and the end of the array. Once the smallest element in that range has been identified, the selection sort algorithm proceeds by exchanging the smallest element with the value in the index position marked by the outer loop. After cycle *i* of the outer loop, the elements in the array up to position *i* are in the correct positions.
7. The method would still work correctly. Once the first `n` – 1 elements have been correctly positioned, the remaining element—which is now guaranteed to be the largest—must also be in the correct position.
8. `System.currentTimeMillis`
9. An algorithm is quadratic if the time required to execute it grows in proportion to the square of the size of the input.
10. The first sorting pass in the radix sort arranges the values in order according to their *last* digit. The next pass sorts on the next character position to the left, and the process continues until it reaches the first digit. Radix sort works because the algorithm ensures that the order within each of the buckets does not change from the order in the input to that pass.
11. $\frac{N^2 + N}{2}$
12. Computational complexity is a qualitative measure of the relationship between the size of a problem, typically denoted by *N,* and the execution time required to run the algorithm on a problem of that size.
13. The two most common simplifications you can use in big-O notation are:
    - You can throw away any terms that become insignificant as *N* becomes large.
    - You can ignore any constant factors.

14. Yes. The formula specified inside the big-O notation represents a bound that correctly describes the behavior of selection sort. That bound, however, is not as useful as the $O(N^2)$ traditionally used to describe selection sort, even though it is some sense more precise. Big-O notation is a *qualitative* measure that provides insight into the behavior of an algorithm. Saying that selection sort has $O(N^2)$ behavior offers greater insight than the seemingly more detailed formula.

15. In formal terms, the answer is again yes because $N^3$ grows more quickly than $N^2$. This description, however, makes selection sort seem even less efficient than it is.

16. As its name implies, *worst-case complexity* assesses the performance of an algorithm in the worst possible situation for a problem of a given size, as in trying to find the very last element in an array using the linear search algorithm. *Average-case complexity* assesses the average performance of an algorithm over all possible inputs. Because this averaging process often requires statistical techniques that are considerably more advanced, computing average-case performance is typically more difficult.

17. Linear search is $O(N)$, binary search is $O(\log N)$, selection sort is $O(N^2)$, and radix sort is $O(N \log N)$.

18. a) $O(N^2)$ b) $O(\log N)$ c) $O(1)$

19. The **`readLine`** method.

20. An *exception* is the mechanism that Java uses to report a condition that occurs outside the normal program flow.

21. To signal that an exception has occurred, one part of the code will *throw* an exception so that a higher level of the program can *catch* that exception and respond.

## Chapter 13. Collection Classes

1. As shown in the code for **`readLineArray`** in Figure 13-1, it is possible to implement he dynamic behavior of the **`ArrayList`** class by allocating an array that can be larger than the actual number of elements stored in the structure. Whenever the implementation runs out of space in the existing array, it can allocate a new array with a greater capacity, copy the elements from the old array into the new one, and then replacing the reference to the old array with the new one so that subsequent operations use the expanded array. Implementing this strategy necessitates keeping track of both the number of elements allocated in the array and the number of elements actually stored.

2. When implementing the dynamically expanding array strategy described in the answer to the preceding question, the term *capacity* is used to indicate the number of elements allocated for the array, and *effective size* refers to the number of elements actually in use.

3. The most important advantage of an interface is that several different classes can implement a common interface even if those classes are widely separated in Java's class hierarchy. Using interfaces therefore separates the *behavior* of a class defined by an interface from the *representation* of that class defined by its position in the class hierarchy.

4. The two operations that define a map are **`put(key, value)`**, which creates an association between the key and the value, and **`get(key)`**, which returns the current value associated with **`key`**.

5. True. The same value can appear many times in a map; the keys must be distinct.

6. The **hashCode** method is defined to return an **int**.
7. The pigeonhole principle says that if you place a certain number of objects in a smaller number of containers, there must be at least one container that contains more than one object.
8. A *collision* occurs whenever two keys in the **hashMap** return the same value for the **hashCode** method.
9. The **SimpleStringMap** implementation places all entries that has to the same key in a linked list of entries.
10. The implementation for the **hashCode** method in the **GPoint** class looks like this:

```
public int hashCode() {
   return new Float((float) xc).hashCode()
                    ^ (37 * new Float((float) yc).hashCode());
}
```

Many other implementations are possible. The important characteristics that should be true of any implementation include:

- The new hash code should be calculated using the hash codes of the individual elements, which in this case are the two components.
- The hash code must be compatible with the **equals** method. Although it might at first seem more reasonable to use **Double** rather than **Float** to calculate these hash codes, the **equals** method for **GPoint** uses single-precision floating-point values.
- The implementation should try to ensure that symmetries in the data are not replicated in the hash code. Multiplying one of the internal **hashCode** results by a prime number ensures that switching the *x* and *y* coordinate values will also change the hash code.

11. The **equals** method.
12. There are three chains that you can follow in Figure 13-5 to establish this fact. First, the **TreeSet** class is shown to be a subclass of **AbstractSet**, which is in turn a subclass of **AbstractCollection**, which implements **Collection**. The second path notes starts by following the class hierarchy arrow from **TreeSet** to **AbstractSet**, but then uses the fact that **AbstractSet** implements the **Set** interface, which is a subinterface of **Collection**. The third path traces **TreeSet**to the **SortedSet** interface and then follows the subinterface hierarchy to **Set** and **Collection**.
13. An *iterator* is an instance of a class that implements tha **Iterator** interface. This interface specifies the methods **hasNext** and **next**, which together make it possible to step through the elements of a collection one at a time.
14. Java 5.0 introduces the following syntax for working with iterators:

```
for (type variable : collection) {
   statements
}
```

This shorthand is equivalent to the following expanded code:

```
Iterator<type> iterator = collection.iterator();
while (iterator.hasNext()) {
   type variable = iterator.next()
   statements
}
```

15. If Java 5.0 is not available, it is still possible to make use of the Java Collection Framework, although doing so is not as convenient. The most significant change is that older versions of Java do not include generics, which means that the collection classes all use **`Object`** as their element type. When you take a value out of a collection without a type parameter, you usually have to cast the **`Object`** to the type you in fact need. A second difference is that older versions of Java do not support boxing and unboxing, which means that you need to use wrapper classes whenever you want to store primitive values.
16. The **`Comparable`** interface describes a class that supports comparison on its own instances. The **`Comparator`** interface denotes a class that is capable of comparing two values of some other class, which is typically specified by including type parameters with the reference to the **`Comparator`** interface.
17. The **`Collection`** interface is the root of the interface hierarchy for the Java Collections Framework. Any class that implements the **`Collection`** interface supports the basic operations that apply to collections of any type. The **`Collections`** class contains a set of static methods that perform useful operations on collections.
18. This chapter argues that a well-designed Java package must be *unified, simple, sufficient, flexible,* and *stable*.

## Chapter 14. Looking Ahead

1. A recursive process is one that operates by breaking a large problem down into simpler subproblems of the same form until those subproblems become simple enough to solve without further decomposition.
2. The **`mystery`** method computes the function $2^n$.
3. Implementing a recursive strategy require you to identify *simple cases* that can be solved easily and a *recursive decomposition* that allows you to break the problem down into simpler cases of the same form.
4. The phrase *recursive leap of faith* refers to the essential programming strategy of believing—without going through the details—that recursive calls will work correctly as long as the subproblem they solve is simpler than the original.
5. Unlike *processes* that run in distinct execution environments, *threads* run concurrently within a single program and are therefore able to share access to the instance variables of the object to which those threads belong.
6. The **`java.net`** package contains most of the classes used in network-based programming.
7. In the world of object-oriented programming, the "Gang of Four" consists of Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, who wrote a pathbreaking book entitled *Design Patterns* (Addison-Wesley, 1995).
8. In the model/view/controller pattern, the *model* is responsible for maintaining the abstract state of some data structure for some application while remaining entirely separate from the details of the user interface. Responsibility for the user interface is split between the *view,* which takes care of how the information is presented to the user, and the *controller,* which allows the user to manipulate that information.