Appendix

Solutions to Exercises

Chapters 1 through 10 close with an "Exercises" section that tests your understanding of the chapter's material through various exercises. 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. The purpose of the JDK's `jar` tool is to create new JAR files, update existing JAR files, and extract files from existing JAR files.

15. *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 definitions (screen or file).

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.

17. Two popular IDEs are NetBeans and Eclipse.

18. *Pseudocode* is a compact and informal high-level description of the problem domain.

19. You would use the `jar` tool along with its `t` (table of contents) and `f` (JAR file's name) options to list `FourOfAKind.jar`'s table of contents; for example, `jar tf FourOfAKind.jar`.

20. Listing 1 presents the `FourOfAKind` application's refactored `FourOfAKind` class that was called for in Chapter 1.

Listing 1. ***Letting the human player pick up the top card from the discard pile or the deck***

```
/**
 *  <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.
 *
 *  @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);
         if (discardPile.topCard() != null)
         {
            System.out.println("Discard pile top card: " +
                               discardPile.topCard());
            System.out.println();
         }
         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';
            }
         }
         Card card = null;
         if (discardPile.topCard() != null)
         {
            int dest = 0;
            while (dest != 'D' && dest != 'P')
            {
               dest = prompt("Pick up top card from deck or discard pile " +
                             "(D, P)? ");
               switch (dest)
               {
                  case 'd': dest = 'D';
                  case 'p': dest = 'P';
               }
            }
            card = (dest == 'D') ? deck.deal() : discardPile.getTopCard();
         }
         else
            card = deck.deal();
         discardPile.setTopCard(hCards[choice-'A']);
         hCards[choice-'A'] = card;
         card = null;
         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();
   }
}
```

Chapter 2: Learning Language Fundamentals

1. A class declaration contains field declarations, method declarations, constructor declarations, and other initializer (instance and class) declarations.
2. Identifier `transient` is a reserved word in Java. Identifier `delegate` is not a reserved word in Java.
3. A *variable* is a memory location whose value can change.
4. Character is Java's only unsigned primitive type. It is represented in source code via the `char` reserved word.
5. The difference between an instance field and a class field is that each object (instance) gets its own copy of an instance field, whereas all objects share a single copy of a class field.
6. An *array* is a multivalued variable in which each element holds one of these values.
7. You declare a one-dimensional array variable with a single set of square brackets, as in `String[] cities;`. You declare a two-dimensional array variable with two sets of square brackets, as in `double[][] temperatures;`.
8. *Scope* refers to a variable's accessibility. For example, the scope of a `private` field is restricted to the class in which it is declared. Also, the scope of a parameter is restricted to the method in which the parameter is declared. Another word for scope is *visibility*.
9. 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.`".
10. 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.
11. The `new` operator is used to create an object.
12. You cannot nest multiline comments.
13. The answer is true: when declaring a method that takes a variable number of arguments, you must specify the three consecutive periods just after the rightmost parameter's type name.
14. Given a two-dimensional array *x*, `x.length` returns the number of rows in the array.

15. The difference between the while and do-while statements is that a while statement performs zero or more iterations, whereas a do-while statement performs one or more iterations.

16. Initializing the `sines` array using the `new` syntax yields `double[] sines = new double[360];`. Initializing the `cosines` array using the `new` syntax yields `double[] cosines = new double[360];`.

17. It is okay for an expression assigned to an instance field to access a class field that is declared after the instance field because all class fields are initialized before any instance fields are initialized. The compiler knows that the virtual machine will know about the class fields before an object is created. As a result, this situation does not result in an illegal forward reference.

18. Creating an array of objects requires that you first use `new` to create the array, and then assign an object reference to each of the array's elements.

19. 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 a parameter's name with `this` or the class name followed by the member access operator.

20. You chain together instance method calls by having each participating method specify the name of the class in which the method is declared as the method's return type, and by having the method return `this`.

21. 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 2 presents the source code.

Listing 2. *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);
}
```

As with the `Math` class's various `static` methods, the `gcd()` method is declared to be `static` because it does not rely on any instance fields.

22. Merging the various `CheckingAccount` code fragments into a complete application results in something similar to Listing 3.

Listing 3. *A* `CheckingAccount` *class that is greater than the sum of its code fragments*

```
public class CheckingAccount
{
   private String owner;
   private int balance;
   public static int counter;
   public CheckingAccount(String acctOwner, int acctBalance)
   {
      owner = acctOwner;
      balance = acctBalance;
      counter++; // keep track of created CheckingAccount objects
   }
   public CheckingAccount(String acctOwner)
   {
      this(acctOwner, 100); // New account requires $100 minimum balance
   }
   public CheckingAccount printBalance()
   {
      System.out.println(owner+"'s balance:");
      int magnitude = (balance < 0) ? -balance : balance;
      String balanceRep = (balance < 0) ? "(" : "";
      balanceRep += magnitude;
      balanceRep += (balance < 0) ? ")" : "";
      System.out.println(balanceRep);
      return this;
   }
   public CheckingAccount deposit(int amount)
   {
      if (amount <= 0.0)
         System.out.println("cannot deposit a negative or zero amount");
      else
         balance += amount;
      return this;
   }
   public CheckingAccount withdraw(int amount)
   {
      if (amount <= 0.0)
         System.out.println("cannot deposit a negative or zero amount");
      else
      if (balance-amount < 0)
         System.out.println("cannot withdraw more funds than are available");
      else
         balance -= amount;
      return this;
   }
   public static void main(String[] args)
   {
      new CheckingAccount("Jane Doe", 1000).withdraw(2000).printBalance();
      CheckingAccount ca = new CheckingAccount("John Doe");
      ca.printBalance().withdraw(50).printBalance().deposit(80).printBalance();
      System.out.println ("Number of created CheckingAccount objects = "+
                          ca.counter);
   }
}
```

Chapter 3: Learning Object-Oriented Language Features

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 does not 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 does not 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.
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 does not 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 does not 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, 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 in order 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 upon a has-a relationship between them.
30. The answer is false: composition is used to implement "has-a" relationships and implementation inheritance is used to implement "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 (such as shape, animal, or vehicle) and generic operations (such as 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. The three forms of RTTI are the virtual machine verifying that a cast is legal, using the `instanceof` operator to determine if an instance is a member of a type, and reflection.

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.
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 4 through 10 declare the `Animal`, `Bird`, `Fish`, `AmericanRobin`, `DomesticCanary`, `RainbowTrout`, and `SockeyeSalmon` classes that were called for in Chapter 3.

Listing 4. ***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 5. *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 6. *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 7. *The `AmericanRobin` class denoting a bird with a red breast*

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

Listing 8. *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 9. *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 10. *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 does not 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 does not 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 does not make sense to subclass them.

50. Listing 11 declares the `Animals` class that was called for in Chapter 3.

Listing 11. *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 12 through 14 declare the `Countable` interface, the modified `Animal` class, and the modified `Animals` class that were called for in Chapter 3.

Listing 12. ***The `Countable` interface for use in taking a census of animals***

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

Listing 13. ***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 14. ***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 4: 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 does not 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 is 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, if 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 do not 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.
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 (such as not being able to open a file) and from flawed code (such as 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 Java's 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 clause 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 clause is to provide cleanup code that is executed whether an exception is thrown or not.

32. Listing 15 presents the `G2D` class that was called for in Chapter 3.

Listing 15. ***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 to support a null device in which messages are thrown away, first introduce Listing 16's `NullDevice` package-private class.

Listing 16. ***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 17.

Listing 17. *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 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 18's `Logger` interface.

Listing 18. *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 19 presents the `CannotConnectException` class.

Listing 19. *An uncomplicated* `CannotConnectException` *class*

```
package logging;

public class CannotConnectException extends Exception
{
}
```

The `NotConnectedException` class has a similar structure.

Listing 20 presents the `Console` class.

Listing 20. *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 21 presents the `File` class.

Listing 21. *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 22.

Listing 22. ***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 5: 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 is 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 (such as `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 `ClassCastExceptions`.
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 (such as `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. The answer is false: `MyList<E super Circle>` does not specify that the `E` type parameter has a lower bound of `Circle`. In contrast, `MyList<? super Circle>` specifies that `Circle` is a lower bound.
25. A *recursive type bound* is a type parameter bound that includes the type parameter.
26. 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<?>.

27. *Reification* is the process or result of treating the abstract as if it was concrete.

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

29. *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 (such as Object), and inserting casts to the appropriate type when the resulting code is not type correct.

30. A *generic method* is a static or non-static method with a type-generalized implementation.

31. Although you might think otherwise, Listing 5-43'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 (if there is no bound). After erasure, we are left with Listing 23's nongeneric methodCaller() method.

Listing 23. ***The nongeneric methodCaller() method that results from erasure***

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

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

33. 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.

34. An *enum* is an enumerated type that is expressed via reserved word enum.

35. 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.

36. 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.

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

38. 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.

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

40. Listing 24 presents a ToDo marker annotation type that annotates only type elements, and that also uses the default retention policy.

Listing 24. *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
{
}
```

41. Listing 25 presents a rewritten StubFinder application that works with Listing 5-15's Stub annotation type (with appropriate @Target and @Retention annotations) and Listing 5-16's Deck class.

Listing 25. *Reporting a stub's ID, due date, and developer via a new version of StubFinder*

```
import java.lang.reflect.*;

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();
         }
   }
}
```

42. Listing 26 presents the generic Stack class and the StackEmptyException and StackFullException helper classes that were called for in Chapter 5.

Listing 26. *`Stack` and its `StackEmptyException` and `StackFullException` helper classes proving that not all helper classes need to be nested*

```
class StackEmptyException extends Exception
{
}
class StackFullException extends Exception
{
}
public class Stack<E>
{
   private E[] elements;
   private int top;
   @SuppressWarnings("unchecked")
   public Stack(int size)
   {
      elements = (E[]) new Object[size];
      top = -1;
   }
   public 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--];
   }
   public 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();
   }
}
```

43. Listing 27 presents the `Compass` enum that was called for in Chapter 5.

Listing 27. *A `Compass` enum with four direction constants*

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

Listing 28 presents the `UseCompass` class that was called for in Chapter 5.

Listing 28. *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 6: 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 does not 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).

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 (such as 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 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.
11. 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.
12. The two main uses of the primitive wrapper classes are to store objects containing primitive values in the collections framework's lists, sets, and maps; and to provide a good place to associate useful constants (such as `MAX_VALUE` and `MIN_VALUE`) and class methods (such as `Integer`'s `parseInt()` methods and `Character`'s `isDigit()`, `isLetter()`, and `toUpperCase()` methods) with the primitive types.
13. You should avoid coding expressions such as `ch >= '0' && ch <= '9'` (test `ch` to see if it contains a digit) or `ch >= 'A' && ch <= 'Z'` (test `ch` to see if it contains an uppercase letter) because it is too easy to introduce a bug into the expressions, the expressions are not very descriptive of what they are testing, and the expressions are biased toward Latin digits (0–9) and letters (A–Z, a–z).
14. The four kinds of reachability are strongly reachable, softly reachable, weakly reachable, and phantom reachable.

15. A *referent* is the object whose reference is stored in a SoftReference, WeakReference, or PhantomReference object.

16. The References API's PhantomReference class is the equivalent of Object's finalize() method. Both entities are used to perform object cleanup tasks.

17. Listing 29 presents the Circle application that was called for in Chapter 6.

Listing 29. *Using asterisks to display a circle shape*

```
public class Circle
{
   final static int NROWS = 22;
   final static int NCOLS = 22;
   final static double RADIUS = 10.0;
   public static void main(String[] args)
   {
      // Create the screen array for storing the cardioid image.
      char[][] screen = new char[NROWS][];
      for (int row = 0; row < NROWS; row++)
         screen[row] = new char[NCOLS];

      // Initialize the screen array to space characters.
      for (int col = 0; col < NCOLS; col++)
         screen[0][col] = ' ';
      for (int row = 1; row < NROWS; row++)
         System.arraycopy(screen[0], 0, screen[row], 0, NCOLS);

      // Create the circle shape.
      for (int angle = 0; angle < 360; angle++)
      {
         int x = (int)(RADIUS*Math.cos(Math.toRadians(angle)))+NCOLS/2;
         int y = (int)(RADIUS*Math.sin(Math.toRadians(angle)))+NROWS/2;
         screen[y][x] = '*';
      }

      // Output the screen array.
      for (int row = 0; row < NROWS; row++)
         System.out.println(screen[row]);
   }
}
```

18. Listing 30 presents the PrimeNumberTest application that was called for in Chapter 6.

Listing 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");
      }
   }
}
```

Chapter 7: Exploring the Basic APIs Part 2

1. *Reflection* is a third form of runtime type identification. Applications use reflection to learn about loaded classes, interfaces, enums (a kind of class), and annotation types (a kind of interface); and to instantiate classes, call methods, access fields, and perform other tasks reflectively.

2. The difference between `Class`'s `getDeclaredFields()` and `getFields()` methods is as follows: `getDeclaredFields()` returns an array of `Field` objects representing all public, protected, default (package) access, and private fields declared by the class or interface represented by this `Class` object while excluding inherited fields, whereas `getFields()` returns an array of `Field` objects representing public fields of the class or interface represented by this `Class` object, including those public fields inherited from superclasses and superinterfaces.

3. You would determine if the method represented by a `Method` object is abstract by calling the object's `getModifiers()` method, bitwise ANDing the return value with `Modifier.ABSTRACT`, and comparing the result with `Modifier.ABSTRACT`. For example, `((method.getModifiers() & Modifier.ABSTRACT) == Modifier.ABSTRACT)` evaluates to true when the method represented by the `Method` object whose reference is stored in `method` is abstract.

4. The three ways of obtaining a `Class` object are to use `Class`'s `forName()` method, `Object`'s `getClass()` method, and a class literal.

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

6. 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.
7. `String` and `StringBuffer` differ in that `String` objects contain immutable sequences of characters, whereas `StringBuffer` objects contain mutable sequences of characters.
8. `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.
9. `System`'s `arraycopy()` method copies all or part of one array's elements to another array.
10. A *thread* is an independent path of execution through an application's code.
11. The purpose of the `Runnable` interface is to identify those objects that supply code for threads to execute via this interface's solitary `public void run()` method.
12. 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.
13. The answer is false: a `Thread` object associates with a single thread.
14. 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.
15. *Synchronization* is the act of allowing only one thread at time to execute code within a method or a block.
16. Synchronization is implemented in terms of monitors and locks.
17. 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.
18. The answer is true: variables of type `long` or `double` are not atomic on 32-bit virtual machines.

19. 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.

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

21. *Deadlock* is a situation where 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.

22. The purpose of the ThreadLocal class is to associate per-thread data (such as a user ID) with a thread.

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

24. Listing 31 presents a more efficient version of Listing 6-14's image names loop.

Listing 31. ***A more efficient image names loop***

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

25. Listing 32 presents the Classify application that was called for in Chapter 7.

Listing 32. ***Classifying a command-line argument as an annotation type, enum, interface, or class***

```
public class Classify
{
   public static void main(String[] args)
   {
      if (args.length != 1)
      {
         System.err.println("usage: java Classify pkgAndTypeName");
         return;
      }
      try
      {
         Class<?> clazz = Class.forName(args[0]);
         if (clazz.isAnnotation())
            System.out.println("Annotation");
         else
         if (clazz.isEnum())
            System.out.println("Enum");
         else
         if (clazz.isInterface())
            System.out.println("Interface");
         else
            System.out.println("Class");
      }
      catch (ClassNotFoundException cnfe)
      {
         System.err.println("could not locate " + args[0]);
      }
   }
}
```

Specify `java Classify java.lang.Override` and you will see `Annotation` as the output. Also, `java.Classify java.math.RoundingMode` outputs `Enum`, `java Classify java.lang.Runnable` outputs `Interface`, and `java Classify java.lang.Class` outputs `Class`.

26. Listing 33 presents the revised `ExploreType` application that was called for in Chapter 7.

Listing 33. *An improved `ExploreType` application*

```
public class ExploreType
{
   public static void main(String[] args)
   {
      if (args.length != 1)
      {
         System.err.println("usage: java ExploreType pkgAndTypeName");
         return;
      }
      try
      {
         Class<?> clazz = Class.forName(args[0]);
         if (clazz.isAnnotation())
            dumpAnnotation(clazz);
         else
         if (clazz.isEnum())
            dumpEnum(clazz);
         else
         if (clazz.isInterface())
            dumpInterface(clazz);
         else
            dumpClass(clazz);
      }
      catch (ClassNotFoundException cnfe)
      {
         System.err.println("could not locate " + args[0]);
      }
   }
   public static void dumpAnnotation(Class clazz)
   {
      // Left blank as an exercise for you to complete.
   }
   public static void dumpClass(Class clazz)
   {
      // Output class header.
      int modifiers = clazz.getModifiers();
      if ((modifiers & Modifier.PUBLIC) == Modifier.PUBLIC)
         System.out.print("public ");
      if ((modifiers & Modifier.ABSTRACT) == Modifier.ABSTRACT)
         System.out.print("abstract ");
      System.out.println("class " + clazz.getName());
      System.out.println("{");

      // Output fields.
      System.out.println ("   // FIELDS");
      Field[] fields = clazz.getDeclaredFields();
      for (int i = 0; i < fields.length; i++)
      {
         System.out.print("   ");
         System.out.println(fields[i]);
      }
      System.out.println();

      // Output constructors.
      System.out.println ("   // CONSTRUCTORS");
      Constructor[] constructors = clazz.getDeclaredConstructors();
      for (int i = 0; i < constructors.length; i++)
      {
         System.out.print("   ");
         System.out.println(constructors[i]);
      }
      System.out.println();

      // Output methods.
      System.out.println ("   // METHODS");
      Method[] methods = clazz.getDeclaredMethods();
      for (int i = 0; i < methods.length; i++)
      {
         System.out.print("   ");
         System.out.println(methods[i]);
      }

      // Output class trailer.
      System.out.println("}");
   }
   public static void dumpEnum(Class clazz)
   {
      // Left blank as an exercise for you to complete.
   }
   public static void dumpInterface(Class clazz)
   {
      // Left blank as an exercise for you to complete.
   }
}
```

I have deliberately written this application so that it can be expanded to output annotation types, enums, and interfaces.

27. Listing 34 presents the revised `CountingThreads` application that was called for in Chapter 7.

Listing 34. *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 35 presents the `StopCountingThreads` application that was called for in Chapter 7.

Listing 35. *Stopping the counting threads when Enter is pressed*

```
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;
```

```
    }
}
```

Chapter 8: Discovering 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 standardized 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 value in an object of a primitive wrapper class type 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 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 (such as `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.

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. The answer is false: `HashSet` is not an example of a sorted set. However, `TreeSet` is an example of a sorted set.
33. 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 does not implement `Comparable`.
34. 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.
35. The answer is true: `Queue`'s `element()` method throws `NoSuchElementException` when it is called on an empty queue.
36. 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.
37. A *map* is a group of key/value pairs (also known as *entries*).
38. 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.
39. The `HashMap` class provides a map implementation that is based on a hashtable data structure.
40. A hashtable uses a *hash function* to map keys to integer values.
41. 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*.
42. A hashtable's *capacity* refers to the number of buckets.
43. A hashtable's *load factor* refers to the ratio of the number of stored entries divided by the number of buckets.

44. 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.

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

46. The WeakHashMap class provides a Map implementation that is based on weakly reachable keys.

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

50. 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.)

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

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

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

54. Listing 36 presents the JavaQuiz application's JavaQuiz source file that was called for in Chapter 8.

Listing 36. ***How much do you know about Java? Take the quiz and find out!***

```
public class JavaQuiz
{
   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");
      }
   }
}
```

JavaQuiz first creates a list of quiz entries. In a more sophisticated application, I would obtain quiz data from a database and dynamically create the entries. JavaQuiz then performs the quiz with the help of iterator() and its returned Iterator instance's hasNext() and next() methods.

Listing 37 reveals the companion QuizEntry class.

Listing 37. *A helper class for storing a quiz's data*

```
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;
   }
}
```

`QuizEntry` is a reusable class that stores quiz data. I did not nest `QuizEntry` in `JavaQuiz` because `QuizEntry` is useful for all kinds of quizzes. However, I made this class package-private by not declaring `QuizEntry` to be a public class because it is a helper class to a quiz's main class (such as `JavaQuiz`).

55. `(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 does not affect the hash code, whereas `>>` shifts a 0 or a 1 to the right, which affects the hash code when a 1 is shifted.

56. Listing 38 presents the `FrequencyDemo` application that was called for in Chapter 8.

Listing 38. ***Reporting the frequency of last command-line argument occurrences in the previous command-line arguments***

```
import java.util.LinkedList;
import java.util.Collections;
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 9: Discovering 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 does not return a value); `Executor` does not 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` does not 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 cancelling 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 `BlockingQueue` (a subinterface of `java.util.Queue` that describes a first-in, first-out data structure, and provides additional operations that wait for the queue to become nonempty when retrieving an element, and wait for space to become available in the queue when storing an element); the `ArrayBlockingQueue`, `LinkedBlockingQueue`, `PriorityBlockingQueue`, and `SynchronousQueue` classes that implement `BlockingQueue`; `ConcurrentMap` (a subinterface of `java.util.Map` that declares additional atomic `putIfAbsent()`, `remove()`, and `replace()` methods); the `ConcurrentHashMap` class that implements `ConcurrentMap`; and the `ConcurrentLinkedQueue` class (an unbounded thread-safe FIFO implementation of the `Queue` interface).

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. *Internationalization* is the process of creating an application that automatically adapts to its current user's culture so that the user can read the application's text, hear audio clips in the user's language (if audio is supported), and so on. This word is commonly abbreviated to *i18n*, with 18 representing the number of letters between the initial *i* and the final *n*.

16. A *locale* is a geographical, political, or a cultural region.

17. The components of a `Locale` object are a language code, an optional country code, and an optional variant code.

18. A *resource bundle* is a container that holds one or more locale-specific elements, and which is associated with one and only one locale.

19. The answer is true: if a property resource bundle and a list resource bundle have the same complete resource bundle name, the list resource bundle takes precedence over the property resource bundle.
20. A *break iterator* is an object that detects logical boundaries within a section of text.
21. The Break Iterator API supports character, word, sentence, and line break iterators.
22. The answer is false: you cannot pass any `Locale` object to any of `BreakIterator`'s factory methods that take `Locale` arguments. Instead, you can only pass `Locale` objects for locales that are identified by `BreakIterator`'s `getAvailableLocales()` method.
23. A *collator* is a `Collator` instance that performs locale-specific comparisons for sorting purposes. For example, a `Collator` for the fr_FR locale takes into account accented characters by first comparing words as if none of the characters contain accents, and then comparing equal words from right to left for accents.
24. A *date* is a recorded temporal moment, a *time zone* is a set of geographical regions that share a common number of hours relative to Greenwich Mean Time (GMT), and a *calendar* is a system of organizing the passage of time.
25. The answer is true: `Date` instances can represent dates prior to or after the Unix epoch.
26. You would obtain a `TimeZone` object that represents Central Standard Time by calling `TimeZone`'s `static TimeZone getTimeZone(String ID)` factory method with argument `"CST"`.
27. Assuming that `cal` identifies a `Calendar` instance and `locale` identifies a specific locale, you would obtain a localized name for the month represented by `cal` by calling `cal.getDisplayName(Calendar.MONTH, Calendar.LONG, locale)` to return the long form of the month name, or by calling `cal.getDisplayName(Calendar.MONTH, Calendar.SHORT, locale)` to return the short form of the month name.
28. A *formatter* is an instance of a class that subclasses `Format`.
29. `NumberFormat` returns formatters that format numbers as currencies, integers, numbers with decimal points, and percentages (and also to parse such values).
30. The answer is false: `DateFormat`'s `getInstance()` factory method is a shortcut to obtaining a default date/time formatter that uses the `SHORT` style for both the date and the time.

31. A *message formatter* lets you convert a *compound message pattern* (a template consisting of static text and brace-delimited placeholders) along with the variable data required by the pattern's placeholders into a localized message.

32. A *preference* is a configuration item.

33. The Properties API is problematic for persisting preferences because properties files tend to grow in size and the probability of name collisions among the various keys increases; a growing application tends to acquire numerous properties files with each part of the application associated with its own properties file (and the names and locations of these properties files must be hard-coded in the application's source code); someone could directly modify these text-based properties files (perhaps inserting gibberish) and cause the application that depends upon the modified properties file to crash unless it is properly coded to deal with this possibility; and properties files cannot be used on diskless computing platforms.

34. The Preferences API persists preferences by storing them in platform-specific storage facilities (such as the Windows registry). Preferences are stored in trees of nodes, which are the analogue of a hierarchical filesystem's directories. Also, preference name/value pairs stored under these nodes are the analogues of a directory's files. There are two kinds of trees: system and user. All users share the *system preference tree*, whereas the *user preference tree* is specific to a single user, which is generally the person who logged into the underlying operating system.

35. 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*.

36. 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.

37. 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.

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

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

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

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

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

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

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

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

46. 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.

47. 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.

48. Listing 39 presents the `SpanishCollation` application that was called for in Chapter 9.

Listing 39. ***Outputting Spanish words according to this language's current collation rules followed by its traditional collation rules***

```
import java.text.Collator;
import java.text.ParseException;
import java.text.RuleBasedCollator;

import java.util.Arrays;
import java.util.Locale;

public class SpanishCollation
{
   public static void main(String[] args)
   {
      String[] words =
      {
         "ñango",   // weak
         "llamado", // called
         "lunes",   // monday
         "champán", // champagne
         "clamor",  // outcry
         "cerca",   // near
         "nombre",  // name
         "chiste",  // joke
      };
      Locale locale = new Locale("es", "");
      Collator c = Collator.getInstance(locale);
      Arrays.sort(words, c);
      for (String word: words)
         System.out.println(word);
      System.out.println();
      // Define the traditional Spanish sort rules.
      String upperNTilde = new String ("\u00D1");
      String lowerNTilde = new String ("\u00F1");
      String spanishRules = "< a,A < b,B < c,C < ch, cH, Ch, CH < d,D < e,E " +
                            "< f,F < g,G < h,H < i,I < j,J < k,K < l,L < ll, " +
                            "lL, Ll, LL < m,M < n,N < " + lowerNTilde + "," +
                            upperNTilde + " < o,O < p,P < q,Q < r,R < s,S < " +
                            "t,T < u,U < v,V < w,W < x,X < y,Y < z,Z";
      try
      {
         c = new RuleBasedCollator(spanishRules);
         Arrays.sort(words, c);
         for (String word: words)
            System.out.println(word);
      }
      catch (ParseException pe)
      {
         System.err.println(pe);
      }
   }
}
```

49. Listing 40 presents the `RearrangeText` application that was called for in Chapter 9.

Listing 40. *Rearranging a single text argument of the form* `x, y` *into the form* `y x`

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

public class RearrangeText
{
   public static void main(String[] args)
   {
      if (args.length != 1)
      {
         System.err.println("usage: java RearrangeText text");
         return;
      }
      try
      {
         Pattern p = Pattern.compile("(.*), (.*)");
         Matcher m = p.matcher(args[0]);
         if (m.matches())
            System.out.println(m.group(2)+" " + m.group(1));
      }
      catch (PatternSyntaxException pse)
      {
         System.err.println(pse);
      }
   }
}
```

50. Listing 41 presents the `ReplaceText` application that was called for in Chapter 9.

Listing 41. *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 10: Performing 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* (such as 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 does not start with the root directory symbol; it is 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 static` 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 is absolute and unique.
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 is not a directory and satisfies other platform-dependent criteria: it is 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 does not 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 if it does not exist in a single operation that is 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 does not crash or the power is not 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 is not lost when the system crashes. No guarantee is made when the file does not 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 does not 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 (such as a part in a parts database) and a *field* stores a single attribute of the entry (such as 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 (for example, a file), flushing the stream only guarantees that bytes previously written to the stream are passed to the underlying platform for writing; it does not guarantee that they are 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. Do not 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` do not 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 is okay to serialize objects of the implementing class.

40. When the serialization mechanism encounters an object whose class does not 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 do not 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 does not detect a public noargument constructor.

53. The difference between PrintStream's print() and println() methods is that the print() methods do not 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 (such as a carriage return followed by a newline/line feed on Windows, or only a newline/line feed on Unix/Linux).

55. The answer is true: PrintStream's %tR format specifier is used to format a Calendar object's time as HH:MM.

56. 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.

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

58. The answer is false: Reader does not declare an available() method.

59. 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.

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

61. 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.

62. Listing 42 presents the Touch application that was called for in Chapter 10.

Listing 42. *Setting a file or directory's timestamp to the current or specified time*

```
import java.io.File;

import java.text.ParseException;
import java.text.SimpleDateFormat;

import java.util.Date;

public class Touch
{
   public static void main(String[] args)
   {
      if (args.length != 1 && args.length != 3)
      {
         System.err.println("usage: java Touch [-d timestamp] pathname");
         return;
      }
      long time = new Date().getTime();
      if (args.length == 3)
      {
         if (args[0].equals("-d"))
         {
            try
            {
               SimpleDateFormat sdf;
               sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss z");
               Date date = sdf.parse(args[1]);
               time = date.getTime();
            }
            catch (ParseException pe)
            {
               pe.printStackTrace();
            }
         }
         else
         {
            System.err.println("invalid option: " + args[0]);
            return;
         }
      }
      new File(args[args.length == 1 ? 0 : 2]).setLastModified(time);
   }
}
```

63. Listing 43 presents the `Media` class that was called for in Chapter 10.

Listing 43. *Obtaining the data from an MP3 file's 128-byte ID3 block, and creating/populating/returning an `ID3` object with this data*

```
import java.io.IOException;
import java.io.RandomAccessFile;

public class Media
{
   public static class ID3
   {
      private String songTitle, artist, album, year, comment, genre;
      private int track; // -1 if track not present
      public ID3(String songTitle, String artist, String album, String year,
                 String comment, int track, String genre)
      {
         this.songTitle = songTitle;
         this.artist = artist;
         this.album = album;
         this.year = year;
         this.comment = comment;
         this.track = track;
         this.genre = genre;
      }
      String getSongTitle() { return songTitle; }
      String getArtist() { return artist; }
      String getAlbum() { return album; }
      String getYear() { return year; }
      String getComment() { return comment; }
      int getTrack() { return track; }
      String getGenre() { return genre; }
   }
   public static ID3 getID3Info(String mp3path) throws IOException
   {
      RandomAccessFile raf = null;
      try
      {
         raf = new RandomAccessFile(mp3path, "r");
         if (raf.length() < 128)
            return null; // Not MP3 file (way too small)
         raf.seek(raf.length()-128);
         byte[] buffer = new byte[128];
         raf.read(buffer);
         raf.close();
         if (buffer[0] != (byte) 'T' && buffer[1] != (byte) 'A' &&
             buffer[2] != (byte) 'G')
            return null; // No ID3 block (must start with TAG)
         String songTitle = new String(buffer, 3, 30);
         String artist = new String(buffer, 33, 30);
         String album = new String(buffer, 63, 30);
         String year = new String(buffer, 93, 4);
         String comment = new String(buffer, 97, 28);
         // buffer[126]&255 converts -128 through 127 to 0 through 255
         int track = (buffer[125] == 0) ? buffer[126]&255 : -1;
         String[] genres = new String[]
                          {
                             "Blues",
                             "Classic Rock",
                             "Country",
                             "Dance",
                             "Disco",
                             "Funk",
                             "Grunge",
                             "Hip-Hop",
                             "Jazz",
                             "Metal",
                             "New Age",
                             "Oldies",
                             "Other",
                             "Pop",
                             "R&B",
                             "Rap",
                             "Reggae",
                             "Rock",
                             "Techno",
                             "Industrial",
                             "Alternative",
                             "Ska",
                             "Death Metal",
                             "Pranks",
"Soundtrack",
"Euro-Techno",
"Ambient",
"Trip-Hop",
"Vocal",
"Jazz+Funk",
"Fusion",
"Trance",
"Classical",
"Instrumental",
"Acid",
"House",
"Game",
"Sound Clip",
"Gospel",
"Noise",
"AlternRock",
"Bass",
"Soul",
"Punk",
"Space",
"Meditative",
"Instrumental Pop",
"Instrumental Rock",
"Ethnic",
"Gothic",
"Darkwave",
"Techno-Industrial",
"Electronic",
"Pop-Folk",
"Eurodance",
"Dream",
"Southern Rock",
"Comedy",
"Cult",
"Gangsta",
"Top 40",
"Christian Rap",
"Pop/Funk",
"Jungle",
"Native American",
"Cabaret",
"New Wave",
"Psychedelic",
"Rave",
"Showtunes",
"Trailer",
"Lo-Fi",
"Tribal",
"Acid Punk",
"Acid Jazz",
"Polka",
"Retro",
"Musical",
"Rock & Roll",
"Hard Rock",
"Folk",
                                "Folk-Rock",
                                "National-Folk",
                                "Swing",
                                "Fast Fusion",
                                "Bebob",
                                "Latin",
                                "Revival",
                                "Celtic",
                                "Bluegrass",
                                "Avantegarde",
                                "Gothic Rock",
                                "Progressive Rock",
                                "Psychedelic Rock",
                                "Symphonic Rock",
                                "Slow Rock",
                                "Big Band",
                                "Chorus",
                                "Easy Listening",
                                "Acoustic",
                                "Humour",
                                "Speech",
                                "Chanson",
                                "Opera",
                                "Chamber Music",
                                "Sonata",
                                "Symphony",
                                "Booty Brass",
                                "Primus",
                                "Porn Groove",
                                "Satire",
                                "Slow Jam",
                                "Club",
                                "Tango",
                                "Samba",
                                "Folklore",
                                "Ballad",
                                "Power Ballad",
                                "Rhythmic Soul",
                                "Freestyle",
                                "Duet",
                                "Punk Rock",
                                "Drum Solo",
                                "A cappella",
                                "Euro-House",
                                "Dance Hall"
                             };
      assert genres.length == 126;
      String genre = (buffer[127] < 0 || buffer[127] > 125)
                     ? "Unknown" : genres[buffer[127]];
      return new ID3(songTitle, artist, album, year, comment, track, genre);
   }
   catch (IOException ioe)
   {
      if (raf != null)
         try
         {
            raf.close();
            }
            catch (IOException ioe2)
            {
               ioe2.printStackTrace();
            }
         throw ioe;
      }
   }
}
```

64. Listing 44 presents the `Split` application that was called for in Chapter 10.

Listing 44. *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 = fis.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);
            bos.close();
            bos = null;
         }
      }
      catch (IOException ioe)
      {
         ioe.printStackTrace();
         if (bis != null)
            try
            {
               bis.close();
            }
            catch (IOException ioe2)
            {
               ioe2.printStackTrace();
            }
         if (bos != null)
            try
            {
               bos.close();
            }
            catch (IOException ioe2)
            {
               ioe2.printStackTrace();
            }
      }
   }
}
```

65. Listing 45 presents the `CircleInfo` application that was called for in Chapter 10.

Listing 45. *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);
            }
         }
         catch (NumberFormatException nfe)
         {
            nfe.printStackTrace();
         }
      }
   }
}
```