Appendix A

Solutions to Exercises

Each of Chapters 1 through 14 closes with an "Exercises" section that tests your understanding of the chapter's material. Solutions to these exercises are presented in this appendix.

Chapter 1: Getting Started with Java

1. Java is a language and a platform. The language is partly patterned after the C and C++ languages to shorten the learning curve for C/C++ developers. The platform consists of a virtual machine and associated execution environment.
2. A virtual machine is a software-based processor that presents its own instruction set.
3. The purpose of the Java compiler is to translate source code into instructions (and associated data) that are executed by the virtual machine.
4. The answer is true: a classfile's instructions are commonly referred to as bytecode.
5. When the virtual machine's interpreter learns that a sequence of bytecode instructions is being executed repeatedly, it informs the virtual machine's Just In Time (JIT) compiler to compile these instructions into native code.
6. The Java platform promotes portability by providing an abstraction over the underlying platform. As a result, the same bytecode runs unchanged on Windows-based, Linux-based, Mac OS X–based, and other platforms.
7. The Java platform promotes security by providing a secure environment in which code executes. It accomplishes this task in part by using a bytecode verifier to make sure that the classfile's bytecode is valid.

8. The answer is false: Java SE is the platform for developing applications and applets.
9. The JRE implements the Java SE platform and makes it possible to run Java programs.
10. The difference between the public and private JREs is that the public JRE exists apart from the JDK, whereas the private JRE is a component of the JDK that makes it possible to run Java programs independently of whether or not the public JRE is installed.
11. The JDK provides development tools (including a compiler) for developing Java programs. It also provides a private JRE for running these programs.
12. The JDK's `javac` tool is used to compile Java source code.
13. The JDK's `java` tool is used to run Java applications.
14. Standard I/O is a mechanism consisting of Standard Input, Standard Output, and Standard Error that makes it possible to read text from different sources (keyboard or file), write nonerror text to different destinations (screen or file), and write error text to different destinations (screen or file).
15. You specify the `main()` method's header as `public static void main(String[] args)`.
16. An IDE is a development framework consisting of a project manager for managing a project's files, a text editor for entering and editing source code, a debugger for locating bugs, and other features. The IDE that Google supports for developing Android apps is Eclipse.

Chapter 2: Learning Language Fundamentals

1. Unicode is a computing industry standard for consistently encoding, representing, and handling text that's expressed in most of the world's writing systems.
2. A comment is a language feature for embedding documentation in source code.
3. The three kinds of comments that Java supports are single-line, multiline, and Javadoc.
4. An identifier is a language feature that consists of letters (A–Z, a–z, or equivalent uppercase/lowercase letters in other human alphabets), digits (0–9 or equivalent digits in other human alphabets), connecting punctuation characters (e.g., the underscore), and currency symbols (e.g., the dollar sign $). This name must begin with a letter, a currency symbol, or a connecting punctuation character; and its length cannot exceed the line in which it appears.

5. The answer is false: Java is a case-sensitive language.
6. A type is a language feature that identifies a set of values (and their representation in memory) and a set of operations that transform these values into other values of that set.
7. A primitive type is a type that's defined by the language and whose values are not objects.
8. Java supports the Boolean, character, byte integer, short integer, integer, long integer, floating-point, and double precision floating-point primitive types.
9. A user-defined type is a type that's defined by the developer using a class, an interface, an enum, or an annotation type and whose values are objects.
10. An array type is a special reference type that signifies an array, a region of memory that stores values in equal-size and contiguous slots, which are commonly referred to as elements.
11. A variable is a named memory location that stores some type of value.
12. An expression is a combination of literals, variable names, method calls, and operators. At runtime, it evaluates to a value whose type is referred to as the expression's type.
13. The two expression categories are simple expression and compound expression.
14. A literal is a value specified verbatim.
15. String literal `"The quick brown fox \jumps\ over the lazy dog."` is illegal because, unlike `\"`, `\j` and `\` (a backslash followed by a space character) are not valid escape sequences. To make this string literal legal, you must escape these backslashes, as in `"The quick brown fox \\jumps\\ over the lazy dog."`.
16. An operator is a sequence of instructions symbolically represented in source code.
17. The difference between a prefix operator and a postfix operator is that a prefix operator precedes its operand and a postfix operator trails its operand.
18. The purpose of the cast operator is to convert from one type to another type. For example, you can use this operator to convert from floating-point type to 32-bit integer type.
19. Precedence refers to an operator's level of importance.
20. The answer is true: most of Java's operators are left-to-right associative.
21. A statement is a language feature that assigns a value to a variable, controls a program's flow by making a decision and/or repeatedly executing another statement, or performs another task.

22. The while statement evaluates its Boolean expression at the top of the loop, whereas the do-while statement evaluates its Boolean expression at the bottom of the loop. As a result, while executes zero or more times, whereas do-while executes one or more times.

23. The difference between the break and continue statements is that break transfers execution to the first statement following a switch statement or a loop, whereas continue skips the remainder of the current loop iteration, reevaluates the loop's Boolean expression, and performs another iteration (when true) or terminates the loop (when false).

24. Listing A-1 presents an `OutputGradeLetter` application (the class is named `OutputGradeLetter`) whose `main()` method executes the grade letter code sequence presented while discussing the if-else statement.

Listing A-1. Classifying a Grade

```
public class OutputGradeLetter
{
   public static void main(String[] args)
   {
      char gradeLetter = 'u'; // unknown
      int testMark = 100;

      if (testMark >= 90)
      {
         gradeLetter = 'A';
         System.out.println("You aced the test.");
      }
      else
      if (testMark >= 80)
      {
         gradeLetter = 'B';
         System.out.println("You did very well on this test.");
      }
      else
      if (testMark >= 70)
      {
         gradeLetter = 'C';
         System.out.println("Not bad, but you need to study more for future tests.");
      }
      else
      if (testMark >= 60)
      {
         gradeLetter = 'D';
         System.out.println("Your test result suggests that you need a tutor.");
      }
```

```
      else
      {
         gradeLetter = 'F';
         System.out.println("Your test result is pathetic; you need summer school.");
      }
   }
}
```

25. Listing A-2 presents a `Triangle` application whose `main()` method uses a pair of nested for statements along with `System.out.print()` to output a 10-row triangle of asterisks, where each row contains an odd number of asterisks (1, 3, 5, 7, and so on).

Listing A-2. Printing a Triangle of Asterisks

```
public class Triangle
{
   public static void main(String[] args)
   {
      for (int row = 1; row < 20; row += 2)
      {
         for (int col = 0; col < 19 - row / 2; col++)
            System.out.print(" ");
         for (int col = 0; col < row; col++)
            System.out.print("*");
         System.out.print('\n');
      }
   }
}
```

Chapter 3: Discovering Classes and Objects

1. A class is a template for manufacturing objects.
2. You declare a class by providing a header followed by a body. The header minimally consists of reserved word `class` followed by an identifier. The body consists of a sequence of declarations placed between a pair of brace characters.
3. An object is a named aggregate of code and data.
4. You instantiate an object by using the `new` operator followed by a constructor.
5. A constructor is a block of code for constructing an object by initializing it in some manner.
6. The answer is true: Java creates a default noargument constructor when a class declares no constructors.

7. A parameter list is a round bracket-delimited and comma-separated list of zero or more parameter declarations. A parameter is a constructor or method variable that receives an expression value passed to the constructor or method when it is called.

8. An argument list is a round bracket-delimited and comma-separated list of zero or more expressions. An argument is one of these expressions whose value is passed to the corresponding parameter when a constructor or method variable is called.

9. The answer is false: you invoke another constructor by specifying `this` followed by an argument list.

10. Arity is the number of arguments passed to a constructor or method or the number of operator operands.

11. A local variable is a variable that is declared in a constructor or method and is not a member of the constructor or method parameter list.

12. Lifetime is a property of a variable that determines how long the variable exists. For example, local variables and parameters come into existence when a constructor or method is called and are destroyed when the constructor or method finishes. Similarly, an instance field comes into existence when an object is created and is destroyed when the object is garbage collected.

13. Scope is a property of a variable that determines how accessible the variable is to code. For example, a parameter can be accessed only by the code within the constructor or method in which the parameter is declared.

14. Encapsulation refers to the merging of state and behaviors into a single source code entity. Instead of separating state and behaviors, which is done in structured programs, state and behaviors are combined into classes and objects, which are the focus of object-based programs. For example, whereas a structured program makes you think in terms of separate balance state and deposit/withdraw behaviors, an object-based program makes you think in terms of bank accounts, which unite balance state with deposit/withdraw behaviors through encapsulation.

15. A field is a variable declared within a class body.

16. The difference between an instance field and a class field is that an instance field describes some attribute of the real-world entity that an object is modeling and is unique to each object, and a class field identifies some data item that is shared by all objects.

17. A blank final is a read-only instance field. It differs from a true constant in that there are multiple copies of blank finals (one per object) and only one true constant (one per class).

18. You prevent a field from being shadowed by changing the name of a same-named local variable or parameter or by qualifying the local variable's name or parameter's name with `this` or the class name followed by the member access operator.

19. A method is a named block of code declared within a class body.

20. The difference between an instance method and a class method is that an instance method describes some behavior of the real-world entity that an object is modeling and can access a specific object's state, and a class method identifies some behavior that is common to all objects and cannot access a specific object's state.

21. Recursion is the act of a method invoking itself.

22. You overload a method by introducing a method with the same name as an existing method but with a different parameter list into the same class.

23. A class initializer is a `static`-prefixed block that is introduced into a class body. An instance initializer is a block that is introduced into a class body as opposed to being introduced as the body of a method or a constructor.

24. A garbage collector is code that runs in the background and occasionally checks for unreferenced objects.

25. The answer is false: `String[] letters = new String[2] { "A", "B" };` is incorrect syntax. Remove the `2` from between the square brackets to make it correct.

26. A ragged array is a two-dimensional array in which each row can have a different number of columns.

27. Calculating the greatest common divisor of two positive integers, which is the greatest positive integer that divides evenly into both positive integers, provides another example of tail recursion. Listing A-3 presents the source code.

Listing A-3. Recursively Calculating the Greatest Common Divisor

```
public static int gcd(int a, int b)
{
   // The greatest common divisor is the largest positive integer that
   // divides evenly into two positive integers a and b. For example,
   // GCD(12, 18) is 6.

   if (b == 0) // Base problem
      return a;
   else
      return gcd(b, a % b);
}
```

28. Listing A-4 presents the source code to a `Book` class with name, author, and International Standard Book Number (ISBN) fields and a suitable constructor and getter methods that return field values. Furthermore, a `main()` method is present that creates an array of `Book` objects and iterates over this array outputting each book's name, author, and ISBN.

Listing A-4. Building a Library of Books

```
public class Book
{
   private String name;
   private String author;
   private String isbn;

   public Book(String name, String author, String isbn)
   {
      this.name = name;
      this.author = author;
      this.isbn = isbn;
   }

   public String getName()
   {
      return name;
   }

   public String getAuthor()
   {
      return author;
   }

   public String getISBN()
   {
      return isbn;
   }

   public static void main(String[] args)
   {
      Book[] books = new Book[]
                     {
                        new Book("Jane Eyre",
                                 "Charlotte Brontë",
                                 "0895772000"),
                        new Book("A Kick in the Seat of the Pants",
                                 "Roger von Oech",
                                 "0060155280"),
                        new Book("The Prince and the Pilgrim",
                                 "Mary Stewart",
                                 "0340649925")
                     };
```

```
        for (int i = 0; i < books.length; i++)
           System.out.println(books[i].getName() + " - " +
                              books[i].getAuthor() + " - " +
                              books[i].getISBN());
     }
}
```

Chapter 4: Discovering Inheritance, Polymorphism, and Interfaces

1. Implementation inheritance is inheritance through class extension.
2. Java supports implementation inheritance by providing reserved word `extends`.
3. A subclass can have only one superclass because Java doesn't support multiple implementation inheritance.
4. You prevent a class from being subclassed by declaring the class `final`.
5. The answer is false: the `super()` call can only appear in a constructor.
6. If a superclass declares a constructor with one or more parameters, and if a subclass constructor doesn't use `super()` to call that constructor, the compiler reports an error because the subclass constructor attempts to call a nonexistent noargument constructor in the superclass. (When a class doesn't declare any constructors, the compiler creates a constructor with no parameters [a noargument constructor] for that class. Therefore, if the superclass didn't declare any constructors, a noargument constructor would be created for the superclass. Continuing, if the subclass constructor didn't use `super()` to call the superclass constructor, the compiler would insert the call and there would be no error.)
7. An immutable class is a class whose instances cannot be modified.
8. The answer is false: a class cannot inherit constructors.
9. Overriding a method means to replace an inherited method with another method that provides the same signature and the same return type but provides a new implementation.
10. To call a superclass method from its overriding subclass method, prefix the superclass method name with reserved word `super` and the member access operator in the method call.
11. You prevent a method from being overridden by declaring the method `final`.

12. You cannot make an overriding subclass method less accessible than the superclass method it is overriding because subtype polymorphism would not work properly if subclass methods could be made less accessible. Suppose you upcast a subclass instance to superclass type by assigning the instance's reference to a variable of superclass type. Now suppose you specify a superclass method call on the variable. If this method is overridden by the subclass, the subclass version of the method is called. However, if access to the subclass's overriding method's access could be made private, calling this method would break encapsulation—private methods cannot be called directly from outside of their class.

13. You tell the compiler that a method overrides another method by prefixing the overriding method's header with the `@Override` annotation.

14. Java doesn't support multiple implementation inheritance because this form of inheritance can lead to ambiguities.

15. The name of Java's ultimate superclass is `Object`. This class is located in the `java.lang` package.

16. The purpose of the `clone()` method is to duplicate an object without calling a constructor.

17. `Object`'s `clone()` method throws `CloneNotSupportedException` when the class whose instance is to be shallowly cloned doesn't implement the `Cloneable` interface.

18. The difference between shallow copying and deep copying is that shallow copying copies each primitive or reference field's value to its counterpart in the clone, whereas deep copying creates, for each reference field, a new object and assigns its reference to the field. This deep copying process continues recursively for these newly created objects.

19. The `==` operator cannot be used to determine if two objects are logically equivalent because this operator only compares object references and not the contents of these objects.

20. `Object`'s `equals()` method compares the current object's `this` reference to the reference passed as an argument to this method. (When I refer to `Object`'s `equals()` method, I am referring to the `equals()` method in the `Object` class.)

21. Expression `"abc" == "a" + "bc"` returns true. It does so because the `String` class contains special support that allows literal strings and string-valued constant expressions to be compared via `==`.

22. You can optimize a time-consuming `equals()` method by first using `==` to determine if this method's reference argument identifies the current object (which is represented in source code via reserved word `this`).

23. The purpose of the `finalize()` method is to provide a safety net for calling an object's cleanup method in case that method is not called.

24. You should not rely on `finalize()` for closing open files because file descriptors are a limited resource and an application might not be able to open additional files until `finalize()` is called, and this method might be called infrequently (or perhaps not at all).

25. A hash code is a small value that results from applying a mathematical function to a potentially large amount of data.

26. The answer is true: you should override the `hashCode()` method whenever you override the `equals()` method.

27. `Object`'s `toString()` method returns a string representation of the current object that consists of the object's class name, followed by the `@` symbol, followed by a hexadecimal representation of the object's hash code. (When I refer to `Object`'s `toString()` method, I am referring to the `toString()` method in the `Object` class.)

28. You should override `toString()` to provide a concise but meaningful description of the object to facilitate debugging via `System.out.println()` method calls. It is more informative for `toString()` to reveal object state than to reveal a class name, followed by the `@` symbol, followed by a hexadecimal representation of the object's hash code.

29. Composition is a way to reuse code by composing classes out of other classes based on a "has-a" relationship between them.

30. The answer is false: composition is used to describe "has-a" relationships and implementation inheritance is used to describe "is-a" relationships.

31. The fundamental problem of implementation inheritance is that it breaks encapsulation. You fix this problem by ensuring that you have control over the superclass as well as its subclasses by ensuring that the superclass is designed and documented for extension or by using a wrapper class in lieu of a subclass when you would otherwise extend the superclass.

32. Subtype polymorphism is a kind of polymorphism where a subtype instance appears in a supertype context, and executing a supertype operation on the subtype instance results in the subtype's version of that operation executing.

33. Subtype polymorphism is accomplished by upcasting the subtype instance to its supertype; by assigning the instance's reference to a variable of that type; and, via this variable, calling a superclass method that has been overridden in the subclass.

34. You would use abstract classes and abstract methods to describe generic concepts (e.g., shape, animal, or vehicle) and generic operations (e.g., drawing a generic shape). Abstract classes cannot be instantiated and abstract methods cannot be called because they have no code bodies.

35. An abstract class can contain concrete methods.

36. The purpose of downcasting is to access subtype features. For example, you would downcast a `Point` variable that contains a `Circle` instance reference to the `Circle` type so that you can call `Circle`'s `getRadius()` method on the instance.

37. Two forms of RTTI are the virtual machine verifying that a cast is legal and using the `instanceof` operator to determine whether or not an instance is a member of a type.

38. A covariant return type is a method return type that, in the superclass's method declaration, is the supertype of the return type in the subclass's overriding method declaration.

39. You formally declare an interface by specifying at least reserved word `interface`, followed by a name, followed by a brace-delimited body of constants and/or method headers.

40. The answer is true: you can precede an interface declaration with the `abstract` reserved word. However, doing so is redundant.

41. A marker interface is an interface that declares no members.

42. Interface inheritance is inheritance through interface implementation or interface extension.

43. You implement an interface by appending an implements clause, consisting of reserved word `implements` followed by the interface's name, to a class header and by overriding the interface's method headers in the class.

44. You might encounter one or more name collisions when you implement multiple interfaces.

45. You form a hierarchy of interfaces by appending reserved word `extends` followed by an interface name to an interface header.

46. Java's interfaces feature is so important because it gives developers the utmost flexibility in designing their applications.

47. Interfaces and abstract classes describe abstract types.

48. Interfaces and abstract classes differ in that interfaces can only declare abstract methods and constants and can be implemented by any class in any class hierarchy. In contrast, abstract classes can declare constants and nonconstant fields; can declare abstract and concrete methods; and can only appear in the upper levels of class hierarchies, where they are used to describe abstract concepts and behaviors.

49. Listings A-5 through A-11 declare the `Animal`, `Bird`, `Fish`, `AmericanRobin`, `DomesticCanary`, `RainbowTrout`, and `SockeyeSalmon` classes that were called for in Chapter 4.

Listing A-5. The `Animal` Class Abstracting Over Birds and Fish (and Other Organisms)

```
public abstract class Animal
{
   private String kind;
   private String appearance;

   public Animal(String kind, String appearance)
   {
      this.kind = kind;
      this.appearance = appearance;
   }

   public abstract void eat();

   public abstract void move();

   @Override
   public final String toString()
   {
      return kind + " -- " + appearance;
   }
}
```

Listing A-6. The `Bird` Class Abstracting Over American Robins, Domestic Canaries, and Other Kinds of Birds

```
public abstract class Bird extends Animal
{
   public Bird(String kind, String appearance)
   {
      super(kind, appearance);
   }

   @Override
   public final void eat()
   {
      System.out.println("eats seeds and insects");
   }

   @Override
   public final void move()
   {
      System.out.println("flies through the air");
   }
}
```

Listing A-7. The `Fish` Class Abstracting Over Rainbow Trout, Sockeye Salmon, and Other Kinds of Fish

```
public abstract class Fish extends Animal
{
   public Fish(String kind, String appearance)
   {
      super(kind, appearance);
   }

   @Override
   public final void eat()
   {
      System.out.println("eats krill, algae, and insects");
   }

   @Override
   public final void move()
   {
      System.out.println("swims through the water");
   }
}
```

Listing A-8. The `AmericanRobin` Class Denoting a Bird with a Red Breast

```
public final class AmericanRobin extends Bird
{
   public AmericanRobin()
   {
      super("americanrobin", "red breast");
   }
}
```

Listing A-9. The `DomesticCanary` Class Denoting a Bird of Various Colors

```
public final class DomesticCanary extends Bird
{
   public DomesticCanary()
   {
      super("domestic canary", "yellow, orange, black, brown, white, red");
   }
}
```

Listing A-10. The `RainbowTrout` Class Denoting a Rainbow-Colored Fish

```
public final class RainbowTrout extends Fish
{
   public RainbowTrout()
   {
      super("rainbowtrout", "bands of brilliant speckled multicolored " +
            "stripes running nearly the whole length of its body");
   }
}
```

Listing A-11. The `SockeyeSalmon` Class Denoting a Red-and-Green Fish

```
public final class SockeyeSalmon extends Fish
{
   public SockeyeSalmon()
   {
      super("sockeyesalmon", "bright red with a green head");
   }
}
```

`Animal`'s `toString()` method is declared `final` because it doesn't make sense to override this method, which is complete in this example. Also, each of `Bird`'s and `Fish`'s overriding `eat()` and `move()` methods is declared `final` because it doesn't make sense to override these methods in this example, which assumes that all birds eat seeds and insects; all fish eat krill, algae, and insects; all birds fly through the air; and all fish swim through the water.

The `AmericanRobin`, `DomesticCanary`, `RainbowTrout`, and `SockeyeSalmon` classes are declared `final` because they represent the bottom of the `Bird` and `Fish` class hierarchies, and it doesn't make sense to subclass them.

50. Listing A-12 declares the `Animals` class that was called for in Chapter 4.

Listing A-12. The `Animals` Class Letting Animals Eat and Move

```
public class Animals
{
   public static void main(String[] args)
   {
      Animal[] animals = { new AmericanRobin(), new RainbowTrout(),
                           new DomesticCanary(), new SockeyeSalmon() };
      for (int i = 0; i < animals.length; i++)
      {
         System.out.println(animals[i]);
         animals[i].eat();
         animals[i].move();
         System.out.println();
      }
   }
}
```

51. Listings A-13 through A-15 declare the `Countable` interface, the modified `Animal` class, and the modified `Animals` class that were called for in Chapter 4.

Listing A-13. The `Countable` Interface for Use in Taking a Census of Animals

```
public interface Countable
{
   String getID();
}
```

Listing A-14. The Refactored `Animal` Class for Help in Census Taking

```
public abstract class Animal implements Countable
{
   private String kind;
   private String appearance;

   public Animal(String kind, String appearance)
   {
      this.kind = kind;
      this.appearance = appearance;
   }

   public abstract void eat();

   public abstract void move();

   @Override
   public final String toString()
   {
      return kind + " -- " + appearance;
   }

   @Override
   public final String getID()
   {
      return kind;
   }
}
```

Listing A-15. The Modified `Animals` Class for Carrying Out the Census

```
public class Animals
{
   public static void main(String[] args)
   {
      Animal[] animals = { new AmericanRobin(), new RainbowTrout(),
                           new DomesticCanary(), new SockeyeSalmon(),
                           new RainbowTrout(), new AmericanRobin() };
      for (int i = 0; i < animals.length; i++)
      {
         System.out.println(animals[i]);
         animals[i].eat();
         animals[i].move();
         System.out.println();
      }

      Census census = new Census();
      Countable[] countables = (Countable[]) animals;
      for (int i = 0; i < countables.length; i++)
         census.update(countables[i].getID());
```

```
        for (int i = 0; i < Census.SIZE; i++)
            System.out.println(census.get(i));
    }
}
```

Chapter 5: Mastering Advanced Language Features Part 1

1. A nested class is a class that is declared as a member of another class or scope.
2. The four kinds of nested classes are static member classes, nonstatic member classes, anonymous classes, and local classes.
3. Nonstatic member classes, anonymous classes, and local classes are also known as inner classes.
4. The answer is false: a static member class doesn't have an enclosing instance.
5. You instantiate a nonstatic member class from beyond its enclosing class by first instantiating the enclosing class and then prefixing the `new` operator with the enclosing class instance as you instantiate the enclosed class. Example: `new EnclosingClass().new EnclosedClass()`.
6. It's necessary to declare local variables and parameters `final` when they are being accessed by an instance of an anonymous class or a local class.
7. The answer is true: an interface can be declared within a class or within another interface.
8. A package is a unique namespace that can contain a combination of top-level classes, other top-level types, and subpackages.
9. You ensure that package names are unique by specifying your reversed Internet domain name as the top-level package name.
10. A package statement is a statement that identifies the package in which a source file's types are located.
11. The answer is false: you cannot specify multiple package statements in a source file.
12. An import statement is a statement that imports types from a package by telling the compiler where to look for unqualified type names during compilation.
13. You indicate that you want to import multiple types via a single import statement by specifying the wildcard character (*).
14. During a runtime search, the virtual machine reports a "no class definition found" error when it cannot find a classfile.

15. You specify the user classpath to the virtual machine via the `-classpath` option used to start the virtual machine or, when not present, the `CLASSPATH` environment variable.

16. A constant interface is an interface that only exports constants.

17. Constant interfaces are used to avoid having to qualify their names with their classes.

18. Constant interfaces are bad because their constants are nothing more than an implementation detail that should not be allowed to leak into the class's exported interface because they might confuse the class's users (what is the purpose of these constants?). Also, they represent a future commitment: even when the class no longer uses these constants, the interface must remain to ensure binary compatibility.

19. A static import statement is a version of the import statement that lets you import a class's static members so that you don't have to qualify them with their class names.

20. You specify a static import statement as `import`, followed by `static`, followed by a member access operator–separated list of package and subpackage names, followed by the member access operator, followed by a class's name, followed by the member access operator, followed by a single static member name or the asterisk wildcard, for example, `import static java.lang.Math.cos;` (import the `cos()` static method from the `Math` class).

21. An exception is a divergence from an application's normal behavior.

22. Objects are superior to error codes for representing exceptions because error code Boolean or integer values are less meaningful than object names and because objects can contain information about what led to the exception. These details can be helpful to a suitable workaround. Furthermore, error codes are easy to ignore.

23. A throwable is an instance of `Throwable` or one of its subclasses.

24. The `getCause()` method returns an exception that is wrapped inside another exception.

25. `Exception` describes exceptions that result from external factors (e.g., not being able to open a file) and from flawed code (e.g., passing an illegal argument to a method). `Error` describes virtual machine-oriented exceptions such as running out of memory or being unable to load a classfile.

26. A checked exception is an exception that represents a problem with the possibility of recovery and for which the developer must provide a workaround.

27. A runtime exception is an exception that represents a coding mistake.

28. You would introduce your own exception class when no existing exception class in the standard class library meets your needs.

29. The answer is false: you use a throws clause to identify exceptions that are thrown from a method by appending this clause to a method's header.

30. The purpose of a try statement is to provide a scope (via its brace-delimited body) in which to present code that can throw exceptions. The purpose of a catch block is to receive a thrown exception and provide code (via its brace-delimited body) that handles that exception by providing a workaround.

31. The purpose of a finally block is to provide cleanup code that is executed whether an exception is thrown or not.

32. Listing A-16 presents the `G2D` class that was called for in Chapter 5.

Listing A-16. The `G2D` Class with Its `Matrix` Nonstatic Member Class

```
public class G2D
{
   private Matrix xform;

   public G2D()
   {
      xform = new Matrix();
      xform.a = 1.0;
      xform.e = 1.0;
      xform.i = 1.0;
   }

   private class Matrix
   {
      double a, b, c;
      double d, e, f;
      double g, h, i;
   }
}
```

33. To extend the `logging` package (presented in Chapter 5's discussion of packages) to support a null device in which messages are thrown away, first introduce Listing A-17's `NullDevice` package-private class.

Listing A-17. Implementing the Proverbial "Bit Bucket" Class

```
package logging;

class NullDevice implements Logger
{
   private String dstName;
```

```
   NullDevice(String dstName)
   {
   }

   public boolean connect()
   {
      return true;
   }

   public boolean disconnect()
   {
      return true;
   }

   public boolean log(String msg)
   {
      return true;
   }
}
```

Continue by introducing, into the `LoggerFactory` class, a `NULLDEVICE` constant and code that instantiates `NullDevice` with a `null` argument—a destination name is not required—when `newLogger()`'s `dstType` parameter contains this constant's value. Check out Listing A-18.

Listing A-18. A Refactored `LoggerFactory` Class

```
package logging;

public abstract class LoggerFactory
{
   public final static int CONSOLE = 0;
   public final static int FILE = 1;
   public final static int NULLDEVICE = 2;

   public static Logger newLogger(int dstType, String...dstName)
   {
      switch (dstType)
      {
         case CONSOLE   : return new Console(dstName.length == 0 ? null
                                                                 : dstName[0]);
         case FILE      : return new File(dstName.length == 0 ? null
                                                              : dstName[0]);
         case NULLDEVICE: return new NullDevice(null);
         default        : return null;
      }
   }
}
```

34. Modifying the `logging` package (presented in Chapter 5's discussion of packages) so that `Logger`'s `connect()` method throws a `CannotConnectException` instance when it cannot connect to its logging destination, and the other two methods each throw a `NotConnectedException` instance when `connect()` was not called or when it threw a `CannotConnectException` instance, results in Listing A-19's `Logger` interface.

Listing A-19. A `Logger` Interface Whose Methods Throw Exceptions

```
package logging;

public interface Logger
{
   void connect() throws CannotConnectException;
   void disconnect() throws NotConnectedException;
   void log(String msg) throws NotConnectedException;
}
```

Listing A-20 presents the `CannotConnectException` class.

Listing A-20. An Uncomplicated `CannotConnectException` Class

```
package logging;

public class CannotConnectException extends Exception
{
}
```

The `NotConnectedException` class has the same structure but with a different name.

Listing A-21 presents the `Console` class.

Listing A-21. The `Console` Class Satisfying `Logger`'s Contract Without Throwing Exceptions

```
package logging;

class Console implements Logger
{
   private String dstName;

   Console(String dstName)
   {
      this.dstName = dstName;
   }

   public void connect() throws CannotConnectException
   {
   }

   public void disconnect() throws NotConnectedException
   {
   }
```

```
   public void log(String msg) throws NotConnectedException
   {
      System.out.println(msg);
   }
}
```

Listing A-22 presents the `File` class.

Listing A-22. The `File` Class Satisfying `Logger`'s Contract by Throwing Exceptions As Necessary

```
package logging;

class File implements Logger
{
   private String dstName;

   File(String dstName)
   {
      this.dstName = dstName;
   }

   public void connect() throws CannotConnectException
   {
      if (dstName == null)
          throw new CannotConnectException();
   }

   public void disconnect() throws NotConnectedException
   {
      if (dstName == null)
          throw new NotConnectedException();
   }

   public void log(String msg) throws NotConnectedException
   {
      if (dstName == null)
          throw new NotConnectedException();
      System.out.println("writing " + msg + " to file " + dstName);
   }
}
```

35. When you modify `TestLogger` to respond appropriately to thrown `CannotConnectException` and `NotConnectedException` objects, you end up with something similar to Listing A-23.

Listing A-23. A `TestLogger` Class That Handles Thrown Exceptions

```
import logging.*;

public class TestLogger
{
   public static void main(String[] args)
```

```
    {
        try
        {
            Logger logger = LoggerFactory.newLogger(LoggerFactory.CONSOLE);
            logger.connect();
            logger.log("test message #1");
            logger.disconnect();
        }
        catch (CannotConnectException cce)
        {
            System.err.println("cannot connect to console-based logger");
        }
        catch (NotConnectedException nce)
        {
            System.err.println("not connected to console-based logger");
        }

        try
        {
            Logger logger = LoggerFactory.newLogger(LoggerFactory.FILE, "x.txt");
            logger.connect();
            logger.log("test message #2");
            logger.disconnect();
        }
        catch (CannotConnectException cce)
        {
            System.err.println("cannot connect to file-based logger");
        }
        catch (NotConnectedException nce)
        {
            System.err.println("not connected to file-based logger");
        }

        try
        {
            Logger logger = LoggerFactory.newLogger(LoggerFactory.FILE);
            logger.connect();
            logger.log("test message #3");
            logger.disconnect();
        }
        catch (CannotConnectException cce)
        {
            System.err.println("cannot connect to file-based logger");
        }
        catch (NotConnectedException nce)
        {
            System.err.println("not connected to file-based logger");
        }
    }
}
```

Chapter 6: Mastering Advanced Language Features Part 2

1. An assertion is a statement that lets you express an assumption of program correctness via a Boolean expression.
2. You would use assertions to validate internal invariants, control-flow invariants, preconditions, postconditions, and class invariants.
3. The answer is false: specifying the `-ea` command-line option with no argument enables all assertions except for system assertions.
4. An annotation is an instance of an annotation type and associates metadata with an application element. It's expressed in source code by prefixing the type name with the `@` symbol.
5. Constructors, fields, local variables, methods, packages, parameters, and types (annotation, class, enum, and interface) can be annotated.
6. The three compiler-supported annotation types are `Override`, `Deprecated`, and `SuppressWarnings`.
7. You declare an annotation type by specifying the `@` symbol, immediately followed by reserved word `interface`, followed by the type's name, followed by a body.
8. A marker annotation is an instance of an annotation type that supplies no data apart from its name—the type's body is empty.
9. An element is a method header that appears in the annotation type's body. It cannot have parameters or a throws clause. Its return type must be primitive (e.g., `int`), `String`, `Class`, an enum type, an annotation type, or an array of the preceding types. It can have a default value.
10. You assign a default value to an element by specifying `default` followed by the value, whose type must match the element's return type. For example, `String developer() default "unassigned";`.
11. A meta-annotation is an annotation that annotates an annotation type.
12. Java's four meta-annotation types are `Target`, `Retention`, `Documented`, and `Inherited`.
13. Generics can be defined as a suite of language features for declaring and using type-agnostic classes and interfaces.
14. You would use generics to ensure that your code is typesafe by avoiding thrown `ClassCastException`s.
15. The difference between a generic type and a parameterized type is that a generic type is a class or interface that introduces a family of parameterized types by declaring a formal type parameter list, and a parameterized type is an instance of a generic type.

16. Anonymous classes cannot be generic because they have no names.

17. The five kinds of actual type arguments are concrete types, concrete parameterized types, array types, type parameters, and wildcards.

18. The answer is true: you cannot specify a primitive-type name (e.g., `double` or `int`) as an actual type argument.

19. A raw type is a generic type without its type parameters.

20. The compiler reports an unchecked warning message when it detects an explicit cast that involves a type parameter. The compiler is concerned that downcasting to whatever type is passed to the type parameter might result in a violation of type safety.

21. You suppress an unchecked warning message by prefixing the constructor or method that contains the unchecked code with the `@SuppressWarnings("unchecked")` annotation.

22. The answer is true: `List<E>`'s `E` type parameter is unbounded.

23. You specify a single upper bound via reserved word `extends` followed by a type name.

24. A recursive type bound is a type parameter bound that includes the type parameter.

25. Wildcard type arguments are necessary because by accepting any actual type argument, they provide a typesafe workaround to the problem of polymorphic behavior not applying to multiple parameterized types that differ only in regard to one type parameter being a subtype of another type parameter. For example, because `List<String>` is not a kind of `List<Object>`, you cannot pass an object whose type is `List<String>` to a method parameter whose type is `List<Object>`. However, you can pass a `List<String>` object to `List<?>` provided that you are not going to add the `List<String>` object to the `List<?>`.

26. A generic method is a class or instance method with a type-generalized implementation.

27. Although you might think otherwise, Listing 6-36's `methodCaller()` generic method calls `someOverloadedMethod(Object o)`. This method, instead of `someOverloadedMethod(Date d)`, is called because overload resolution happens at compile time, when the generic method is translated to its unique bytecode representation, and erasure (which takes care of that mapping) causes type parameters to be replaced by their leftmost bound or `Object` (when there is no bound). After erasure, you are left with Listing A-24's nongeneric `methodCaller()` method.

Listing A-24. The Nongeneric `methodCaller()` Method That Results from Erasure

```
public static void methodCaller(Object t)
{
   someOverloadedMethod(t);
}
```

28. Reification is representing the abstract as if it was concrete.

29. The answer is false: type parameters are not reified.

30. Erasure is the throwing away of type parameters following compilation so that they are not available at runtime. Erasure also involves replacing uses of other type variables by the upper bound of the type variable (e.g., `Object`) and inserting casts to the appropriate type when the resulting code is not type correct.

31. An enumerated type is a type that specifies a named sequence of related constants as its legal values.

32. Three problems that can arise when you use enumerated types whose constants are `int`-based are lack of compile-time type safety, brittle applications, and the inability to translate `int` constants into meaningful string-based descriptions.

33. An enum is an enumerated type that is expressed via reserved word `enum`.

34. You use a switch statement with an enum by specifying an enum constant as the statement's selector expression and constant names as case values.

35. You can enhance an enum by adding fields, constructors, and methods—you can even have the enum implement interfaces. Also, you can override `toString()` to provide a more useful description of a constant's value and subclass constants to assign different behaviors.

36. The purpose of the abstract `Enum` class is to serve as the common base class of all Java language-based enumeration types.

37. The difference between `Enum`'s `name()` and `toString()` methods is that `name()` always returns a constant's name, but `toString()` can be overridden to return a more meaningful description instead of the constant's name.

38. The answer is true: `Enum`'s generic type is `Enum<E extends Enum<E>>`.

39. Listing A-25 presents a `ToDo` marker annotation type that annotates only type elements and that also uses the default retention policy.

Listing A-25. The `ToDo` Annotation Type for Marking Types That Need to Be Completed

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;
```

```
@Target(ElementType.TYPE)
public @interface ToDo
{
}
```

40. Listing A-26 presents a rewritten `StubFinder` application that works with Listing 6-13's `Stub` annotation type (with appropriate `@Target` and `@Retention` annotations) and Listing 6-14's `Deck` class.

Listing A-26. Reporting a Stub's ID, Due Date, and Developer via a New Version of `StubFinder`

```
import java.lang.reflect.Method;

public class StubFinder
{
   public static void main(String[] args) throws Exception
   {
      if (args.length != 1)
      {
         System.err.println("usage: java StubFinder classfile");
         return;
      }
      Method[] methods = Class.forName(args[0]).getMethods();
      for (int i = 0; i < methods.length; i++)
         if (methods[i].isAnnotationPresent(Stub.class))
         {
            Stub stub = methods[i].getAnnotation(Stub.class);
            System.out.println("Stub ID = " + stub.id());
            System.out.println("Stub Date = " + stub.dueDate());
            System.out.println("Stub Developer = " + stub.developer());
            System.out.println();
         }
   }
}
```

41. Listing A-27 presents the generic `Stack` class and the `StackEmptyException` and `StackFullException` helper classes that were called for in Chapter 6.

Listing A-27. `Stack` and Its `StackEmptyException` and `StackFullException` Helper Classes Proving That Not All Helper Classes Need to Be Nested

```
public class Stack<E>
{
   private E[] elements;
   private int top;

   @SuppressWarnings("unchecked")
   Stack(int size)
   {
      if (size < 2)
         throw new IllegalArgumentException("" + size);
```

```
        elements = (E[]) new Object[size];
        top = -1;
    }

    void push(E element) throws StackFullException
    {
        if (top == elements.length - 1)
            throw new StackFullException();
        elements[++top] = element;
    }

    E pop() throws StackEmptyException
    {
        if (isEmpty())
            throw new StackEmptyException();
        return elements[top--];
    }

    boolean isEmpty()
    {
        return top == -1;
    }

    public static void main(String[] args)
        throws StackFullException, StackEmptyException
    {
        Stack<String> stack = new Stack<String>(5);
        assert stack.isEmpty();
        stack.push("A");
        stack.push("B");
        stack.push("C");
        stack.push("D");
        stack.push("E");
        // Uncomment the following line to generate a StackFullException.
        //stack.push("F");
        while (!stack.isEmpty())
            System.out.println(stack.pop());
        // Uncomment the following line to generate a StackEmptyException.
        //stack.pop();
        assert stack.isEmpty();
    }
}

class StackEmptyException extends Exception
{
}

class StackFullException extends Exception
{
}
```

42. Listing A-28 presents the `Compass` enum that was called for in Chapter 6.

Listing A-28. A `Compass` Enum with Four Direction Constants

```
enum Compass
{
   NORTH, SOUTH, EAST, WEST
}
```

Listing A-29 presents the `UseCompass` class that was called for in Chapter 6.

Listing A-29. Using the `Compass` Enum to Keep from Getting Lost

```
public class UseCompass
{
   public static void main(String[] args)
   {
      int i = (int) (Math.random() * 4);
      Compass[] dir = { Compass.NORTH, Compass.EAST, Compass.SOUTH,
                        Compass.WEST };
      switch(dir[i])
      {
         case NORTH: System.out.println("heading north"); break;
         case EAST : System.out.println("heading east"); break;
         case SOUTH: System.out.println("heading south"); break;
         case WEST : System.out.println("heading west"); break;
         default   : assert false; // Should never be reached.
      }
   }
}
```

Chapter 7: Exploring the Basic APIs Part 1

1. `Math` declares `double` constants `E` and `PI` that represent, respectively, the natural logarithm base value (2.71828. . .) and the ratio of a circle's circumference to its diameter (3.14159. . .). `E` is initialized to `2.718281828459045` and `PI` is initialized to `3.141592653589793`.

2. `Math.abs(Integer.MIN_VALUE)` equals `Integer.MIN_VALUE` because there doesn't exist a positive 32-bit integer equivalent of `MIN_VALUE`. (`Integer.MIN_VALUE` equals -2147483648 and `Integer.MAX_VALUE` equals 2147483647.)

3. `Math`'s `random()` method returns a pseudorandom number between 0.0 (inclusive) and 1.0 (exclusive). Expression `(int) Math.random() * limit` is incorrect because this expression always returns 0. The `(int)` cast operator has higher precedence than *, which means that the cast is performed before multiplication. `random()` returns a fractional value and the cast converts this value to 0, which is then multiplied by `limit`'s value, resulting in an overall value of 0.

4. The five special values that can arise during floating-point calculations are +infinity, –infinity, NaN, +0.0, and –0.0.

5. `Math` and `StrictMath` differ in the following ways:

   - `StrictMath`'s methods return exactly the same results on all platforms. In contrast, some of `Math`'s methods might return values that vary ever so slightly from platform to platform.
   - Because `StrictMath` cannot utilize platform-specific features such as an extended-precision math coprocessor, an implementation of `StrictMath` might be less efficient than an implementation of `Math`.

6. The purpose of `strictfp` is to restrict floating-point calculations to ensure portability. This reserved word accomplishes portability in the context of intermediate floating-point representations and overflows/underflows (generating a value too large or small to fit a representation). Furthermore, it can be applied at the method level or at the class level.

7. `BigDecimal` is an immutable class that represents a signed decimal number (e.g., 23.653) of arbitrary precision (number of digits) with an associated scale (an integer that specifies the number of digits after the decimal point). You might use this class to accurately store floating-point values that represent monetary values and properly round the result of each monetary calculation.

8. The `RoundingMode` constant that describes the form of rounding commonly taught at school is `HALF_UP`.

9. `BigInteger` is an immutable class that represents a signed integer of arbitrary precision. It stores its value in two's complement format (all bits are flipped—1s to 0s and 0s to 1s—and 1 has been added to the result to be compatible with the two's complement format used by Java's byte integer, short integer, integer, and long integer types).

10. The answer is true: a string literal is a `String` object.

11. The purpose of `String`'s `intern()` method is to store a unique copy of a `String` object in an internal table of `String` objects. `intern()` makes it possible to compare strings via their references and `==` or `!=`. These operators are the fastest way to compare strings, which is especially valuable when sorting a huge number of strings.

12. `String` and `StringBuffer` differ in that `String` objects contain immutable sequences of characters, whereas `StringBuffer` objects contain mutable sequences of characters.

13. `StringBuffer` and `StringBuilder` differ in that `StringBuffer` methods are synchronized, whereas `StringBuilder`'s equivalent methods are not synchronized. As a result, you would use the thread-safe but slower `StringBuffer` class in multithreaded situations and the nonthread-safe but faster `StringBuilder` class in single-threaded situations.

14. The purpose of `Package`'s `isSealed()` method is to indicate whether or not a package is sealed (all classes that are part of the package are archived in the same JAR file). This method returns true when the package is sealed.

15. The answer is true: `getPackage()` requires at least one classfile to be loaded from the package before it returns a `Package` object describing that package.

16. Listing A-30 presents the `PrimeNumberTest` application that was called for in Chapter 7.

Listing A-30. Checking a Positive Integer Argument to Discover If It Is Prime

```
public class PrimeNumberTest
{
   public static void main(String[] args)
   {
      if (args.length != 1)
      {
         System.err.println("usage: java PrimeNumberTest integer");
         System.err.println("integer must be 2 or higher");
         return;
      }
      try
      {
         int n = Integer.parseInt(args[0]);
         if (n < 2)
         {
             System.err.println(n + " is invalid because it is less than 2");
             return;
         }
         for (int i = 2; i <= Math.sqrt(n); i++)
            if (n % i == 0)
            {
               System.out.println (n + " is not prime");
               return;
            }
         System.out.println(n + " is prime");
      }
      catch (NumberFormatException nfe)
      {
         System.err.println("unable to parse " + args[0] + " into an int");
      }
   }
}
```

17. The following loop uses `StringBuffer` to minimize object creation:

```
String[] imageNames = new String[NUM_IMAGES];
StringBuffer sb = new StringBuffer();
for (int i = 0; i < imageNames.length; i++)
```

```
{
   sb.append("image");
   sb.append(i);
   sb.append(".png");
   imageNames[i] = sb.toString();
   sb.setLength(0); // Erase previous StringBuffer contents.
}
```

18. Listing A-31 presents the `DigitsToWords` application that was called for in Chapter 7.

Listing A-31. Converting an Integer Value to Its Textual Representation

```
public class DigitsToWords
{
   public static void main(String[] args)
   {
      if (args.length != 1)
      {
         System.err.println("usage: java DigitsToWords integer");
         return;
      }
      System.out.println(convertDigitsToWords(Integer.parseInt(args[0])));
   }

   static String convertDigitsToWords(int integer)
   {
      if (integer < 0 || integer > 9999)
          throw new IllegalArgumentException("Out of range: " + integer);
      if (integer == 0)
          return "zero";
      String[] group1 =
      {
         "one",
         "two",
         "three",
         "four",
         "five",
         "six",
         "seven",
         "eight",
         "nine"
      };
      String[] group2 =
      {
         "ten",
         "eleven",
         "twelve",
         "thirteen",
         "fourteen",
         "fifteen",
         "sixteen",
```

```
            "seventeen",
            "eighteen",
            "nineteen"
        };
        String[] group3 =
        {
            "twenty",
            "thirty",
            "fourty",
            "fifty",
            "sixty",
            "seventy",
            "eighty",
            "ninety"
        };
        StringBuffer result = new StringBuffer();
        if (integer >= 1000)
        {
            int tmp = integer / 1000;
            result.append(group1[tmp - 1] + " thousand");
            integer -= tmp * 1000;
            if (integer == 0)
                return result.toString();
            result.append(" ");
        }
        if (integer >= 100)
        {
            int tmp = integer / 100;
            result.append(group1[tmp - 1] + " hundred");
            integer -= tmp * 100;
            if (integer == 0)
                return result.toString();
            result.append(" and ");
        }
        if (integer >= 10 && integer <= 19)
        {
            result.append(group2[integer - 10]);
            return result.toString();
        }
        if (integer >= 20)
        {
            int tmp = integer / 10;
            result.append(group3[tmp - 2]);
            integer -= tmp * 10;
            if (integer == 0)
                return result.toString();
            result.append("-");
        }
        result.append(group1[integer - 1]);
        return result.toString();
    }
}
```

Chapter 8: Exploring the Basic APIs Part 2

1. A primitive type wrapper class is a class whose instances wrap themselves around values of primitive types.
2. Java's primitive type wrapper classes include `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, and `Short`.
3. Java provides primitive type wrapper classes so that primitive-type values can be stored in collections and to provide a good place to associate useful constants and class methods with primitive types.
4. The answer is false: `Boolean` is the smallest of the primitive type wrapper classes.
5. You should use `Character` class methods instead of expressions such as `ch >= '0' && ch <= '9'` to determine whether or not a character is a digit, a letter, and so on because it's too easy to introduce a bug into the expression, expressions are not very descriptive of what they are testing, and the expressions are biased toward Latin digits (0–9) and letters (A–Z and a–z).
6. You determine whether or not `double` variable `d` contains +infinity or –infinity by passing this variable as an argument to `Double`'s `boolean isInfinite(double d)` class method, which returns true when this argument is +infinity or –infinity.
7. `Number` is the superclass of `Byte`, `Character`, and the other primitive type wrapper classes.
8. A thread is an independent path of execution through an application's code.
9. The purpose of the `Runnable` interface is to identify those objects that supply code for threads to execute via this interface's solitary `void run()` method.
10. The purpose of the `Thread` class is to provide a consistent interface to the underlying operating system's threading architecture. It provides methods that make it possible to associate code with threads as well as to start and manage those threads.
11. The answer is false: a `Thread` object associates with a single thread.
12. A race condition is a scenario in which multiple threads update the same object at the same time or nearly at the same time. Part of the object stores values written to it by one thread, and another part of the object stores values written to it by another thread.
13. Thread synchronization is the act of allowing only one thread at a time to execute code within a method or a block.
14. Synchronization is implemented in terms of monitors and locks.

15. Synchronization works by requiring that a thread that wants to enter a monitor-controlled critical section first acquire a lock. The lock is released automatically when the thread exits the critical section.

16. The answer is true: variables of type `long` or `double` are not atomic on 32-bit virtual machines.

17. The purpose of reserved word `volatile` is to let threads running on multiprocessor or multicore machines access a single copy of an instance field or class field. Without `volatile`, each thread might access its cached copy of the field and will not see modifications made by other threads to their copies.

18. The answer is false: `Object`'s `wait()` methods cannot be called from outside of a synchronized method or block.

19. Deadlock is a situation in which locks are acquired by multiple threads, neither thread holds its own lock but holds the lock needed by some other thread, and neither thread can enter and later exit its critical section to release its held lock because some other thread holds the lock to that critical section.

20. The purpose of the `ThreadLocal` class is to associate per-thread data (e.g., a user ID) with a thread.

21. `InheritableThreadLocal` differs from `ThreadLocal` in that the former class lets a child thread inherit a thread-local value from its parent thread.

22. The four `java.lang` package system classes discussed in Chapter 8 are `System`, `Runtime`, `Process`, and `ProcessBuilder`.

23. You invoke `System.arraycopy()` to copy an array to another array.

24. The `exec(String program)` method executes the program named `program` in a separate native process. The new process inherits the environment of the method's caller, and a `Process` object is returned to allow communication with the new process. `IOException` is thrown when an I/O error occurs.

25. `Process`'s `getInputStream()` method returns an `InputStream` reference for reading bytes that the new process writes to its output stream.

26. Listing A-32 presents the `MultiPrint` application that was called for in Chapter 8.

Listing A-32. Printing a Line of Text Multiple Times

```
public class MultiPrint
{
   public static void main(String[] args)
   {
      if (args.length != 2)
```

```
        {
           System.err.println("usage: java MultiPrint text count");
           return;
        }
        String text = args[0];
        int count = Integer.parseInt(args[1]);
        for (int i = 0; i < count; i++)
           System.out.println(text);
     }
}
```

27. Listing A-33 presents the revised `CountingThreads` application that was called for in Chapter 8.

Listing A-33. Counting via Daemon Threads

```
public class CountingThreads
{
   public static void main(String[] args)
   {
      Runnable r = new Runnable()
                   {
                      @Override
                      public void run()
                      {
                         String name = Thread.currentThread().getName();
                         int count = 0;
                         while (true)
                            System.out.println(name + ": " + count++);
                      }
                   };
      Thread thdA = new Thread(r);
      thdA.setDaemon(true);
      Thread thdB = new Thread(r);
      thdB.setDaemon(true);
      thdA.start();
      thdB.start();
   }
}
```

When you run this application, the two daemon threads start executing and you will probably see some output. However, the application will end as soon as the default main thread leaves the `main()` method and dies.

28. Listing A-34 presents the `StopCountingThreads` application that was called for in Chapter 8.

Listing A-34. Stopping the Counting Threads When Return/Enter Is Pressed

```
import java.io.IOException;

public class StopCountingThreads
{
   private static volatile boolean stopped = false;

   public static void main(String[] args)
   {
      Runnable r = new Runnable()
                   {
                      @Override
                      public void run()
                      {
                         String name = Thread.currentThread().getName();
                         int count = 0;
                         while (!stopped)
                            System.out.println(name + ": " + count++);
                      }
                   };
      Thread thdA = new Thread(r);
      Thread thdB = new Thread(r);
      thdA.start();
      thdB.start();
      try { System.in.read(); } catch (IOException ioe) {}
      stopped = true;
   }
}
```

29. Listing A-35 presents the `EVDump` application that was called for in Chapter 8.

Listing A-35. Dumping All Environment Variables to Standard Output

```
public class EVDump
{
   public static void main(String[] args)
   {
      System.out.println(System.getenv()); // System.out.println() calls toString()
                                           // on its object argument and outputs this
                                           // string
   }
}
```

Chapter 9: Exploring the Collections Framework

1. A collection is a group of objects that are stored in an instance of a class designed for this purpose.
2. The Collections Framework is a group of types that offers a standard architecture for representing and manipulating collections.

3. The Collections Framework largely consists of core interfaces, implementation classes, and utility classes.
4. A comparable is an object whose class implements the `Comparable` interface.
5. You would have a class implement the `Comparable` interface when you want objects to be compared according to their natural ordering.
6. A comparator is an object whose class implements the `Comparator` interface. Its purpose is to allow objects to be compared according to an order that is different from their natural ordering.
7. The answer is false: a collection uses a comparable (an object whose class implements the `Comparable` interface) to define the natural ordering of its elements.
8. The `Iterable` interface describes any object that can return its contained objects in some sequence.
9. The `Collection` interface represents a collection of objects that are known as elements.
10. A situation where `Collection`'s `add()` method would throw an instance of the `UnsupportedOperationException` class is an attempt to add an element to an unmodifiable collection.
11. `Iterable`'s `iterator()` method returns an instance of a class that implements the `Iterator` interface. This interface provides a `hasNext()` method to determine if the end of the iteration has been reached, a `next()` method to return a collection's next element, and a `remove()` method to remove the last element returned by `next()` from the collection.
12. The purpose of the enhanced for loop statement is to simplify collection or array iteration.
13. The enhanced for loop statement is expressed as `for (`*type id*`:` *collection*`)` or `for (`*type id*`:` *array*`)` and reads "for each *type* object in *collection*, assign this object to *id* at the start of the loop iteration"; or "for each *type* object in *array*, assign this object to *id* at the start of the loop iteration."
14. The answer is true: the enhanced for loop works with arrays. For example, `int[] x = { 1, 2, 3 }; for (int i: x) System.out.println(i);` declares array `x` and outputs all of its `int`-based elements.
15. Autoboxing is the act of wrapping a primitive-type value in an object of a primitive type wrapper class whenever a primitive type is specified but a reference is required. This feature saves the developer from having to explicitly instantiate a wrapper class when storing the primitive value in a collection.

16. Unboxing is the act of unwrapping a primitive-type value from its wrapper object whenever a reference is specified but a primitive type is required. This feature saves the developer from having to explicitly call a method on the object (e.g., `intValue()`) to retrieve the wrapped value.

17. A list is an ordered collection, which is also known as a sequence. Elements can be stored in and accessed from specific locations via integer indexes.

18. A `ListIterator` instance uses a cursor to navigate through a list.

19. A view is a list that is backed by another list. Changes that are made to the view are reflected in this backing list.

20. You would use the `subList()` method to perform range-view operations over a collection in a compact manner. For example, `list.subList(fromIndex, toIndex).clear();` removes a range of elements from `list`, where the first element is located at `fromIndex` and the last element is located at `toIndex - 1`.

21. The `ArrayList` class provides a list implementation that is based on an internal array.

22. The `LinkedList` class provides a list implementation that is based on linked nodes.

23. A node is a fixed sequence of value and link memory locations (i.e., an arrangement of a specific number of values and links, such as one value location followed by one link location). From an object-oriented perspective, it's an object whose fields store values and references to other node objects. These references are also known as links.

24. The answer is false: `ArrayList` provides slower element insertions and deletions than `LinkedList`.

25. A set is a collection that contains no duplicate elements.

26. The `TreeSet` class provides a set implementation that is based on a tree data structure. As a result, elements are stored in sorted order.

27. The `HashSet` class provides a set implementation that is backed by a hashtable data structure.

28. The answer is true: to avoid duplicate elements in a hashset, your own classes must correctly override `equals()` and `hashCode()`.

29. The difference between `HashSet` and `LinkedHashSet` is that `LinkedHashSet` uses a linked list to store its elements, resulting in its iterator returning elements in the order in which they were inserted.

30. The `EnumSet` class provides a `Set` implementation that is based on a bitset.

31. A sorted set is a set that maintains its elements in ascending order, sorted according to their natural ordering or according to a comparator that is supplied when the sorted set is created. Furthermore, the set's implementation class must implement the `SortedSet` interface.

32. A navigable set is a sorted set that can be iterated over in descending order as well as ascending order and which can report closest matches for given search targets.

33. The answer is false: `HashSet` is not an example of a sorted set. However, `TreeSet` is an example of a sorted set.

34. A sorted set's `add()` method would throw `ClassCastException` when you attempt to add an element to the sorted set because the element's class doesn't implement `Comparable`.

35. A queue is a collection in which elements are stored and retrieved in a specific order. Most queues are categorized as "first-in, first out," "last-in, first-out," or priority.

36. The answer is true: `Queue`'s `element()` method throws `NoSuchElementException` when it's called on an empty queue.

37. The `PriorityQueue` class provides an implementation of a priority queue, which is a queue that orders its elements according to their natural ordering or by a comparator provided when the queue is instantiated.

38. A map is a group of key/value pairs (also known as entries).

39. The `TreeMap` class provides a map implementation that is based on a red-black tree. As a result, entries are stored in sorted order of their keys.

40. The `HashMap` class provides a map implementation that is based on a hashtable data structure.

41. A hashtable uses a hash function to map keys to integer values.

42. Continuing from the previous exercise, the resulting integer values are known as hash codes; they identify hashtable array elements, which are known as buckets or slots.

43. A hashtable's capacity refers to the number of buckets.

44. A hashtable's load factor refers to the ratio of the number of stored entries divided by the number of buckets.

45. The difference between `HashMap` and `LinkedHashMap` is that `LinkedHashMap` uses a linked list to store its entries, resulting in its iterator returning entries in the order in which they were inserted.

46. The `IdentityHashMap` class provides a `Map` implementation that uses reference equality (`==`) instead of object equality (`equals()`) when comparing keys and values.

47. The `EnumMap` class provides a `Map` implementation whose keys are the members of the same enum.

48. A sorted map is a map that maintains its entries in ascending order, sorted according to the keys' natural ordering or according to a comparator that is supplied when the sorted map is created. Furthermore, the map's implementation class must implement the `SortedMap` interface.

49. A navigable map is a sorted map that can be iterated over in descending order as well as ascending order and which can report closest matches for given search targets.

50. The answer is true: `TreeMap` is an example of a sorted map.

51. The purpose of the `Arrays` class's `static <T> List<T> asList(T... array)` method is to return a fixed-size list backed by the specified `array`. (Changes to the returned list "write through" to the `array`.)

52. The answer is false: binary search is faster than linear search.

53. You would use `Collections`' `static <T> Set<T> synchronizedSet(Set<T> s)` method to return a synchronized variation of a hashset.

54. The seven legacy collections-oriented types are `Vector`, `Enumeration`, `Stack`, `Dictionary`, `Hashtable`, `Properties`, and `BitSet`.

55. Listing A-36 presents the `JavaQuiz` application that was called for in Chapter 9.

Listing A-36. How Much Do You Know About Java? Take the Quiz and Find Out!

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class JavaQuiz
{
   private static class QuizEntry
   {
      private String question;
      private String[] choices;
      private char answer;

      QuizEntry(String question, String[] choices, char answer)
      {
         this.question = question;
         this.choices = choices;
         this.answer = answer;
      }
```

```
   String[] getChoices()
   {
      // Demonstrate returning a copy of the choices array to prevent clients
      // from directly manipulating (and possibly screwing up) the internal
      // choices array.
      String[] temp = new String[choices.length];
      System.arraycopy(choices, 0, temp, 0, choices.length);
      return temp;
   }

   String getQuestion()
   {
      return question;
   }

   char getAnswer()
   {
      return answer;
   }
}

static QuizEntry[] quizEntries =
{
   new QuizEntry("What was Java's original name?",
                 new String[] { "Oak", "Duke", "J", "None of the above" },
                 'A'),
   new QuizEntry("Which of the following reserved words is also a literal?",
                 new String[] { "for", "long", "true", "enum" },
                 'C'),
   new QuizEntry("The conditional operator (?:) resembles which statement?",
                 new String[] { "switch", "if-else", "if", "while" },
                 'B')
};

public static void main(String[] args)
{
   // Populate the quiz list.
   List<QuizEntry> quiz = new ArrayList<QuizEntry>();
   for (QuizEntry entry: quizEntries)
      quiz.add(entry);
   // Perform the quiz.
   System.out.println("Java Quiz");
   System.out.println("---------\n");
   Iterator<QuizEntry> iter = quiz.iterator();
   while (iter.hasNext())
   {
      QuizEntry qe = iter.next();
      System.out.println(qe.getQuestion());
      String[] choices = qe.getChoices();
      for (int i = 0; i < choices.length; i++)
         System.out.println("  " + (char) ('A' + i) + ": " + choices[i]);
      int choice = -1;
```

```
            while (choice < 'A' || choice > 'A' + choices.length)
            {
               System.out.print("Enter choice letter: ");
               try
               {
                  choice = System.in.read();
                  // Remove trailing characters up to and including the newline
                  // to avoid having these characters automatically returned in
                  // subsequent System.in.read() method calls.
                  while (System.in.read() != '\n');
                  choice = Character.toUpperCase((char) choice);
               }
               catch (java.io.IOException ioe)
               {
               }
            }
            if (choice == qe.getAnswer())
               System.out.println("You are correct!\n");
            else
               System.out.println("You are not correct!\n");
         }
      }
   }
```

56. `(int) (f ^ (f >>> 32))` is used instead of `(int) (f ^ (f >> 32))` in the hash code generation algorithm because `>>>` always shifts a 0 to the right, which doesn't affect the hash code, whereas `>>` shifts a 0 or a 1 to the right (whatever value is in the sign bit), which affects the hash code when a 1 is shifted.

57. Listing A-37 presents the `FrequencyDemo` application that was called for in Chapter 9.

Listing A-37. Reporting the Frequency of Last Command-Line Argument Occurrences in the Previous Command-Line Arguments

```
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

public class FrequencyDemo
{
   public static void main(String[] args)
   {
      List<String> listOfArgs = new LinkedList<String>();
      String lastArg = (args.length == 0) ? null : args[args.length - 1];
      for (int i = 0; i < args.length - 1; i++)
         listOfArgs.add(args[i]);
      System.out.println("Number of occurrences of " + lastArg + " = " +
                         Collections.frequency(listOfArgs, lastArg));
   }
}
```

Chapter 10: Exploring Additional Utility APIs

1. A task is an object whose class implements the `Runnable` interface (a runnable task) or the `Callable` interface (a callable task).
2. An executor is an object whose class directly or indirectly implements the `Executor` interface, which decouples task submission from task-execution mechanics.
3. The `Executor` interface focuses exclusively on `Runnable`, which means that there is no convenient way for a runnable task to return a value to its caller (because `Runnable`'s `run()` method doesn't return a value); `Executor` doesn't provide a way to track the progress of executing runnable tasks, cancel an executing runnable task, or determine when the runnable task finishes execution; `Executor` cannot execute a collection of runnable tasks; and `Executor` doesn't provide a way for an application to shut down an executor (much less to properly shut down an executor).
4. `Executor`'s limitations are overcome by providing the `ExecutorService` interface.
5. The differences existing between `Runnable`'s `run()` method and `Callable`'s `call()` method are as follows: `run()` cannot return a value, whereas `call()` can return a value; and `run()` cannot throw checked exceptions, whereas `call()` can throw checked exceptions.
6. The answer is false: you can throw checked and unchecked exceptions from `Callable`'s `call()` method but can only throw unchecked exceptions from `Runnable`'s `run()` method.
7. A future is an object whose class implements the `Future` interface. It represents an asynchronous computation and provides methods for canceling a task, for returning a task's value, and for determining whether or not the task has finished.
8. The `Executors` class's `newFixedThreadPool()` method creates a thread pool that reuses a fixed number of threads operating off of a shared unbounded queue. At most, `nThreads` threads are actively processing tasks. If additional tasks are submitted when all threads are active, they wait in the queue for an available thread. If any thread terminates because of a failure during execution before the executor shuts down, a new thread will take its place when needed to execute subsequent tasks. The threads in the pool will exist until the executor is explicitly shut down.
9. A synchronizer is a class that facilitates a common form of synchronization.

10. Four commonly used synchronizers are countdown latches, cyclic barriers, exchangers, and semaphores. A countdown latch lets one or more threads wait at a "gate" until another thread opens this gate, at which point these other threads can continue. A cyclic barrier lets a group of threads wait for each other to reach a common barrier point. An exchanger lets a pair of threads exchange objects at a synchronization point. A semaphore maintains a set of permits for restricting the number of threads that can access a limited resource.

11. The concurrency-oriented extensions to the Collections Framework provided by the Concurrency Utilities are `ArrayBlockingQueue`, `BlockingDeque`, `BlockingQueue`, `ConcurrentHashMap`, `ConcurrentMap`, `ConcurrentNavigableMap`, `ConcurrentLinkedQueue`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet`, `CopyOnWriteArrayList`, `CopyOnWriteArraySet`, `DelayQueue`, `LinkedBlockingDeque`, `LinkedBlockingQueue`, `PriorityBlockingQueue`, and `SynchronousQueue`.

12. A lock is an instance of a class that implements the `Lock` interface, which provides more extensive locking operations than can be achieved via the `synchronized` reserved word. `Lock` also supports a wait/notification mechanism through associated `Condition` objects.

13. The biggest advantage that `Lock` objects hold over the implicit locks that are obtained when threads enter critical sections (controlled via the `synchronized` reserved word) is their ability to back out of an attempt to acquire a lock.

14. An atomic variable is an instance of a class that encapsulates a single variable and supports lock-free, thread-safe operations on that variable, for example, `AtomicInteger`.

15. The `Date` class describes a date in terms of a long integer that is relative to the Unix epoch.

16. The `Formatter` class is an interpreter for `printf()`-style format strings. This class provides support for layout justification and alignment; common formats for numeric, string, and date/time data; and more. Commonly used Java types (e.g., `byte` and `BigDecimal`) are supported.

17. Instances of the `Random` class generate sequences of random numbers by starting with a special 48-bit value that is known as a seed. This value is subsequently modified by a mathematical algorithm, which is known as a linear congruential generator.

18. The `Scanner` class parses an input stream of characters into primitive types, strings, and big integers/decimals under the control of regular expressions.

19. You call one of `Scanner`'s "`hasNext`" methods to determine if a character sequence represents an integer or some other kind of value before scanning that sequence.

20. Two differences between `ZipFile` and `ZipInputStream` are `ZipFile` allows random access to ZIP entries, whereas `ZipInputStream` allows sequential access; and `ZipFile` internally caches ZIP entries for improved performance, whereas `ZipInputStream` doesn't cache entries.

21. Listing A-38 presents the `ZipList` application that was called for in Chapter 10.

Listing A-38. Listing Archive Contents

```
import java.io.FileInputStream;
import java.io.IOException;

import java.util.Date;

import java.util.zip.ZipEntry;
import java.util.zip.ZipInputStream;

public class ZipList
{
   public static void main(String[] args) throws IOException
   {
      if (args.length != 1)
      {
         System.err.println("usage: java ZipList zipfile");
         return;
      }
      ZipInputStream zis = null;
      try
      {
         zis = new ZipInputStream(new FileInputStream(args[0]));
         ZipEntry ze;
         while ((ze = zis.getNextEntry()) != null)
         {
            System.out.println(ze.getName());
            System.out.println("   Compressed Size: " + ze.getCompressedSize());
            System.out.println("   Uncompressed Size: " + ze.getSize());
            if (ze.getTime() != -1)
               System.out.println("   Modification Time: " + new Date(ze.getTime()));
            System.out.println();
            zis.closeEntry();
         }
      }
      catch (IOException ioe)
      {
         System.err.println("I/O error: " + ioe.getMessage());
      }
      finally
      {
         if (zis != null)
            try
```

```
            {
               zis.close();
            }
            catch (IOException ioe)
            {
               assert false; // shouldn't happen in this context
            }
         }
      }
   }
```

Chapter 11: Performing Classic I/O

1. The purpose of the `File` class is to offer access to the underlying platform's available filesystem(s).
2. Instances of the `File` class contain the pathnames of files and directories that may or may not exist in their filesystems.
3. `File`'s `listRoots()` method returns an array of `File` objects denoting the root directories (roots) of available filesystems.
4. A path is a hierarchy of directories that must be traversed to locate a file or a directory. A pathname is a string representation of a path; a platform-dependent separator character (e.g., the Windows backslash [\] character) appears between consecutive names.
5. The difference between an absolute pathname and a relative pathname is as follows: an absolute pathname is a pathname that starts with the root directory symbol, whereas a relative pathname is a pathname that doesn't start with the root directory symbol; it's interpreted via information taken from some other pathname.
6. You obtain the current user (also known as working) directory by specifying `System.getProperty("user.dir")`.
7. A parent pathname is a string that consists of all pathname components except for the last name.
8. Normalize means to replace separator characters with the default name-separator character so that the pathname is compliant with the underlying filesystem.
9. You obtain the default name-separator character by accessing `File`'s `separator` and `separatorChar` class fields. The first field stores the character as a `char` and the second field stores it as a `String`.
10. A canonical pathname is a pathname that's absolute and unique and is formatted the same way every time.

11. The difference between `File`'s `getParent()` and `getName()` methods is that `getParent()` returns the parent pathname and `getName()` returns the last name in the pathname's name sequence.

12. The answer is false: `File`'s `exists()` method determines whether or not a file or directory exists.

13. A normal file is a file that's not a directory and satisfies other platform-dependent criteria: it's not a symbolic link or named pipe, for example. Any nondirectory file created by a Java application is guaranteed to be a normal file.

14. `File`'s `lastModified()` method returns the time that the file denoted by this `File` object's pathname was last modified or 0 when the file doesn't exist or an I/O error occurred during this method call. The returned value is measured in milliseconds since the Unix epoch (00:00:00 GMT, January 1, 1970).

15. The answer is true: `File`'s `list()` method returns an array of `String`s where each entry is a filename rather than a complete path.

16. The difference between the `FilenameFilter` and `FileFilter` interfaces is as follows: `FilenameFilter` declares a single `boolean accept(File dir, String name)` method, whereas `FileFilter` declares a single `boolean accept(String pathname)` method. Either method accomplishes the same task of accepting (by returning true) or rejecting (by returning false) the inclusion of the file or directory identified by the argument(s) in a directory listing.

17. The answer is false: `File`'s `createNewFile()` method checks for file existence and creates the file when it doesn't exist in a single operation that's atomic with respect to all other filesystem activities that might affect the file.

18. The default temporary directory where `File`'s `createTempFile(String, String)` method creates temporary files can be located by reading the `java.io.tmpdir` system property.

19. You ensure that a temporary file is removed when the virtual machine ends normally (it doesn't crash and the power isn't lost) by registering the temporary file for deletion through a call to `File`'s `deleteOnExit()` method.

20. You would accurately compare two `File` objects by first calling `File`'s `getCanonicalFile()` method on each `File` object and then comparing the returned `File` objects.

21. The purpose of the `RandomAccessFile` class is to create and/or open files for random access in which a mixture of write and read operations can occur until the file is closed.

22. The purpose of the `"rwd"` and `"rws"` mode arguments is to ensure than any writes to a file located on a local storage device are written to the device, which guarantees that critical data isn't lost when the system crashes. No guarantee is made when the file doesn't reside on a local device.

23. A file pointer is a cursor that identifies the location of the next byte to write or read. When an existing file is opened, the file pointer is set to its first byte at offset 0. The file pointer is also set to 0 when the file is created.

24. The answer is false: when you call `RandomAccessFile`'s `seek(long)` method to set the file pointer's value, and if this value is greater than the length of the file, the file's length doesn't change. The file length will only change by writing after the offset has been set beyond the end of the file.

25. A flat file database is a single file organized into records and fields. A record stores a single entry (e.g., a part in a parts database) and a field stores a single attribute of the entry (e.g., a part number).

26. A stream is an ordered sequence of bytes of arbitrary length. Bytes flow over an output stream from an application to a destination and flow over an input stream from a source to an application.

27. The purpose of `OutputStream`'s `flush()` method is to write any buffered output bytes to the destination. If the intended destination of this output stream is an abstraction provided by the underlying platform (e.g., a file), flushing the stream only guarantees that bytes previously written to the stream are passed to the underlying platform for writing; it doesn't guarantee that they're actually written to a physical device such as a disk drive.

28. The answer is true: `OutputStream`'s `close()` method automatically flushes the output stream. If an application ends before `close()` is called, the output stream is automatically closed and its data is flushed.

29. The purpose of `InputStream`'s `mark(int)` and `reset()` methods is to reread a portion of a stream. `mark(int)` marks the current position in this input stream. A subsequent call to `reset()` repositions this stream to the last marked position so that subsequent read operations reread the same bytes. Don't forget to call `markSupported()` to find out if the subclass supports `mark()` and `reset()`.

30. You would access a copy of a `ByteArrayOutputStream` instance's internal byte array by calling `ByteArrayOutputStream`'s `toByteArray()` method.

31. The answer is false: `FileOutputStream` and `FileInputStream` don't provide internal buffers to improve the performance of write and read operations.

32. You would use `PipedOutputStream` and `PipedInputStream` to communicate data between a pair of executing threads.

33. A filter stream is a stream that buffers, compresses/uncompresses, encrypts/decrypts, or otherwise manipulates an input stream's byte sequence before it reaches its destination.

34. Two streams are chained together when a stream instance is passed to another stream class's constructor.

35. You improve the performance of a file output stream by chaining a `BufferedOutputStream` instance to a `FileOutputStream` instance and calling the `BufferedOutputStream` instance's `write()` methods so that data is buffered before flowing to the file output stream. You improve the performance of a file input stream by chaining a `BufferedInputStream` instance to a `FileInputStream` instance so that data flowing from a file input stream is buffered before being returned from the `BufferedInputStream` instance by calling this instance's `read()` methods.

36. `DataOutputStream` and `DataInputStream` support `FileOutputStream` and `FileInputStream` by providing methods to write and read primitive-type values and strings in a platform-independent way. In contrast, `FileOutputStream` and `FileInputStream` provide methods for writing/reading bytes and arrays of bytes only.

37. Object serialization is a virtual machine mechanism for serializing object state into a stream of bytes. Its deserialization counterpart is a virtual machine mechanism for deserializing this state from a byte stream.

38. The three forms of serialization and deserialization that Java supports are default serialization and deserialization, custom serialization and deserialization, and externalization.

39. The purpose of the `Serializable` interface is to tell the virtual machine that it's okay to serialize objects of the implementing class.

40. When the serialization mechanism encounters an object whose class doesn't implement `Serializable`, it throws an instance of the `NotSerializableException` class.

41. The three stated reasons for Java not supporting unlimited serialization are as follows: security, performance, and objects not amenable to serialization.

42. You initiate serialization by creating an `ObjectOutputStream` instance and calling its `writeObject()` method. You initialize deserialization by creating an `ObjectInputStream` instance and calling its `readObject()` method.

43. The answer is false: class fields are not automatically serialized.

44. The purpose of the `transient` reserved word is to mark instance fields that don't participate in default serialization and default deserialization.

45. The deserialization mechanism causes `readObject()` to throw an instance of the `InvalidClassException` class when it attempts to deserialize an object whose class has changed.

46. The deserialization mechanism detects that a serialized object's class has changed as follows: Every serialized object has an identifier. The deserialization mechanism compares the identifier of the object being deserialized with the serialized identifier of its class (all serializable classes are automatically given unique identifiers unless they explicitly specify their own identifiers) and causes `InvalidClassException` to be thrown when it detects a mismatch.

47. You can add an instance field to a class and avoid trouble when deserializing an object that was serialized before the instance field was added by introducing a `long serialVersionUID =` *long integer value*`;` declaration into the class. The *long integer value* must be unique and is known as a stream unique identifier (SUID). You can use the JDK's `serialver` tool to help with this task.

48. You customize the default serialization and deserialization mechanisms without using externalization by declaring private `void writeObject(ObjectOutputStream)` and `void readObject(ObjectInputStream)` methods in the class.

49. You tell the serialization and deserialization mechanisms to serialize or deserialize the object's normal state before serializing or deserializing additional data items by first calling `ObjectOutputStream`'s `defaultWriteObject()` method in `writeObject(ObjectOutputStream)` and by first calling `ObjectInputStream`'s `defaultReadObject()` method in `readObject(ObjectInputStream)`.

50. Externalization differs from default and custom serialization and deserialization in that it offers complete control over the serialization and deserialization tasks.

51. A class indicates that it supports externalization by implementing the `Externalizable` interface instead of `Serializable` and by declaring `void writeExternal(ObjectOutput)` and `void readExternal(ObjectInput in)` methods instead of `void writeObject(ObjectOutputStream)` and `void readObject(ObjectInputStream)` methods.

52. The answer is true: during externalization, the deserialization mechanism throws `InvalidClassException` with a "no valid constructor" message when it doesn't detect a `public` noargument constructor.

53. The difference between `PrintStream`'s `print()` and `println()` methods is that the `print()` methods don't append a line terminator to their output, whereas the `println()` methods append a line terminator.

54. `PrintStream`'s noargument `void println()` method outputs the `line.separator` system property's value to ensure that lines are terminated in a portable manner (e.g., a carriage return followed by a newline/line feed on Windows or only a newline/line feed on Unix/Linux).

55. Java's stream classes are not good at streaming characters because bytes and characters are two different things: a byte represents an 8-bit data item and a character represents a 16-bit data item. Also, byte streams have no knowledge of character sets and their character encodings.

56. Java provides writer and reader classes as the preferred alternative to stream classes when it comes to character I/O.

57. The answer is false: `Reader` doesn't declare an `available()` method.

58. The purpose of the `OutputStreamWriter` class is to serve as a bridge between an incoming sequence of characters and an outgoing stream of bytes. Characters written to this writer are encoded into bytes according to the default or specified character encoding. The purpose of the `InputStreamReader` class is to serve as a bridge between an incoming stream of bytes and an outgoing sequence of characters. Characters read from this reader are decoded from bytes according to the default or specified character encoding.

59. You identify the default character encoding by reading the value of the `file.encoding` system property.

60. The purpose of the `FileWriter` class is to conveniently connect to the underlying file output stream using the default character encoding. The purpose of the `FileReader` class is to conveniently connect to the underlying file input stream using the default character encoding.

61. Listing A-39 presents the `Touch` application that was called for in Chapter 11.

Listing A-39. Setting a File or Directory's Timestamp to the Current Time

```
import java.io.File;

import java.util.Date;

public class Touch
{
   public static void main(String[] args)
   {
      if (args.length != 1)
      {
         System.err.println("usage: java Touch pathname");
         return;
      }
      new File(args[0]).setLastModified(new Date().getTime());
   }
}
```

62. Listing A-40 presents the Copy application that was called for in Chapter 11.

Listing A-40. Copying a Source File to a Destination File with Buffered I/O

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class Copy
{
   public static void main(String[] args)
   {
      if (args.length != 2)
      {
         System.err.println("usage: java Copy srcfile dstfile");
         return;
      }
      BufferedInputStream bis = null;
      BufferedOutputStream bos = null;
      try
      {
         FileInputStream fis = new FileInputStream(args[0]);
         bis = new BufferedInputStream(fis);
         FileOutputStream fos = new FileOutputStream(args[1]);
         bos = new BufferedOutputStream(fos);
         int b; // I chose b instead of byte because byte is a reserved word.
         while ((b = bis.read()) != -1)
            bos.write(b);
      }
      catch (FileNotFoundException fnfe)
      {
         System.err.println(args[0] + " could not be opened for input, or " +
                            args[1] + " could not be created for output");
      }
      catch (IOException ioe)
      {
         System.err.println("I/O error: " + ioe.getMessage());
      }
      finally
      {
         if (bis != null)
            try
            {
               bis.close();
            }
            catch (IOException ioe)
            {
               assert false; // shouldn't happen in this context
            }
```

```
            if (bos != null)
               try
               {
                  bos.close();
               }
               catch (IOException ioe)
               {
                  assert false; // shouldn't happen in this context
               }
         }
      }
   }
```

63. Listing A-41 presents the `Split` application that was called for in Chapter 11.

Listing A-41. Splitting a Large File into Numerous Smaller Part Files

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class Split
{
   static final int FILESIZE = 1400000;
   static byte[] buffer = new byte[FILESIZE];

   public static void main(String[] args)
   {
      if (args.length != 1)
      {
         System.err.println("usage: java Split pathname");
         return;
      }
      File file = new File(args[0]);
      long length = file.length();
      int nWholeParts = (int) (length / FILESIZE);
      int remainder = (int) (length % FILESIZE);
      System.out.printf("Splitting %s into %d parts%n", args[0],
                        (remainder == 0) ? nWholeParts : nWholeParts + 1);
      BufferedInputStream bis = null;
      BufferedOutputStream bos = null;
      try
      {
         FileInputStream fis = new FileInputStream(args[0]);
         bis = new BufferedInputStream(fis);
         for (int i = 0; i < nWholeParts; i++)
         {
            bis.read(buffer);
            System.out.println("Writing part " + i);
```

```
                FileOutputStream fos = new FileOutputStream("part" + i);
                bos = new BufferedOutputStream(fos);
                bos.write(buffer);
                bos.close();
                bos = null;
             }
             if (remainder != 0)
             {
                int br = bis.read(buffer);
                if (br != remainder)
                {
                   System.err.println("Last part mismatch: expected " + remainder
                                      + " bytes");
                   System.exit(0);
                }
                System.out.println("Writing part " + nWholeParts);
                FileOutputStream fos = new FileOutputStream("part" + nWholeParts);
                bos = new BufferedOutputStream(fos);
                bos.write(buffer, 0, remainder);
             }
          }
          catch (IOException ioe)
          {
             ioe.printStackTrace();
          }
          finally
          {
             if (bis != null)
                try
                {
                   bis.close();
                }
                catch (IOException ioe)
                {
                   assert false; // shouldn't happen in this context
                }
             if (bos != null)
                try
                {
                   bos.close();
                }
                catch (IOException ioe)
                {
                   assert false; // shouldn't happen in this context
                }
          }
       }
    }
```

64. Listing A-42 presents the `CircleInfo` application that was called for in Chapter 11.

Listing A-42. Reading Lines of Text from Standard Input That Represent Circle Radii and Outputting Circumference and Area Based on the Current Radius

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class CircleInfo
{
   public static void main(String[] args) throws IOException
   {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      while (true)
      {
         System.out.print("Enter circle's radius: ");
         String str = br.readLine();
         double radius;
         try
         {
            radius = Double.valueOf(str).doubleValue();
            if (radius <= 0)
               System.err.println("radius must not be 0 or negative");
            else
            {
               System.out.println("Circumference: " + Math.PI * 2.0 * radius);
               System.out.println("Area: " + Math.PI * radius * radius);
               System.out.println();
            }
         }
         catch (NumberFormatException nfe)
         {
            System.err.println("not a number: " + nfe.getMessage());
         }
      }
   }
}
```

Chapter 12: Accessing Networks

1. A network is a group of interconnected nodes that can be shared among the network's users.

2. An intranet is a network located within an organization, and an internet is a network connecting organizations to each other.

3. Intranets and internets often use TCP/IP to communicate between nodes. Transmission Control Protocol (TCP) is a connection-oriented protocol, User Datagram Protocol (UDP) is a connectionless protocol, and Internet Protocol (IP) is the basic protocol over which TCP and UDP perform their tasks.
4. A host is a computer-based TCP/IP node.
5. A socket is an endpoint in a communications link between two processes.
6. A socket is identified by an IP address that identifies the host and by a port number that identifies the process running on that host.
7. An IP address is a 32-bit or 128-bit unsigned integer that uniquely identifies a network host or some other network node.
8. A packet is an addressable message chunk. Packets are often referred to as IP datagrams.
9. A socket address is comprised of an IP address and a port number.
10. The `InetAddress` subclasses that are used to represent IPv4 and IPv6 addresses are `Inet4Address` and `Inet6Address`.
11. The loopback interface is a software-based network interface where outgoing data loops back as incoming data.
12. The answer is false: in network byte order, the most significant byte comes first.
13. The local host is represented by hostname localhost or by an IP address that's commonly expressed as 127.0.0.1 (IPv4) or ::1 (IPv6).
14. A socket option is a parameter for configuring socket behavior.
15. Socket options are described by constants that are declared in the `SocketOptions` interface.
16. The answer is false: you don't set a socket option by calling the `void setOption(int optID, Object value)` method. Instead, you call one of the type-safe socket option methods that are declared in a `Socket`-suffixed class.
17. Sockets based on the `Socket` class are commonly referred to as stream sockets because `Socket` is associated with the `InputStream` and `OutputStream` classes.
18. In the context of a `Socket` instance, binding makes a client socket address available to a server socket so that a server process can communicate with the client process via the server socket.
19. A proxy is a host that sits between an intranet and the Internet for security purposes. Java represents proxy settings via instances of the `java.net.Proxy` class.

20. The answer is false: the `ServerSocket()` constructor creates an unbound server socket.

21. The difference between the `DatagramSocket` and `MulticastSocket` classes is as follows: `DatagramSocket` lets you perform UDP-based communications between a pair of hosts, whereas `MulticastSocket` lets you perform UDP-based communications between many hosts.

22. A datagram packet is an array of bytes associated with an instance of the `DatagramPacket` class.

23. The difference between unicasting and multicasting is as follows: unicasting is the act of a server sending a message to a single client, whereas multicasting is the act of a server sending a message to multiple clients.

24. A URL is a character string that specifies where a resource (e.g., a web page) is located on a TCP/IP-based network (e.g., the Internet). Also, it provides the means to retrieve that resource.

25. A URN is a character string that names a resource and doesn't provide a way to access that resource (the resource might not be available).

26. The answer is true: URLs and URNs are also URIs.

27. The `URL(String s)` constructor throws `MalformedURLException` when you pass `null` to `s`.

28. The equivalent of `openStream()` is to execute `openConnection().getInputStream()`.

29. The answer is false: you don't need to invoke `URLConnection`'s `void setDoInput(boolean doInput)` method with `true` as the argument before you can input content from a web resource. The default setting is true.

30. When it encounters a space character, `URLEncoder` converts it to a plus sign.

31. The `NetworkInterface` class represents a network interface as a name and a list of IP addresses assigned to this interface. Furthermore, it's used to identify the local interface on which a multicast group is joined.

32. A MAC address is an array of bytes containing a network interface's hardware address.

33. MTU stands for Maximum Transmission Unit. This size represents the maximum length of a message that can fit into an IP datagram without needing to fragment the message into multiple IP datagrams.

34. The answer is false: `NetworkInterface`'s `getName()` method returns a network interface's name (e.g., `eth0` or `lo`), not a human-readable display name.

35. `InterfaceAddress`'s `getNetworkPrefixLength()` method returns the subnet mask under IPv4.

36. HTTP cookie (cookie for short) is a state object.

37. It's preferable to store cookies on the client rather than on the server because of the potential for millions of cookies (depending on a website's popularity).

38. The four `java.net` types that are used to work with cookies are `CookieHandler`, `CookieManager`, `CookiePolicy`, and `CookieStore`.

39. Listing A-43 presents the enhanced `EchoClient` application that was called for in Chapter 12.

Listing A-43. Echoing Data to and Receiving It Back from a Server and Explicitly Closing the Socket

```
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;

import java.net.Socket;
import java.net.UnknownHostException;

public class EchoClient
{
   public static void main(String[] args)
   {
      if (args.length != 1)
      {
         System.err.println("usage  : java EchoClient message");
         System.err.println("example: java EchoClient \"This is a test.\"");
         return;
      }
      Socket socket = null;
      try
      {
         socket = new Socket("localhost", 9999);
         OutputStream os = socket.getOutputStream();
         OutputStreamWriter osw = new OutputStreamWriter(os);
         PrintWriter pw = new PrintWriter(osw);
         pw.println(args[0]);
         pw.flush();
         InputStream is = socket.getInputStream();
         InputStreamReader isr = new InputStreamReader(is);
         BufferedReader br = new BufferedReader(isr);
         System.out.println(br.readLine());
      }
      catch (UnknownHostException uhe)
      {
         System.err.println("unknown host: " + uhe.getMessage());
      }
```

```
      catch (IOException ioe)
      {
         System.err.println("I/O error: " + ioe.getMessage());
      }
      finally
      {
         if (socket != null)
            try
            {
               socket.close();
            }
            catch (IOException ioe)
            {
               assert false; // shouldn't happen in this context
            }
      }
   }
}
```

40. Listing A-44 presents the enhanced `EchoServer` application that was called for in Chapter 12.

Listing A-44. Receiving Data from and Echoing It Back to a Client and Explicitly Closing the Socket After a `Kill` File Appears

```
import java.io.BufferedReader;
import java.io.File;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;

import java.net.ServerSocket;
import java.net.Socket;

public class EchoServer
{
   public static void main(String[] args)
   {
      System.out.println("Starting echo server...");
      ServerSocket ss = null;
      try
      {
         ss = new ServerSocket(9999);
         File file = new File("kill");
         while (!file.exists())
         {
            Socket s = ss.accept(); // waiting for client request
            try
```

```
            {
               InputStream is = s.getInputStream();
               InputStreamReader isr = new InputStreamReader(is);
               BufferedReader br = new BufferedReader(isr);
               String msg = br.readLine();
               System.out.println(msg);
               OutputStream os = s.getOutputStream();
               OutputStreamWriter osw = new OutputStreamWriter(os);
               PrintWriter pw = new PrintWriter(osw);
               pw.println(msg);
               pw.flush();
            }
            catch (IOException ioe)
            {
               System.err.println("I/O error: " + ioe.getMessage());
            }
            finally
            {
               try
               {
                  s.close();
               }
               catch (IOException ioe)
               {
                  assert false; // shouldn't happen in this context
               }
            }
         }
      }
      catch (IOException ioe)
      {
         System.err.println("I/O error: " + ioe.getMessage());
      }
      finally
      {
         if (ss != null)
            try
            {
               ss.close();
            }
            catch (IOException ioe)
            {
               assert false; // shouldn't happen in this context
            }
      }
   }
}
```

Chapter 13: Migrating to New I/O

1. New I/O is an architecture that supports memory-mapped file I/O, readiness selection, file locking, and more. This architecture consists of buffers, channels, selectors, regular expressions, and charsets.
2. A buffer is an object that stores a fixed amount of data to be sent to or received from an I/O service (a means for performing input/output).
3. A buffer's four properties are capacity, limit, position, and mark.
4. When you invoke `Buffer`'s `array()` method on a buffer backed by a read-only array, this method throws `ReadOnlyBufferException`.
5. When you invoke `Buffer`'s `flip()` method on a buffer, the limit is set to the current position and then the position is set to zero. When the mark is defined, it's discarded. The buffer is now ready to be drained.
6. When you invoke `Buffer`'s `reset()` method on a buffer where a mark has not been set, this method throws `InvalidMarkException`.
7. The answer is false: buffers are not thread-safe.
8. The classes that extend the abstract `Buffer` class are `ByteBuffer`, `CharBuffer`, `DoubleBuffer`, `FloatBuffer`, `IntBuffer`, `LongBuffer`, and `ShortBuffer`. Furthermore, this package includes `MappedByteBuffer` as an abstract `ByteBuffer` subclass.
9. You create a byte buffer by invoking one of its `allocate()`, `allocateDirect()`, or `wrap()` class methods.
10. A view buffer is a buffer that manages another buffer's data.
11. A view buffer is created by calling a `Buffer` subclass's `duplicate()` method.
12. You create a read-only view buffer by calling a `Buffer` subclass method such as `ByteBuffer asReadOnlyBuffer()` or `CharBuffer asReadOnlyBuffer()`.
13. `ByteBuffer`'s methods for storing a single byte in a byte buffer are `ByteBuffer put(int index, byte b)` and `ByteBuffer put(byte b)`. `ByteBuffer`'s methods for fetching a single byte from a byte buffer are `byte get(int index)` method and `byte get()`.
14. Attempting to use the relative `put()` method or the relative `get()` method when the current position is greater than or equal to the limit causes `BufferOverflowException` or `BufferUnderflowException` to occur.
15. The equivalent of executing `buffer.flip();` is to execute `buffer.limit(buffer.position()).position(0);`.
16. The answer is false: calling `flip()` twice doesn't return you to the original state. Instead, the buffer has a zero size.

17. The difference between `Buffer`'s `clear()` and `reset()` methods is as follows: the `clear()` method marks a buffer as empty, whereas `reset()` changes the buffer's current position to the previously set mark or throws `InvalidMarkException` when there's no previously set mark.

18. `ByteBuffer`'s `compact()` method copies all bytes between the current position and the limit to the beginning of the buffer. The byte at index `p = position()` is copied to index 0, the byte at index `p + 1` is copied to index 1, and so on until the byte at index `limit() - 1` is copied to index `n = limit() - 1 - p`. The buffer's current position is then set to `n + 1` and its limit is set to its capacity. The mark, when defined, is discarded.

19. The purpose of the `ByteOrder` class is to help you deal with byte-order issues when writing/reading multibyte values to/from a multibyte buffer.

20. A direct byte buffer is a byte buffer that interacts with channels and native code to perform I/O. The direct byte buffer attempts to store byte elements in a memory area that a channel uses to perform direct (raw) access via native code that tells the operating system to drain or fill the memory area directly.

21. You obtain a direct byte buffer by invoking `ByteBuffer`'s `allocateDirect()` method.

22. A channel is an object that represents an open connection to a hardware device, a file, a network socket, an application component, or another entity that's capable of performing write, read, and other I/O operations. Channels efficiently transfer data between byte buffers and I/O service sources or destinations.

23. The capabilities that the `Channel` interface provides are closing a channel (via the `close()` method) and determining whether or not a channel is open (via the `isOpen()`) method.

24. The three interfaces that directly extend `Channel` are `WritableByteChannel`, `ReadableByteChannel`, and `InterruptibleChannel`.

25. The answer is true: a channel that implements `InterruptibleChannel` is asynchronously closeable.

26. The two ways to obtain a channel are to invoke a `Channels` class method, such as `WritableByteChannel newChannel(OutputStream outputStream)`, and to invoke a channel method on a classic I/O class, such as `RandomAccessFile` 's `FileChannel getChannel()` method.

27. Scatter/gather I/O is the ability to perform a single I/O operation across multiple buffers.

28. The `ScatteringByteChannel` and `GatheringByteChannel` interfaces are provided for achieving scatter/gather I/O.

29. A file channel is a channel to an underlying file.

30. The answer is false: file channels support scatter/gather I/O.

31. `FileChannel` provides the `MappedByteBuffer map(FileChannel.MapMode mode, long position, long size)` method for mapping a region of a file into memory.

32. The fundamental difference between `FileChannel`'s `lock()` and `tryLock()` methods is that the `lock()` methods can block and the `tryLock()` methods never block.

33. A regular expression (also known as a regex or regexp) is a string-based pattern that represents the set of strings that match this pattern.

34. Instances of the `Pattern` class represent patterns via compiled regexes. Regexes are compiled for performance reasons; pattern matching via compiled regexes is much faster than if the regexes were not compiled.

35. `Pattern`'s `compile()` methods throw instances of the `PatternSyntaxException` class when they discover illegal syntax in their regular expression arguments.

36. Instances of the `Matcher` class attempt to match compiled regexes against input text.

37. The difference between `Matcher`'s `matches()` and `lookingAt()` methods is that unlike `matches()`, `lookingAt()` doesn't require the entire region to be matched.

38. A character class is a set of characters appearing between `[` and `]`.

39. There are six kinds of character classes: simple, negation, range, union, intersection, and subtraction.

40. A capturing group saves a match's characters for later recall during pattern matching.

41. A zero-length match is a match of zero length in which the start and end indexes are equal.

42. A quantifier is a numeric value implicitly or explicitly bound to a pattern. Quantifiers are categorized as greedy, reluctant, or possessive.

43. The difference between a greedy quantifier and a reluctant quantifier is that a greedy quantifier attempts to find the longest match, whereas a reluctant quantifier attempts to find the shortest match.

44. Possessive and greedy quantifiers differ in that a possessive quantifier only makes one attempt to find the longest match, whereas a greedy quantifier can make multiple attempts.

45. Listing A-45 presents the enhanced `Copy` application that was called for in Chapter 13.

Listing A-45. Copying a File via a Byte Buffer and a File Channel

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import java.nio.ByteBuffer;

import java.nio.channels.FileChannel;

public class Copy
{
   public static void main(String[] args)
   {
      if (args.length != 2)
      {
         System.err.println("usage: java Copy srcfile dstfile");
         return;
      }
      FileChannel fcSrc = null;
      FileChannel fcDest = null;
      try
      {
         FileInputStream fis = new FileInputStream(args[0]);
         fcSrc = fis.getChannel();
         FileOutputStream fos = new FileOutputStream(args[1]);
         fcDest = fos.getChannel();
         ByteBuffer buffer = ByteBuffer.allocateDirect(2048);
         while ((fcSrc.read(buffer)) != -1)
         {
            buffer.flip();
            while (buffer.hasRemaining())
               fcDest.write(buffer);
            buffer.clear();
         }
      }
      catch (FileNotFoundException fnfe)
      {
         System.err.println(args[0] + " could not be opened for input, or " +
                            args[1] + " could not be created for output");
      }
      catch (IOException ioe)
      {
         System.err.println("I/O error: " + ioe.getMessage());
      }
      finally
      {
         if (fcSrc != null)
            try
```

```
            {
               fcSrc.close();
            }
            catch (IOException ioe)
            {
               assert false; // shouldn't happen in this context
            }

         if (fcDest != null)
            try
            {
               fcDest.close();
            }
            catch (IOException ioe)
            {
               assert false; // shouldn't happen in this context
            }
      }
   }
}
```

46. Listing A-46 presents the `ReplaceText` application that was called for in Chapter 13.

Listing A-46. Replacing All Matches of the Pattern with Replacement Text

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.regex.PatternSyntaxException;

public class ReplaceText
{
   public static void main(String[] args)
   {
      if (args.length != 3)
      {
         System.err.println("usage: java ReplaceText text oldText newText");
         return;
      }
      try
      {
         Pattern p = Pattern.compile(args[1]);
         Matcher m = p.matcher(args[0]);
         String result = m.replaceAll(args[2]);
         System.out.println(result);
      }
      catch (PatternSyntaxException pse)
      {
         System.err.println(pse);
      }
   }
}
```

Chapter 14: Accessing Databases

1. A database is an organized collection of data.
2. A relational database is a database that organizes data into tables that can be related to each other.
3. Two other database categories are hierarchical databases and object-oriented databases.
4. A database management system is a set of programs that enables you to store, modify, and extract information from a database. It also provides users with tools to add, delete, access, modify, and analyze data stored in one location.
5. Java DB is a distribution of Apache's open-source Derby product, which is based on IBM's Cloudscape RDBMS code base.
6. The answer is false: Java DB's embedded driver causes the database engine to run in the same virtual machine as the application.
7. `setEmbeddedCP` adds `derby.jar` and `derbytools.jar` to the classpath so that you can access Java DB's embedded driver from your application.
8. The answer is false: you run Java DB's `sysinfo` command-line tool to view the Java environment/Java DB configuration.
9. SQLite is a very simple and popular RDBMS that implements a self-contained, serverless, zero-configuration, transactional SQL database engine and is considered to be the most widely deployed database engine in the world.
10. Manifest typing is the ability to store any value of any data type into any column regardless of the declared type of that column.
11. SQLite provides the `sqlite3` tool for accessing and modifying SQLite databases.
12. JDBC is an API for communicating with RDBMSs in an RDBMS-independent manner.
13. A data source is a data-storage facility ranging from a simple file to a complex relational database managed by an RDBMS.
14. A JDBC driver implements the `java.sql.Driver` interface.
15. The answer is false: there are four kinds of JDBC drivers.
16. A type three JDBC driver doesn't depend on native code and communicates with a middleware server via an RDBMS-independent protocol. The middleware server then communicates the client's requests to the data source.
17. JDBC provides the `java.sql.DriverManager` class and the `javax.sql.DataSource` interface for communicating with a data source.
18. You obtain a connection to a Java DB data source via the embedded driver by passing a URL of the form `jdbc:derby:`*databaseName*`;`*URLAttributes* to one of `DriverManager`'s `getConnection()` methods.

19. The answer is false: `int getErrorCode()` returns a vendor-specific error code.

20. A SQL state error code is a five-character string consisting of a two-character class value followed by a three-character subclass value.

21. The difference between `SQLNonTransientException` and `SQLTransientException` is as follows: `SQLNonTransientException` describes failed operations that cannot be retried without changing application source code or some aspect of the data source, and `SQLTransientException` describes failed operations that can be retried immediately.

22. JDBC's three statement types are `Statement`, `PreparedStatement`, and `CallableStatement`.

23. The `Statement` method that you call to execute an SQL `SELECT` statement is `ResultSet executeQuery(String sql)`.

24. A result set's cursor provides access to a specific row of data.

25. The SQL `FLOAT` type maps to Java's `double` type.

26. A prepared statement represents a precompiled SQL statement.

27. The answer is true: `CallableStatement` extends `PreparedStatement`.

28. A stored procedure is a list of SQL statements that perform a specific task.

29. You call a stored procedure by first obtaining a `CallableStatement` implementation instance (via one of `Connection`'s `prepareCall()` methods) that's associated with an escape clause, by next executing `CallableStatement` methods such as `void setInt(String parameterName, int x)` to pass arguments to escape clause parameters, and by finally invoking the `boolean execute()` method that `CallableStatement` inherits from its `PreparedStatement` superinterface.

30. An escape clause is RDBMS-independent syntax.

31. Metadata is data about data.

32. Metadata includes a list of catalogs, base tables, views, indexes, schemas, and additional information.

33. Listing A-47 presents the enhanced `JDBCDemo` application that was called for in Chapter 14.

Listing A-47. Outputting Database Metadata for the SQLite or Java DB Embedded Driver

```
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
```

```
public class JDBCDemo
{
   final static String URL1 = "jdbc:derby:employee;create=true";
   final static String URL2 = "jdbc:sqlite:employee";

   public static void main(String[] args)
   {
      String url = null;
      if (args.length != 1)
      {
         System.err.println("usage 1: java JDBCDemo javadb");
         System.err.println("usage 2: java JDBCDemo sqlite");
         return;
      }
      if (args[0].equals("javadb"))
         url = URL1;
      else
      if (args[0].equals("sqlite"))
         url = URL2;
      else
      {
         System.err.println("invalid command-line argument");
         return;
      }
      Connection con = null;
      try
      {
         if (args[0].equals("sqlite"))
            Class.forName("org.sqlite.JDBC");
         con = DriverManager.getConnection(url);
         dump(con.getMetaData());
      }
      catch (ClassNotFoundException cnfe)
      {
         System.err.println("unable to load sqlite driver");
      }
      catch (SQLException sqlex)
      {
         while (sqlex != null)
         {
            System.err.println("SQL error : " + sqlex.getMessage());
            System.err.println("SQL state : " + sqlex.getSQLState());
            System.err.println("Error code: " + sqlex.getErrorCode());
            System.err.println("Cause: " + sqlex.getCause());
            sqlex = sqlex.getNextException();
         }
      }
      finally
      {
         if (con != null)
            try
```

```
            {
               con.close();
            }
            catch (SQLException sqle)
            {
               sqle.printStackTrace();
            }
         }
      }

      static void dump(DatabaseMetaData dbmd) throws SQLException
      {
         System.out.println("DB Major Version = " + dbmd.getDatabaseMajorVersion());
         System.out.println("DB Minor Version = " + dbmd.getDatabaseMinorVersion());
         System.out.println("DB Product = " + dbmd.getDatabaseProductName());
         System.out.println("Driver Name = " + dbmd.getDriverName());
         System.out.println("Numeric function names for escape clause = " +
                            dbmd.getNumericFunctions());
         System.out.println("String function names for escape clause = " +
                            dbmd.getStringFunctions());
         System.out.println("System function names for escape clause = " +
                            dbmd.getSystemFunctions());
         System.out.println("Time/date function names for escape clause = " +
                            dbmd.getTimeDateFunctions());
         System.out.println("Catalog term: " + dbmd.getCatalogTerm());
         System.out.println("Schema term: " + dbmd.getSchemaTerm());
         System.out.println();
         System.out.println("Catalogs");
         System.out.println("--------");
         ResultSet rsCat = dbmd.getCatalogs();
         while (rsCat.next())
            System.out.println(rsCat.getString("TABLE_CAT"));
         System.out.println();
         System.out.println("Schemas");
         System.out.println("-------");
         ResultSet rsSchem = dbmd.getSchemas();
         while (rsSchem.next())
            System.out.println(rsSchem.getString("TABLE_SCHEM"));
         System.out.println();
         System.out.println("Schema/Table");
         System.out.println("------------");
         rsSchem = dbmd.getSchemas();
         while (rsSchem.next())
         {
            String schem = rsSchem.getString("TABLE_SCHEM");
            ResultSet rsTab = dbmd.getTables(null, schem, "%", null);
            while (rsTab.next())
               System.out.println(schem + " " + rsTab.getString("TABLE_NAME"));
         }
      }
   }
```

Appendix B

Four of a Kind

Application development isn't an easy task. If you don't plan carefully before you develop an application, you'll probably waste your time and money as you endeavor to create it, and waste your users' time and money when it doesn't meet their needs.

> **Caution** It's extremely important to test your software carefully. You could face a lawsuit if malfunctioning software causes financial harm to its users.

In this appendix, I present one technique for developing applications efficiently. I present this technique in the context of a Java application that lets you play a simple card game called *Four of a Kind* against the computer.

Understanding Four of a Kind

Before sitting down at the computer and writing code, you need to fully understand the problem domain that you are trying to model via that code. In this case, the problem domain is *Four of a Kind*, and you want to understand how this card game works.

Two to four players play *Four of a Kind* with a standard 52-card deck. The object of the game is to be the first player to put down four cards that have the same rank (four aces, for example), which wins the game.

The game begins by shuffling the deck and placing it face down. Each player takes a card from the top of the deck. The player with the highest ranked card (king is highest) deals four cards to each player, starting with the player to the dealer's left. The dealer then starts its turn.

The player examines its cards to determine which cards are optimal for achieving four of a kind. The player then throws away the least helpful card on a discard pile and picks up another card from the top of the deck. (If each card has a different rank, the player randomly selects a card to throw away.) If the player has four of a kind, the player puts down these cards (face up) and wins the game.

Modeling Four of a Kind in Pseudocode

Now that you understand how *Four of a Kind* works, you can begin to model this game. You will not model the game in Java source code because you would get bogged down in too many details. Instead, you will use pseudocode for this task.

Pseudocode is a compact and informal high-level description of the problem domain. Unlike the previous description of *Four of a Kind*, the pseudocode equivalent is a step-by-step recipe for solving the problem. Check out Listing B-1.

Listing B-1. Four of a Kind Pseudocode for Two Players (Human and Computer)

```
 1. Create a deck of cards and shuffle the deck.
 2. Create empty discard pile.
 3. Have each of the human and computer players take a card from the top of the deck.
 4. Designate the player with the highest ranked card as the current player.
 5. Return both cards to the bottom of the deck.
 6. The current player deals four cards to each of the two players in alternating fashion, with the
    first card being dealt to the other player.
 7. The current player examines its current cards to see which cards are optimal for achieving four
    of a kind. The current player throws the least helpful card onto the top of the discard pile.
 8. The current player picks up the deck's top card. If the current player has four of a kind, it
    puts down its cards and wins the game.
 9. Designate the other player as the current player.
10. If the deck has no more cards, empty the discard pile to the deck and shuffle the deck.
11. Repeat at step 7.
```

Deriving Listing B-1's pseudocode from the previous description is the first step in achieving an application that implements *Four of a Kind*. This pseudocode performs various tasks including decision making and repetition.

Despite being a more useful guide to understanding how *Four of a Kind* works, Listing B-1 is too high level for translation to Java. Therefore, you must refine this pseudocode to facilitate the translation process. Listing B-2 presents this refinement.

Listing B-2. Refined Four of a Kind Pseudocode for Two Players (Human and Computer)

```
 1. deck = new Deck()
 2. deck.shuffle()
 3. discardPile = new DiscardPile()
 4. hCard = deck.deal()
 5. cCard = deck.deal()
 6. if hCard.rank() == cCard.rank()
    6.1. deck.putBack(hCard)
    6.2. deck.putBack(cCard)
    6.3. deck.shuffle()
    6.4. Repeat at step 4
 7. curPlayer = HUMAN
    7.1. if cCard.rank() > hCard.rank()
         7.1.1. curPlayer = COMPUTER
 8. deck.putBack(hCard)
```

```
 9. deck.putBack(cCard)
10. if curPlayer == HUMAN
    10.1. for i = 0 to 3
          10.1.1. cCards[i] = deck.deal()
          10.1.2. hCards[i] = deck.deal()
    else
    10.2. for i = 0 to 3
          10.2.1. hCards[i] = deck.deal()
          10.2.2. cCards[i] = deck.deal()
11. if curPlayer == HUMAN
    11.01. output(hCards)
    11.02. choice = prompt("Identify card to throw away")
    11.03. discardPile.setTopCard(hCards[choice])
    11.04. hCards[choice] = deck.deal()
    11.05. if isFourOfAKind(hCards)
           11.05.1. output("Human wins!")
           11.05.2. putDown(hCards)
           11.05.3. output("Computer's cards:")
           11.05.4. putDown(cCards)
           11.05.5. End game
    11.06. curPlayer = COMPUTER
    else
    11.07. choice = leastDesirableCard(cCards)
    11.08. discardPile.setTopCard(cCards[choice])
    11.09. cCards[choice] = deck.deal()
    11.10. if isFourOfAKind(cCards)
           11.10.1. output("Computer wins!")
           11.10.2. putDown(cCards)
           11.10.3. End game
    11.11. curPlayer = HUMAN
12. if deck.isEmpty()
    12.1. if discardPile.topCard() != null
          12.1.1. deck.putBack(discardPile.getTopCard())
          12.1.2. Repeat at step 12.1.
    12.2. deck.shuffle()
13. Repeat at step 11.
```

In addition to being longer than Listing B-1, Listing B-2 shows the refined pseudocode becoming more like Java. For example, Listing B-2 reveals Java expressions (such as `new Deck()`, to create a `Deck` object), operators (such as `==`, to compare two values for equality), and method calls (such as `deck.isEmpty()`, to call `deck`'s `isEmpty()` method to return a Boolean value indicating whether [true] or not [false] the deck identified by `deck` is empty of cards).

Converting Pseudocode to Java Code

Now that you've had a chance to absorb Listing B-2's Java-like pseudocode, you're ready to examine the process of converting that pseudocode to Java source code. This process consists of a couple of steps.

The first step in converting Listing B-2's pseudocode to Java involves identifying important components of the game's structure and implementing these components as classes, which I formally introduced in Chapter 3.

Apart from the computer player (which is implemented via game logic), the important components are card, deck, and discard pile. I represent these components via `Card`, `Deck`, and `DiscardPile` classes. Listing B-3 presents `Card`.

Listing B-3. Merging Suits and Ranks into Cards

```
/**
 *  Simulating a playing card.
 *
 *  @author Jeff Friesen
 */

public enum Card
{
   ACE_OF_CLUBS(Suit.CLUBS, Rank.ACE),
   TWO_OF_CLUBS(Suit.CLUBS, Rank.TWO),
   THREE_OF_CLUBS(Suit.CLUBS, Rank.THREE),
   FOUR_OF_CLUBS(Suit.CLUBS, Rank.FOUR),
   FIVE_OF_CLUBS(Suit.CLUBS, Rank.FIVE),
   SIX_OF_CLUBS(Suit.CLUBS, Rank.SIX),
   SEVEN_OF_CLUBS(Suit.CLUBS, Rank.SEVEN),
   EIGHT_OF_CLUBS(Suit.CLUBS, Rank.EIGHT),
   NINE_OF_CLUBS(Suit.CLUBS, Rank.NINE),
   TEN_OF_CLUBS(Suit.CLUBS, Rank.TEN),
   JACK_OF_CLUBS(Suit.CLUBS, Rank.JACK),
   QUEEN_OF_CLUBS(Suit.CLUBS, Rank.QUEEN),
   KING_OF_CLUBS(Suit.CLUBS, Rank.KING),
   ACE_OF_DIAMONDS(Suit.DIAMONDS, Rank.ACE),
   TWO_OF_DIAMONDS(Suit.DIAMONDS, Rank.TWO),
   THREE_OF_DIAMONDS(Suit.DIAMONDS, Rank.THREE),
   FOUR_OF_DIAMONDS(Suit.DIAMONDS, Rank.FOUR),
   FIVE_OF_DIAMONDS(Suit.DIAMONDS, Rank.FIVE),
   SIX_OF_DIAMONDS(Suit.DIAMONDS, Rank.SIX),
   SEVEN_OF_DIAMONDS(Suit.DIAMONDS, Rank.SEVEN),
   EIGHT_OF_DIAMONDS(Suit.DIAMONDS, Rank.EIGHT),
   NINE_OF_DIAMONDS(Suit.DIAMONDS, Rank.NINE),
   TEN_OF_DIAMONDS(Suit.DIAMONDS, Rank.TEN),
   JACK_OF_DIAMONDS(Suit.DIAMONDS, Rank.JACK),
   QUEEN_OF_DIAMONDS(Suit.DIAMONDS, Rank.QUEEN),
   KING_OF_DIAMONDS(Suit.DIAMONDS, Rank.KING),
   ACE_OF_HEARTS(Suit.HEARTS, Rank.ACE),
   TWO_OF_HEARTS(Suit.HEARTS, Rank.TWO),
   THREE_OF_HEARTS(Suit.HEARTS, Rank.THREE),
   FOUR_OF_HEARTS(Suit.HEARTS, Rank.FOUR),
   FIVE_OF_HEARTS(Suit.HEARTS, Rank.FIVE),
   SIX_OF_HEARTS(Suit.HEARTS, Rank.SIX),
   SEVEN_OF_HEARTS(Suit.HEARTS, Rank.SEVEN),
   EIGHT_OF_HEARTS(Suit.HEARTS, Rank.EIGHT),
```

```
NINE_OF_HEARTS(Suit.HEARTS, Rank.NINE),
TEN_OF_HEARTS(Suit.HEARTS, Rank.TEN),
JACK_OF_HEARTS(Suit.HEARTS, Rank.JACK),
QUEEN_OF_HEARTS(Suit.HEARTS, Rank.QUEEN),
KING_OF_HEARTS(Suit.HEARTS, Rank.KING),
ACE_OF_SPADES(Suit.SPADES, Rank.ACE),
TWO_OF_SPADES(Suit.SPADES, Rank.TWO),
THREE_OF_SPADES(Suit.SPADES, Rank.THREE),
FOUR_OF_SPADES(Suit.SPADES, Rank.FOUR),
FIVE_OF_SPADES(Suit.SPADES, Rank.FIVE),
SIX_OF_SPADES(Suit.SPADES, Rank.SIX),
SEVEN_OF_SPADES(Suit.SPADES, Rank.SEVEN),
EIGHT_OF_SPADES(Suit.SPADES, Rank.EIGHT),
NINE_OF_SPADES(Suit.SPADES, Rank.NINE),
TEN_OF_SPADES(Suit.SPADES, Rank.TEN),
JACK_OF_SPADES(Suit.SPADES, Rank.JACK),
QUEEN_OF_SPADES(Suit.SPADES, Rank.QUEEN),
KING_OF_SPADES(Suit.SPADES, Rank.KING);

private Suit suit;

/**
 *  Return <code>Card</code>'s suit.
 *
 *  @return <code>CLUBS</code>, <code>DIAMONDS</code>, <code>HEARTS</code>,
 *  or <code>SPADES</code>
 */

public Suit suit()
{
   return suit;
}

private Rank rank;

/**
 *  Return <code>Card</code>'s rank.
 *
 *  @return <code>ACE</code>, <code>TWO</code>, <code>THREE</code>,
 *  <code>FOUR</code>, <code>FIVE</code>, <code>SIX</code>,
 *  <code>SEVEN</code>, <code>EIGHT</code>, <code>NINE</code>,
 *  <code>TEN</code>, <code>JACK</code>, <code>QUEEN</code>,
 *  <code>KING</code>.
 */

public Rank rank()
{
   return rank;
}
```

```
   Card(Suit suit, Rank rank)
   {
      this.suit = suit;
      this.rank = rank;
   }

   /**
    *  A card's suit is its membership.
    *
    *  @author Jeff Friesen
    */

   public enum Suit
   {
      CLUBS, DIAMONDS, HEARTS, SPADES
   }

   /**
    *  A card's rank is its integer value.
    *
    *  @author Jeff Friesen
    */

   public enum Rank
   {
      ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN,
      KING
   }
}
```

Listing B-3 begins with a Javadoc comment that's used to briefly describe the subsequently declared `Card` class and identify this class's author. (I briefly introduced Javadoc comments in Chapter 2.)

> **Note** One feature of Javadoc comments is the ability to embed HTML tags. These tags specify different kinds of formatting for sections of text within these comments. For example, `<code>` and `</code>` specify that their enclosed text is to be formatted as a code listing. Later in this appendix, you'll learn how to convert these comments into HTML documentation.

`Card` is an example of an *enum*, which is a special kind of class that I discussed in Chapter 6. If you haven't read that chapter, think of `Card` as a place to create and store `Card` objects that identify all 52 cards that make up a standard deck.

`Card` declares a nested `Suit` enum. (I discussed nesting in Chapter 5.) A card's suit denotes its membership. The only legal `Suit` values are `CLUBS`, `DIAMONDS`, `HEARTS`, and `SPADES`.

`Card` also declares a nested `Rank` enum. A card's rank denotes its value: `ACE`, `TWO`, `THREE`, `FOUR`, `FIVE`, `SIX`, `SEVEN`, `EIGHT`, `NINE`, `TEN`, `JACK`, `QUEEN`, and `KING` are the only legal `Rank` values.

A Card object is created when Suit and Rank objects are passed to its constructor. (I discussed constructors in Chapter 3.) For example, KING_OF_HEARTS(Suit.HEARTS, Rank.KING) combines Suit.HEARTS and Rank.KING into KING_OF_HEARTS.

Card provides a rank() method for returning a Card's Rank object. Similarly, Card provides a suit() method for returning a Card's Suit object. For example, KING_OF_HEARTS.rank() returns Rank.KING, and KING_OF_HEARTS.suit() returns Suit.HEARTS.

Listing B-4 presents the Java source code to the Deck class, which implements a deck of 52 cards.

Listing B-4. Pick a Card, Any Card

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

/**
 *  Simulate a deck of cards.
 *
 *  @author Jeff Friesen
 */

public class Deck
{
   private Card[] cards = new Card[]
   {
      Card.ACE_OF_CLUBS,
      Card.TWO_OF_CLUBS,
      Card.THREE_OF_CLUBS,
      Card.FOUR_OF_CLUBS,
      Card.FIVE_OF_CLUBS,
      Card.SIX_OF_CLUBS,
      Card.SEVEN_OF_CLUBS,
      Card.EIGHT_OF_CLUBS,
      Card.NINE_OF_CLUBS,
      Card.TEN_OF_CLUBS,
      Card.JACK_OF_CLUBS,
      Card.QUEEN_OF_CLUBS,
      Card.KING_OF_CLUBS,
      Card.ACE_OF_DIAMONDS,
      Card.TWO_OF_DIAMONDS,
      Card.THREE_OF_DIAMONDS,
      Card.FOUR_OF_DIAMONDS,
      Card.FIVE_OF_DIAMONDS,
      Card.SIX_OF_DIAMONDS,
      Card.SEVEN_OF_DIAMONDS,
      Card.EIGHT_OF_DIAMONDS,
      Card.NINE_OF_DIAMONDS,
      Card.TEN_OF_DIAMONDS,
      Card.JACK_OF_DIAMONDS,
      Card.QUEEN_OF_DIAMONDS,
      Card.KING_OF_DIAMONDS,
```

```
      Card.ACE_OF_HEARTS,
      Card.TWO_OF_HEARTS,
      Card.THREE_OF_HEARTS,
      Card.FOUR_OF_HEARTS,
      Card.FIVE_OF_HEARTS,
      Card.SIX_OF_HEARTS,
      Card.SEVEN_OF_HEARTS,
      Card.EIGHT_OF_HEARTS,
      Card.NINE_OF_HEARTS,
      Card.TEN_OF_HEARTS,
      Card.JACK_OF_HEARTS,
      Card.QUEEN_OF_HEARTS,
      Card.KING_OF_HEARTS,
      Card.ACE_OF_SPADES,
      Card.TWO_OF_SPADES,
      Card.THREE_OF_SPADES,
      Card.FOUR_OF_SPADES,
      Card.FIVE_OF_SPADES,
      Card.SIX_OF_SPADES,
      Card.SEVEN_OF_SPADES,
      Card.EIGHT_OF_SPADES,
      Card.NINE_OF_SPADES,
      Card.TEN_OF_SPADES,
      Card.JACK_OF_SPADES,
      Card.QUEEN_OF_SPADES,
      Card.KING_OF_SPADES
   };

   private List<Card> deck;

   /**
    *  Create a <code>Deck</code> of 52 <code>Card</code> objects. Shuffle
    *  these objects.
    */

   public Deck()
   {
      deck = new ArrayList<Card>();
      for (int i = 0; i < cards.length; i++)
      {
         deck.add(cards[i]);
         cards[i] = null;
      }
      Collections.shuffle(deck);
   }

   /**
    *  Deal the <code>Deck</code>'s top <code>Card</code> object.
    *
    *  @return the <code>Card</code> object at the top of the
    *  <code>Deck</code>
    */
```

```
   public Card deal()
   {
      return deck.remove(0);
   }

   /**
    *  Return an indicator of whether or not the <code>Deck</code> is empty.
    *
    *  @return true if the <code>Deck</code> contains no <code>Card</code>
    *  objects; otherwise, false
    */

   public boolean isEmpty()
   {
      return deck.isEmpty();
   }

   /**
    *  Put back a <code>Card</code> at the bottom of the <code>Deck</code>.
    *
    *  @param card <code>Card</code> object being put back
    */

   public void putBack(Card card)
   {
      deck.add(card);
   }

   /**
    *  Shuffle the <code>Deck</code>.
    */

   public void shuffle()
   {
      Collections.shuffle(deck);
   }
}
```

`Deck` initializes a private `cards` array to all 52 `Card` objects. Because it's easier to implement `Deck` via a list that stores these objects, `Deck`'s constructor creates this list and adds each `Card` object to the list. (I discussed `List` and `ArrayList` in Chapter 9.)

`Deck` also provides `deal()`, `isEmpty()`, `putBack()`, and `shuffle()` methods to deal a single `Card` from the `Deck` (the `Card` is physically removed from the `Deck`), determine whether or not the `Deck` is empty, put a `Card` back into the `Deck`, and shuffle the `Deck`'s `Card`s.

Listing B-5 presents the source code to the `DiscardPile` class, which implements a discard pile on which players can throw away a maximum of 52 cards.

Listing B-5. A Garbage Dump for Cards

```
import java.util.ArrayList;
import java.util.List;

/**
 *  Simulate a pile of discarded cards.
 *
 *  @author Jeff Friesen
 */

public class DiscardPile
{
   private Card[] cards;
   private int top;

   /**
    *  Create a <code>DiscardPile</code> that can accommodate a maximum of 52
    *  <code>Card</code>s. The <code>DiscardPile</code> is initially empty.
    */

   public DiscardPile()
   {
      cards = new Card[52]; // Room for entire deck on discard pile (should
                            // never happen).
      top = -1;
   }

   /**
    *  Return the <code>Card</code> at the top of the <code>DiscardPile</code>.
    *
    *  @return <code>Card</code> object at top of <code>DiscardPile</code> or
    *  null if <code>DiscardPile</code> is empty
    */

   public Card getTopCard()
   {
      if (top == -1)
         return null;
      Card card = cards[top];
      cards[top--] = null;
      return card;
   }

   /**
    *  Set the <code>DiscardPile</code>'s top card to the specified
    *  <code>Card</code> object.
    *
    *  @param card <code>Card</code> object being thrown on top of the
    *  <code>DiscardPile</code>
    */
```

```
   public void setTopCard(Card card)
   {
      cards[++top] = card;
   }

   /**
    *  Identify the top <code>Card</code> on the <code>DiscardPile</code>
    *  without removing this <code>Card</code>.
    *
    *  @return top <code>Card</code>, or null if <code>DiscardPile</code> is
    *  empty
    */

   public Card topCard()
   {
      return (top == -1) ? null : cards[top];
   }
}
```

`DiscardPile` implements a discard pile on which to throw `Card` objects. It implements the discard pile via a stack metaphor: the last `Card` object thrown on the pile sits at the top of the pile and is the first `Card` object to be removed from the pile.

This class stores its stack of `Card` objects in a private `cards` array. I found it convenient to specify 52 as this array's storage limit because the maximum number of `Card`s is 52. (Game play will never result in all `Card`s being stored in the array.)

Along with its constructor, `DiscardPile` provides `getTopCard()`, `setTopCard()`, and `topCard()` methods to remove and return the stack's top `Card`, store a new `Card` object on the stack as its top `Card`, and return the top `Card` without removing it from the stack.

The constructor demonstrates a single-line comment, which starts with the `//` character sequence. This comment documents that the `cards` array has room to store the entire `Deck` of `Card`s. I formally introduced single-line comments in Chapter 2.

The second step in converting Listing B-2's pseudocode to Java involves introducing a `FourOfAKind` class whose `main()` method contains the Java code equivalent of this pseudocode. Listing B-6 presents `FourOfAKind`.

Listing B-6. `FourOfAKind` Application Source Code

```
/**
 *  <code>FourOfAKind</code> implements a card game that is played between two
 *  players: one human player and the computer. You play this game with a
 *  standard 52-card deck and attempt to beat the computer by being the first
 *  player to put down four cards that have the same rank (four aces, for
 *  example), and win.
 *
 *  <p>
 *  The game begins by shuffling the deck and placing it face down. Each
 *  player takes a card from the top of the deck. The player with the highest
 *  ranked card (king is highest) deals four cards to each player starting
```

```
 *  with the other player. The dealer then starts its turn.
 *
 *  <p>
 *  The player examines its cards to determine which cards are optimal for
 *  achieving four of a kind. The player then throws away one card on a
 *  discard pile and picks up another card from the top of the deck. If the
 *  player has four of a kind, the player puts down these cards (face up) and
 *  wins the game.
 *
 *  @author Jeff Friesen
 *  @version 1.0
 */

public class FourOfAKind
{
   /**
    *  Human player
    */

   final static int HUMAN = 0;

   /**
    *  Computer player
    */

   final static int COMPUTER = 1;

   /**
    *  Application entry point.
    *
    *  @param args array of command-line arguments passed to this method
    */

   public static void main(String[] args)
   {
      System.out.println("Welcome to Four of a Kind!");
      Deck deck = new Deck(); // Deck automatically shuffled
      DiscardPile discardPile = new DiscardPile();
      Card hCard;
      Card cCard;
      while (true)
      {
         hCard = deck.deal();
         cCard = deck.deal();
         if (hCard.rank() != cCard.rank())
            break;
         deck.putBack(hCard);
         deck.putBack(cCard);
         deck.shuffle(); // prevent pathological case where every successive
      }                  // pair of cards have the same rank
      int curPlayer = HUMAN;
      if (cCard.rank().ordinal() > hCard.rank().ordinal())
```

```
      curPlayer = COMPUTER;
   deck.putBack(hCard);
   hCard = null;
   deck.putBack(cCard);
   cCard = null;
   Card[] hCards = new Card[4];
   Card[] cCards = new Card[4];
   if (curPlayer == HUMAN)
      for (int i = 0; i < 4; i++)
      {
         cCards[i] = deck.deal();
         hCards[i] = deck.deal();
      }
   else
      for (int i = 0; i < 4; i++)
      {
         hCards[i] = deck.deal();
         cCards[i] = deck.deal();
      }
   while (true)
   {
      if (curPlayer == HUMAN)
      {
         showHeldCards(hCards);
         int choice = 0;
         while (choice < 'A' || choice > 'D')
         {
            choice = prompt("Which card do you want to throw away (A, B, " +
                            "C, D)? ");
            switch (choice)
            {
               case 'a': choice = 'A'; break;
               case 'b': choice = 'B'; break;
               case 'c': choice = 'C'; break;
               case 'd': choice = 'D';
            }
         }
         discardPile.setTopCard(hCards[choice - 'A']);
         hCards[choice - 'A'] = deck.deal();
         if (isFourOfAKind(hCards))
         {
            System.out.println();
            System.out.println("Human wins!");
            System.out.println();
            putDown("Human's cards:", hCards);
            System.out.println();
            putDown("Computer's cards:", cCards);
            return; // Exit application by returning from main()
         }
         curPlayer = COMPUTER;
      }
```

```
         else
         {
            int choice = leastDesirableCard(cCards);
            discardPile.setTopCard(cCards[choice]);
            cCards[choice] = deck.deal();
            if (isFourOfAKind(cCards))
            {
               System.out.println();
               System.out.println("Computer wins!");
               System.out.println();
               putDown("Computer's cards:", cCards);
               return; // Exit application by returning from main()
            }
            curPlayer = HUMAN;
         }
         if (deck.isEmpty())
         {
            while (discardPile.topCard() != null)
               deck.putBack(discardPile.getTopCard());
            deck.shuffle();
         }
      }
   }

   /**
    *  Determine if the <code>Card</code> objects passed to this method all
    *  have the same rank.
    *
    *  @param cards array of <code>Card</code> objects passed to this method
    *
    *  @return true if all <code>Card</code> objects have the same rank;
    *  otherwise, false
    */

   static boolean isFourOfAKind(Card[] cards)
   {
      for (int i = 1; i < cards.length; i++)
         if (cards[i].rank() != cards[0].rank())
            return false;
      return true;
   }

   /**
    *  Identify one of the <code>Card</code> objects that is passed to this
    *  method as the least desirable <code>Card</code> object to hold onto.
    *
    *  @param cards array of <code>Card</code> objects passed to this method
    *
    *  @return 0-based rank (ace is 0, king is 13) of least desirable card
    */
```

```
static int leastDesirableCard(Card[] cards)
{
   int[] rankCounts = new int[13];
   for (int i = 0; i < cards.length; i++)
      rankCounts[cards[i].rank().ordinal()]++;
   int minCount = Integer.MAX_VALUE;
   int minIndex = -1;
   for (int i = 0; i < rankCounts.length; i++)
      if (rankCounts[i] < minCount && rankCounts[i] != 0)
      {
         minCount = rankCounts[i];
         minIndex = i;
      }
   for (int i = 0; i < cards.length; i++)
      if (cards[i].rank().ordinal() == minIndex)
         return i;
   return 0; // Needed to satisfy compiler (should never be executed)
}

/**
 *  Prompt the human player to enter a character.
 *
 *  @param msg message to be displayed to human player
 *
 *  @return integer value of character entered by user.
 */

static int prompt(String msg)
{
   System.out.print(msg);
   try
   {
      int ch = System.in.read();
      // Erase all subsequent characters including terminating \n newline
      // so that they do not affect a subsequent call to prompt().
      while (System.in.read() != '\n');
      return ch;
   }
   catch (java.io.IOException ioe)
   {
   }
   return 0;
}

/**
 *  Display a message followed by all cards held by player. This output
 *  simulates putting down held cards.
 *
 *  @param msg message to be displayed to human player
 *  @param cards array of <code>Card</code> objects to be identified
 */
```

```
   static void putDown(String msg, Card[] cards)
   {
      System.out.println(msg);
      for (int i = 0; i < cards.length; i++)
         System.out.println(cards[i]);
   }

   /**
    *  Identify the cards being held via their <code>Card</code> objects on
    *  separate lines. Prefix each line with an uppercase letter starting with
    *  <code>A</code>.
    *
    *  @param cards array of <code>Card</code> objects to be identified
    */

   static void showHeldCards(Card[] cards)
   {
      System.out.println();
      System.out.println("Held cards:");
      for (int i = 0; i < cards.length; i++)
         if (cards[i] != null)
            System.out.println((char) ('A' + i) + ". " + cards[i]);
      System.out.println();
   }
}
```

Listing B-6 follows the steps outlined by and expands on Listing B-2's pseudocode. Because of the various comments, I don't have much to say about this listing. However, there are a couple of items that deserve mention:

- `Card`'s nested `Rank` enum stores a sequence of 13 `Rank` objects beginning with `ACE` and ending with `KING`. These objects cannot be compared directly via > to determine which object has the greater rank. However, their integer-based ordinal (positional) values can be compared by calling the `Rank` object's `ordinal()` method. For example, `Card.ACE_OF_SPADES.rank().ordinal()` returns 0 because `ACE` is located at position 0 within `Rank`'s list of `Rank` objects, and `Card.KING_OF_CLUBS.rank().ordinal()` returns 12 because `KING` is located at the last position in this list.
- The `leastDesirableCard()` method counts the ranks of the `Card`s in the array of four `Card` objects passed to this method and stores these counts in a `rankCounts` array. For example, given two of clubs, ace of spades, three of clubs, and ace of diamonds in the array passed to this method, `rankCounts` identifies one two, two aces, and one three. This method then searches `rankCounts` from smallest index (representing ace) to largest index (representing king) for the first smallest nonzero count (there might be a tie, as in one two and one three)—a zero count represents no `Card`s having that rank in the array of `Card` objects. Finally, the method searches the array of `Card` objects to identify the object whose rank ordinal matches the index of the smallest nonzero count and returns the index of this `Card` object.This behavior implies that the least desirable card is always the

smallest ranked card. For example, given two of spades, three of diamonds, five of spades, and nine of clubs, two of spades is least desirable because it has the smallest rank.

Also, when there are multiple cards of the same rank, and when this rank is smaller than the rank of any other card in the array, this method will choose the first (in a left-to-right manner) of the multiple cards having the same rank as the least desirable card. For example, given (in this order) two of spades, two of hearts, three of diamonds, and jack of hearts, two of spades is least desirable because it's the first card with the smallest rank. However, when the rank of the multiple cards isn't the smallest, another card with the smallest rank is chosen as least desirable.

The JDK provides a `javadoc` tool that extracts all Javadoc comments from one or more source files and generates a set of HTML files containing this documentation in an easy-to-read format. These files serve as the program's documentation.

For example, suppose that the current directory contains `Card.java`, `Deck.java`, `DiscardPile.java`, and `FourOfAKind.java`. To extract all of the Javadoc comments that appear in these files, specify the following command:

```
javadoc *.java
```

The `javadoc` tool responds by outputting the following messages:

```
Loading source file Card.java...
Loading source file Deck.java...
Loading source file DiscardPile.java...
Loading source file FourOfAKind.java...
Constructing Javadoc information...
Standard Doclet version 1.7.0_06
Building tree for all the packages and classes...
Generating \Card.html...
Generating \Card.Rank.html...
Generating \Card.Suit.html...
Generating \Deck.html...
Generating \DiscardPile.html...
Generating \FourOfAKind.html...
Generating \package-frame.html...
Generating \package-summary.html...
Generating \package-tree.html...
Generating \constant-values.html...
Building index for all the packages and classes...
Generating \overview-tree.html...
Generating \index-all.html...
Generating \deprecated-list.html...
Building index for all classes...
Generating \allclasses-frame.html...
Generating \allclasses-noframe.html...
Generating \index.html...
Generating \help-doc.html...
```

Furthermore, it generates a series of files, including the `index.html` entry-point file. If you point your web browser to this file, you should see a page that is similar to the page shown in Figure B-1.

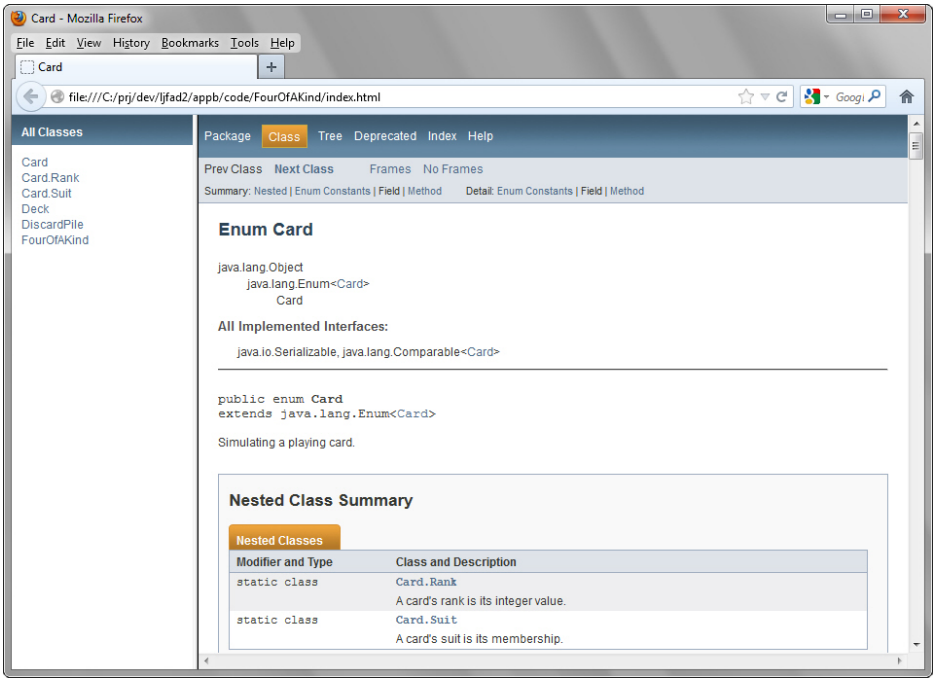

Figure B-1. *Viewing the entry-point page in the generated Javadoc for* `FourOfAKind` *and supporting classes*

`javadoc` defaults to generating HTML-based documentation for public classes and public/protected members of classes. You learned about public classes and public/protected members of classes in Chapter 3.

For this reason, `FourOfAKind`'s documentation reveals only the public `main()` method. It doesn't reveal `isFourOfAKind()` and the other package-private methods. If you want to include these methods in the documentation, you must specify `-package` with `javadoc`:

```
javadoc -package *.java
```

> **Note** The standard class library's documentation from Oracle was also generated by `javadoc` and adheres to the same format.

Compiling, Running, and Distributing FourOfAKind

Unlike Chapter 1's `DumpArgs` and `EchoText` applications, which each consist of one source file, `FourOfAKind` consists of `Card.java`, `Deck.java`, `DiscardPile.java`, and `FourOfAKind.java`. You can compile all four source files via the following command line:

```
javac FourOfAKind.java
```

The `javac` tool launches the Java compiler, which recursively compiles the source files of the various classes it encounters during compilation. Assuming successful compilation, you should end up with six classfiles in the current directory.

> **Tip** You can compile all Java source files in the current directory by specifying `javac *.java`.

After successfully compiling `FourOfAKind.java` and the other three source files, specify the following command line to run this application:

```
java FourOfAKind
```

In response, you see an introductory message and the four cards that you are holding. The following output reveals a single session:

```
Welcome to Four of a Kind!

Held cards:
A. SIX_OF_CLUBS
B. QUEEN_OF_DIAMONDS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. NINE_OF_HEARTS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. FOUR_OF_DIAMONDS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES
```

```
Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. KING_OF_HEARTS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. QUEEN_OF_CLUBS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. KING_OF_DIAMONDS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. TWO_OF_HEARTS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. FIVE_OF_DIAMONDS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. JACK_OF_CLUBS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES
```

```
Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. TWO_OF_SPADES
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Human wins!

Human's cards:
SIX_OF_CLUBS
SIX_OF_DIAMONDS
SIX_OF_HEARTS
SIX_OF_SPADES

Computer's cards:
SEVEN_OF_HEARTS
TEN_OF_HEARTS
SEVEN_OF_CLUBS
SEVEN_OF_DIAMONDS
```

Although *Four of a Kind* isn't much of a card game, you might decide to share the `FourOfAKind` application with a friend. However, if you forget to include even one of the application's five supporting classfiles, your friend will not be able to run the application.

You can overcome this problem by bundling `FourOfAKind`'s six classfiles into a single *JAR* (Java ARchive) file, which is a ZIP file that contains a special directory and the `.jar` file extension. You can then distribute this single JAR file to your friend.

The JDK provides the `jar` tool for working with JAR files. To bundle all six classfiles into a JAR file named `FourOfAKind.jar`, you could specify the following command line, where `c` tells `jar` to create a JAR file and `f` identifies the JAR file's name:

```
jar cf FourOfAKind.jar *.class
```

After creating the JAR file, try to run the application via the following command line:

```
java -jar FourOfAKind.jar
```

Instead of the application running, you'll receive an error message having to do with the `java` application launcher tool not knowing which of the JAR file's six classfiles is the *main classfile* (the file whose class's `main()` method executes first).

You can provide this knowledge via a text file that's merged into the JAR file's *manifest*, a special file named `MANIFEST.MF` that stores information about the contents of a JAR file and is stored in the JAR file's `META-INF` directory. Consider Listing B-7.

Listing B-7. Identifying the Application's Main Class

```
Main-Class: FourOfAKind
```

Listing B-7 tells `java` which of the JAR's classfiles is the main classfile. (You must leave a blank line after `Main-Class: FourOfAKind`.)

The following command line, which creates `FourOfAKind.jar`, includes `m` and the name of the text field providing manifest content:

```
jar cfm FourOfAKind.jar manifest *.class
```

This time, `java -jar FourOfAKind.jar` succeeds and the application runs because `java` is able to identify `FourOfAKind` as the main classfile.

> **Note** Now that you've finished this book, you're ready to dig into Android app development. Check out Apress's *Beginning Android* and *Android Recipes* books for guidance. After you've learned some app development basics, perhaps you might consider transforming *Four of a Kind* into an Android app.

Index

A

B

C

D

E

F

G

H

I

J, K

L

M

N

O

P

Q

R

S

T

U

V

W

X, Y

Z