C212 Final Exam

JOSHUA ISAACSON JSISAACS@IU.EDU

1. **Define: list, collection, set, maps.**
   - **List:** A *list* is a collection that remembers the order of its elements.
   - **Collection:** A *collection* groups together elements and allows them to be retrieved later.
   - **Set:** A *set* is an unordered collection of unique elements.
   - **Map:** A *map* keeps associations between key and value objects.
2. **Define: linked list.**
   - A *linked list* is a data structure used for collecting a sequence of objects that allows efficient addition and removal of elements in the middle of the sequence. It consists of a number of nodes, each of which has a reference to the next node.
3. **Define: list iterator.**
   - An iterator encapsulates a position anywhere inside the linked list. You can equate it to a cursor in a word processor which rests between two characters. You use a list iterator to access elements inside a linked list.
4. **Do linked lists take more storage space than arrays of the same size?**
   - Linked lists use more storage space because they store both the node as well as a reference to the next node, while an array just stores the value in sequential order with no reference to the elements next to it.
5. **Why don't we need iterators with arrays?**
   - You don't need an iterator with an array because when you initialize it, you declare its size, which allows you to access any position individually without having to iterate over the rest of the array to get to it.
6. **Consider the type HashSet and TreeSet. Do they have anything in common?**
   - The *HashSet* and *TreeSet* classes both implement the *Set* interface. Their implementations are based on hash tables and binary search trees, respectively. For the *HashSet*, set elements are grouped into smaller collections of elements that share similar hash codes. For the *TreeSet*, elements are kept in sorted order, stored in nodes arranged in a tree

shape. To use the *TreeSet*, you have to implement the Comparable interface, to show how the elements can be compared in terms of size.

7. **Write a loop that removes all strings with length less than four from a linked list of strings called words.**

```
while ( iterator.hasNext( ) ) {
    String s = iterator.next( ) ;
    if ( s.length( ) < 4 ) {
        iterator.remove( );
    }
}
```

8. **Write a loop that prints every second element of a linked list of strings called words.**

```
while ( iterator.hasNext( ) ) {
    String s = iter.next( );
    if ( ( words.indexOf( s ) % 2 ) == 1 ) {
        System.out.println( s );
    }
}
```

9. **Can you add an element to a set at an iterator position? Explain why or why not.**
   - You cannot because the *HashSet* and *TreeSet* take the elements and sort them in a non-linear way, so the their positions don't mean anything.
10. **Arrays and lists remember the order in which you added elements, sets do not. Why would you want to use a set instead of an array or list?**
    - You would use a set when you need to optimize for efficiency. Inserting and removing elements is more efficient with a set than with a list.
11. **Why are set iterators different from list iterators?**
    - A set iterator visits the elements in the order in which the set implementation keeps them.
12. **Write a loop that prints all elements that are in both Set<String> s and Set<String> t.**

```
HashSet< String > commonElements = new HashSet<>( t );
commonElements.retainAll( s );
```

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

13. **How do you find all keys and values in a map?**

```
for ( String key : map.keySet( ) ) {
    Integer value = map.get( key );
}
```

14. **Consider the types HashMap and TreeMap. What do they have in common?**
    - Both implementations sort the reference to the map object in a Map reference, use the *put* method to add an association, change the value of the existing association with the *put* method as well, *get* method returns the value associated with the key, and delete associations with the *remove* method.
15. **What is the difference between a set and a map?**
    - A *map* allows you to associate elements from a *key set* with elements from a *value collection*. A set doesn't allow duplicate elements, while a map lets you have duplicate values for unique keys.
16. **Why is the collection of the keys of a map a set and not a list?**
    - It is a map because the only pair relationship that matters for this data structure is the key-value pair. The set is more efficient than the list as long as the positions are important.
17. **Why is the collection of the values of a map not a set?**
    - Multiple keys can map to the same value.
18. **Suppose you want to track how many times each word occurs in a document. Declare a suitable map variable.**
    - Map< String, Integer > frequency = new Map<>( );
19. **What is a Map<String, HashSet<String>>? Give a possible use for such a structure.**
    - You would use this structure if you want keys to be Strings and values to be a Set of Strings. One example of this is if you want to have a user's username as the String and a Set of Strings containing other user info.
20. **What is a hash function? What is a *good* hash function?**
    - A *hash function* is a function that computes an integer value, the *hash code* from an object in such a way that different objectors are likely to

yield different hash codes. A good hash function produces different hash values for each object so that they are scattered about in a hash table.

21. **Define: stack, queue, priority queue.**
    - **Stack:** A *stack* is a collection of elements with "last-in, first-out" retrieval. The structure lets you insert and remove elements only at one end, traditionally called the top of the stack.
    - **Queue:** A *queue* is a collection of elements with "first-in, first-out" retrieval. It lets you add items to one end of the queue and remove them from the other end of the queue.
    - **Priority Queue:** A *priority queue* collects elements, each of which has a priority. When removing an element from a priority queue, the element with the most urgent priority is retrieved.
22. **Why would you want to declare a variable as Queue< String > q = new LinkedList<>( ) instead of simply declaring it as a linked list?**
    - Because the Queue interface has *add*, *remove*, and *peek*. It also brings performance benefits.
23. **Why wouldn't you want to use an array list for implementing a queue?**
    - Because the remove operation would have lackluster performance.
24. **Why wouldn't you want to use a stack to manage print jobs?**
    - Because whatever job is last to initiate would be the one first run. The queue is a better choice because whatever is first in line is run first.
25. **What does this code print?**

```
Queue<String> q = new LinkedList<>();
q.add("A");
q.add("B");
q.add("C");
while (q.size() > 0) { System.out.print (q.remove() + " " ); }
```

    A
    B
    C
26. **What is the value of the reverse Polish notation expression: 2 3 4 + 5 * *?**
    - 70.

27. **What steps does the selection sort algorithm go through to sort the sequence 6 5 4 3 2 1?**
    - Finds the minimum value which is 1 and swaps it with 6, 2 and 5, and 3 and 4 are swapped.
28. **Define: selection sort algorithm.**
    - The *selection sort* algorithm sorts an array by repeatedly finding the smallest element of the unsorted tail region and moving it to the front.
29. **Define: big-O notation.**
    - Big-O notation describes the growth rate of a function. It is used to analyze the performance of algorithms.
30. **Selection sort is an O(n^2) algorithm. What does this mean?**
    - It means that doubling the dataset means a fourfold increase in processing time.
31. **How large does *n* need to be so that $n^2/2$ is bigger than 5/2 * *n* - 3?**

***

32. **Define: merge sort algorithm.**
    - The merge sort algorithm sorts an array by cutting the array in half, recursively sorting each half, and then merging the sorted halves.
33. **Manually run the merge sort algorithm on the array 8 7 6 5 4 3 2 2.**
    - 8765 and 4322 are split. 2 is taken from the left side, then 2, then 3, then 4, then 5, then 6, then 7, then 8.
34. **Define: binary search.**
    - Binary search locates a value in a sorted array by determining whether the value occurs in the first or second half, then repeating the search in one of the halves. It locates a value in a sorted array in O (log (n)) steps.
35. **Suppose you need to look through 1,000,000 records to find a telephone number. How many records do you expect to search before finding the number?**
    - Using a binary search algorithm will make 500,000 comparisons.
36. **Suppose you need to look through a sorted array of 1,000,000 records to find a telephone number. Using the binary search algorithm how many records do you expect to search before finding the number?**
    - It would search about 20 because the binary log of 1024 is 10.
37. **Why can't the *Arrays.sort* method sort an array of *Rectangle* objects?**

- It is only designed to sort arrays of integers and floats.

38. **When designing a program how do you decide what classes you will need in your program?**
    - To discover what classes you need, look for nouns in the problem description.
39. **Define: inheritance, aggregation (also called composition).**
    - **Inheritance:** Inheritance is a relationship between a more general class (superclass) and a more specialized class (subclass). This is often described as the "is-a" relationship. Every track *is a* vehicle. Every savings account *is a* bank account.
    - **Aggregation:** The aggregation relationship states that objects of one class contain objects of another class. Consider a quiz which *has a* series of questions.
40. **Why should coupling be minimized between classes?**
    - It is a good practice to minimize coupling (i.e., dependency) between classes. This is because any changes to a coupled class will affect all of its dependencies.
41. **You are implementing a system to manage a library, keeping track of which books are checked out by whom. Should the *Book* class aggregate *Patron* or the other way around?**
    - It should be the other way around because it would be *Patron* "has-a" *Book*.
42. **What is *javadoc*? Where should you use its comments?**
    - *Javadoc* comments are used to record the behavior of classes.
43. **What is UML?**
    - UML diagrams are used to record class relationships.
44. **Define: CRC Cards.**
    - CRC cards are used to find classes, responsibilities, and collaborators.
45. **Why do the *format* methods return *String* objects instead of directly printing to *System.out*?**
    - The *format* methods return String because if allows you to use the format method outputs as inputs for other possible methods. This is good if you want to use a class as a dependency for another class that might have the main method inside of it.

46. Define: interface type.

- An *interface type* is used to specify required operations. It declares methods that can be applied to the variable of that type. The declaration is similar to the declaration of a class.

47. Can you convert from an interface type to a class type?

- Yes. If you are completely sure that the method in question refers to the equivalent class object, you can use cast notation to convert its type back (Object). If the object doesn't actually refer to the class in question, then a run-time exception will occur.

48. What is the design purpose of the *Comparable* interface?

- The *Comparable* interface is used so that objects of your class can be compared, for example, in a sort method.

49. Can you compare floating point numbers by subtraction? How about integers?

- When you implement a comparison method, you need to return a negative integer to indicate that the first object should come before the other, zero if they are equal, or a positive integer otherwise. You have seen how to implement this decision with 3 branches. You can use subtraction only when comparing *non-negative* integers. Use *Integer.compare* if you have negative integers. You *cannot* compare floating-point values by subtraction, instead use the *Double.compare method.*

50. Define: callback(s).

- A *callback* is a mechanism for bundling up a block of code so that it can invoked at a later time.

```
public static double average ( Object[] objects, Measurer measurer ) {
    double sum = 0;
    for ( Object obj : objects ) {
        sum = sum + measurer.measure( obj );
    }
    if ( objects.length > 0 ) { return sum / objects.length; }
    else { return 0; }
}
```

In this case, the *average* method makes a callback to the *measure* method whenever it needs to measure any object. Here is what the *measure* method looks like to measure the area of a Rectangle object.

```
public class AreaMeasurer implements Measurer {
   public double measure ( Object obj ) {
      Rectangle rect = ( Rectangle ) obj;
      double area = rect.getWidth( ) * rect.getHeight( );
      return area;
   }
}
```

51. Define: generic interface types.

- A *generic type* is a generic class or interface that is parameterized over types. Here is an example with a Box class.

```
public class Box {
   private Object object;
   public void set ( Object object) { this.object = object; }
   public Object get ( ) { return object; }
}
```

The generic version:

```
public class Box<T> {
   //T stands for "Type"
   private T t;
   public void set ( T t ) { this.t = t; }
   public T get ( ) { return t; }
}
```

52. Define: inner class(es). Why would you use an inner class instead of a regular one?

- An *inner class* is declared inside another class. They are commonly used for utility classes that should not be visible elsewhere in the program. This kind of class inside a method is not publicly accessible. As a note, the JVM turns an inner class into a regular class file.

53. **Define: anonymous class(es). Why would you use an anonymous class instead of a regular one?**
    - An *anonymous class* is an entity without a name. You would use this kind of class if it is only used once because you won't have to go through the trouble of coming up with a name and writing extra unnecessary lines of code. As of Java 8, it is better to just use a lambda expression.
54. **Define: event listeners.**
    - An *event listener* belong to a class that is provided by the application programmer. Its methods describe the actions to be taken when an event occurs. Event sources report on events. When an event occurs, the event source notifies all event listeners.
55. **Describe a common, basic use for *JPanel* objects.**
    - Use a a *JPanel* container to group multiple user-interface components together.
56. **Describe a common, basic use of the *ActionListener* interface.**
    - Use an *ActionListener* to respond to a JButton press.
57. **Define: timer, timer events.**
    - **Timer:** A timer generates timer events at fixed intervals.
    - **Timer events:** A timer event calls an action event after a specified timer interval.
58. **Describe the meaning and use of a *JComponent's repaint* method.**
    - The *repaint* method causes a component to repaint itself. Call *repaint* whenever you modify the shapes that the *paintComponent* method draws.
59. **Why does a timer require a listener object?**
    - The timer needs to call some method whenever the time interface expires. It calls the action performed method of the listener object.
60. **Define: mouse events.**
    - A *mouse event* indicates that a mouse action occurred in a component.
61. **Define: keyboard events.**
    - *Keyboard events* are instances of key presses, dictated by the key listener.
62. **Explain: the *Character* class has methods for classifying characters.**
    - It has the following methods: isDigit, isLetter, isUpperCase, isLowerCase, isWhiteSpace. When you read a character, or when you analyze the

characters in a word or line, you often want to know what kind of character it is. These methods all take in a *char* and return a *boolean* value.

63. **How do you convert *Strings* to numbers in Java?**
    - If a string contains the digits of a number, you use the *Integer.parseInt* or *Double.parseDouble* method to obtain the number value.
64. **Suppose the input contains the characters *Hello, World!* What are the values of *world* and *input* after this code fragment?**

    String word = in.next( );
    String input = in.nextLine( );
    Hello,
    World!
65. **Your input file contains a sequence of numbers, but sometimes a value is not available and is marked as N/A. How can you read the numbers and skip over the markers?**

    scanner.skip(Pattern.compile ("N/A"));
66. **Define: command line arguments.**
    - *Command line arguments* are method arguments that are given at the command line when compiling and running a java file.
67. **Encrypt CAESAR using the Caesar cipher.**
    - C → F, A → D, E → H, S → V, A → D, R → U.
    - Output: FDHVDU
68. **When do you use the *throw* statement.**
    - You use the *throw* statement to throw an exception object when you detect an error condition. *Exception handling* provides. Flexible mechanism for passing control from the point of error detection to a handler that can deal with the error.
69. **What is the *catch* clause used for?**
    - It contains the handler for an exception type. In your *try* statement, you give all possible situations that could return particular errors, and the catch clause actually returns the specific error associated with the action.
70. **What are checked exceptions?**

- *Checked exceptions* are due to external circumstances that the programmer cannot prevent. The compiler checks that your program handles these exceptions.

71. **When do you use the *throws* clause?**
    - The *throws* clause signals to the caller of your method that it may encounter a *FileNotFoundException*. Then the caller needs to make the same decision - handle the exception or declare that the exception may be thrown. It is similar to a truck carrying hazardous loads to display warning signs, the clause warns the caller that an exception may occur.
72. **How (and why) do you design your own *Exception* types?**
    - You would design your own types if none of the standard exception types describe your particular error condition well enough. To design your own, create a class that extends *Exception* and choose to create a checked or unchecked exception. Checked exceptions are supposed to flag problematic situations and unchecked exceptions represent a bug when compiling.
73. **Why is an *ArrayIndexOutOfBoundsException* not a checked exception?**
    - Because the technical issue is that the array index values are not valid, which ends up being a logic issue, not a syntax one.
74. **Is there a difference between catching checked and unchecked exceptions?**
    - No, you can catch both exception types in the same way.
75. **What does it mean: throw early, catch late?**
    - Throw an exception as soon as a problem is detected, catch it only when the problem can be handled.
76. **What are all permute ions of the word *beat*?**
    - They are -b- followed by 6 permutations of e-a-t. There are -e- followed by 6 permutations of b-a-t. There are -a- followed by 6 permutations of b-e-t. There are -t- followed by 6 permutations of -b-e-t.
77. **What is: mutual recursion?**
    - A set of cooperating methods call each other repeatedly.
78. **What is: backtracking?**

- *Backtracking* examines partial solutions, abandoning unsuitable ones and returning to consider other candidates.

79. **How many solutions are there altogether for the 4 queens problem?**
    - 2.
80. **How can you accidentally replicate instance variables from the superclass?**
    - A subclass has no access to the private instance variables of the superclass.
81. **Where does the sub/super terminology come from?**
    - It comes from set theory.
82. **What is: Accidental Overloading?**
    - The compiler will not complain. It thinks that you want to provide a method just for PrintStream arguments, while inheriting another method.
83. **Define: Constructor Chaining.**
    - *Constructor chaining* is the process of calling one constructor from another constructor with respect to the current object.
84. **What is polymorphism?**
    - *Polymorphism* allows us to manipulate objects that share a set of tasks, even though the tasks are executed in different ways.
85. **Define: Dynamic method lookup.**
    - When the virtual machine calls an instance method, it locates the method of the implicit parameter's class.
86. **Is the method call Math.sqrt(2) resolved through dynamic method lookup?**
    - No. This is a static method of the Math class. There is no implicit parameter object that could be used to dynamically look up a method.
87. **Define: abstract classes.**
    - A class for which you cannot create object is called an abstract class. In Java, you much declare all abstract class (es) with the reserved word abstract.
88. **Define: final method and classes.**
    - To prevent other programmers from creating subclasses or from overriding certain method. In these situations, you can use the final reserved word. That means that nobody can extend the class.

89. **Define: protected access.**
    - *Protected* is a version of *public* restricted only to subclasses.
90. **Why don't we simply store all objects in variables of type Object?**
    - There are only a few methods that can invoke on variables of type Object.
91. **Why does the call System.out.println(System.out); produce a result such as java.io.PrintStream@7a84e4?**
    - Because the implementor of the PrintSteam class did not supply a toString method.
92. **Will the following code fragment compile? Will it run? If not, what is error is reported?**

    ```
    Object obj = "Hello";
    System.out.println( obj.length( ) );
    ```

    - The second line will not compile. The class Object does not have a method length.
93. **Will the following code fragment compile? Will it run? If not, what error is reported?**
    - The code will compile, but the second line will throw a class cast exception because Question is not a superclass of String.
94. **Assuming that x is an object reference, what is the value of x instanceOf Object?**
    - The value is false if x is null and true otherwise.
95. **When and why do you use the reserved word *super*?**
    - Use the reserved word *super* to call a superclass method. It just forces execution of the superclass method.
96. **When and why do you use *super* with parentheses?**
    - It indicates a call to the superclass constructor. When used in this way the constructor call must be the first statement of the subclass constructor.
97. **Assuming *SavingAccount* is a subclass of *BankAccount,* which of the following code fragments are valid in Java?**

    A. BankAccount account = new SavingsAccount( );

    B. SavingsAccount account2 = new BankAccount( );

    C. BankAccount account = null;

    D. SavingsAccount account2 = account;

    - A. is valid.

98. **What is the purpose of the method *toString* in class *java.lang.Object*?**
    - Override the *toString* method to yield a string that describe the object's state.
99. **What is the purpose of the method *equals* in class *java.lang.Object*?**
    - The *equals* method checks whether the 2 objects have the same contents.
100. **How do you check that 2 objects belong to the same class?**
    - Boolean result = object1.getClass( ).equals(object2.get( ));
101. **What is the simple rule of thumb for finding classes?**
    - Look for nouns in the problem description.
102. **What is the criterion of cohesion for a class's public interface?**
    - The public interface of a class is cohesive if all of its features are related to the concept that the class represents.
103. **What is an immutable class?**
    - An *immutable class* has no mutator methods. References to objects of an immutable class can be safely shared.
104. **Why is it a good idea to minimize dependencies between classes?**
    - If a class doesn't have dependencies, it won't be affected by interface changes in the other classes.
105. **Is the *substring* method of the String class an accessor or a mutator?**
    - It is an accessor - calling substring does not modify the string on which the method is invoked. In fact, all methods of the String class are accessors.
106. **Is the *Rectangle* class immutable?**
    - No - translate is a mutator method.
107. **If *a* refers to a bank account, then the call *a.deposit(100)* modifies the bank account object. Is that a side effect?**
    - It is a side effect; this kind of side effect is common in object-oriented programming.
108. **What is the difference between a *static* variable and an instance variable?**
    - A static variable belongs to the class, not to any object of the class. An instance variable are used by Objects to stored their states.
109. **Name 2 *static* variables of the *System* class.**
    - *System.in* and *System.out* are 2 static variables.

110. **Name a *static* constant of the *Math* class.**
    - Math.Pi is a static constant of the Math class.
111. **Name a *static* method of the *Math* class.**
    - abs(double a) method returns the absolute value of a double value a.
112. **The following method computes the average of an array of numbers:**
     public static double average(double[ ] values);
     Why should it *not* be defined as an instance method?
    - The method needs no data of any object. The only required input is the values argument.
113. **What could go wrong if you try to access instance variables in static methods directly?**
    - A static method does operate on an object. It has no implicit parameter and you can't directly access any instance variables.
114. **What is a package?**
    - It is a set of related classes.
115. **What does *import* do?**
    - The import directive lets you refer to a class of a package by its class name, without the package prefix.
116. **What of the following are packages: java, java.lang, java.util, java.lang.Math?**
    - 1 and 4 are not, 2 and 3 are.
117. **What is a unit test framework?**
    - *Unite test frameworks* simplify the task of writing classes that contain many test cases.
118. **What is the JUnit philosophy?**
    - It is to run all tests whenever you change your code.
119. **Declare an array contains two strings, "Yes" and "No".**
    - String[ ] words = { "Yes", "No" };
120. **Declare an array called *words* that can hold 10 String elements.**
    - String[ ] words = new String[10];
121. **What is a standard rule of design for parallel arrays?**
    - Avoid parallel arrays by changing them into arrays of objects.
122. **In Java can you write methods that receive a variable number of arguments?**

- It is possible to declare methods that receive a variable number of arguments. For example we can write a method that can add an arbitrary number of scores to a student.

123. Why is the enhanced *for* loop not an appropriate shortcut for all basic *for* loops?

- The loop writes a value into values[I]. The enhanced for loop does not have the index variable i.

124. What is a linear search?

- A linear search inspects elements in sequence until a match is found.

125. What does *java.util.Array*'s *copyOf* method do?

- It is for copying the elements of an array into a new array.

126. Write a loop that counts how many elements in an array are equal to zero.

```
int count = 0;
for ( double x : values ) {
   if ( x == 0 ) {
      count++;
   }
}
```

127. Why don't we initialize *largest* and *I* with 0 when trying to find the largest element in an array?

```
double largest = 0;
for ( int I = 0; I < values.length; i++) {
   if ( values[ I ] > largest ) {
      largest = values[ I ];
   }
}
```

If all the elements of value are negative, then the result is incorrectly computed as 0.

128. Can you use *sort* method to sort a partially filled array?

- Yes. If the array is partially filled, call Array.sort(values, 0, currentSize);

129. Why should one be familiar with fundamental algorithms?

- It helps you solve problems.

130. **How can you print all positive values in an array separated by commas?**

```
boolean first = true;
for ( int I = 0; I < values.length; i++) {
    if ( values[ I ] > 0 ) {
        if ( first ) { first = false; }
        else { System.out.print( ", "); }
    }
    System.out.println( values[ I ] );
}
```

131. **Consider an 8 x 8 array for a board game:**

- Int[ ][ ] board = new int[8][8];
- Using 2 nested loops, initialize the board so that zeroes and ones alternate.

```
for ( int I = 0; I < 8; i++ ) {
    for ( int j = 0; j < 8; j++ ) {
        board[ I ][ j ] = ( I + j ) % 2;
    }
}
```

132. **Can you have 2D arrays with variable row lengths in Java? If no, explain why. If yes, give an example.**

- Yes. Here is an example of a triangularly-shaped 2D array.

```
b[ 0 ][ 0 ]
b[ 1 ][ 0 ] b[ 1 ][ 1 ]
b[ 2 ][ 0 ] b[ 2 ][ 1 ] b[ 2 ][ 2 ]
```

133. **What are multidimensional arrays?**

- You can declare arrays with more than 2 dimensions. For example here is a 3D array:

```
int[ ][ ][ ] cube = new int[ 3 ][ 3 ][ 3 ];
```

134. **What is wrong with this code snippet:**

```
ArrayList< String > names;
names.add( "Bob" );
```

- The *names* variable hasn't been initialized.

135. **What does the array list *names* contain after the following statements?**
    - It contains "Ann", and "Cal".
136. **Suppose you want to store the names of the weekdays. Should you use an array list or an array of seven strings and why?**
    - Because the number of weekdays does not change, there is no disadvantage to using an array vs. an ArrayList.
137. **What is regression testing?**
    - Regression testing involve repeating previously run tests to ensure that known failures of prior versions do not appear in new versions of the software.
138. **Suppose a customer of your program finds an error. What action should you take beyond fixing that error?**
    - Add a test case to the test suite that verifies that the error is fixed.
139. **How many numbers does this loop print?**

    for (int n = 10; n >= 0; n—) System.out.println(n);
    - It prints 11 numbers: 10 9 8 7 6 5 4 3 2 1 0.
140. **When should you use a *do* loop instead of a *while* or a *for* loop?**
    - The do loop is appropriate when the loop body must be executed at least once.
141. **Suppose Java didn't have a *do* loop. Could you rewrite any *do* loop as a *while* loop?**
    - Yes.
142. **Write a *do* loop that reads integers and computes their sum. Stop when reading the value 0.**

```
int x;
int sum = 0;
do {
   x = in.nextInt( );
   sum = sum + x;
}
while ( x != 0 );
```

143. **What is a sentinel value?**
    - A sentinel value denotes the end of a data set, but it is not part of the data.

144. **What is wrong with the following loop for reading a sequence of values?**

```
System.out.println("Enter values, Q to quit: ");
do {
  double value = in.nextDouble();
  sum = sum + value;
  count++;
} while (in.hasNextDouble());
```

- If the user doesn't provide any numeric input, the first call to in.nextDouble( ) will fail.

145. **What do the *break* and *continue* statements do? When do you use them?**

- **Break:** terminates the loop when reached.
- **Continue:** You don't have to place the rest of the loop code inside an else clause, but is rarely used.

146. **Describe common loop algorithms: sum and average.**

```
double total = 0;
int count = 0;
while ( in.hasNextDouble( ) ) {
   double input = in.nextDouble( );
   total = total + input;
   count++;
}
double average = 0;
if ( count > 0 ) {
   average = total / count;
}
```

147. **Describe common loop algorithms: counting matches.**

```
int spaces = 0;
for ( int I = 0; I < str.length( ); I++) {
   char ch = str.charAt( I );
   if ( ch == ' ' ) {
      spaces++;
```

```
    }
}
```

148. Describe common loop algorithms: finding the 1st match.

```
boolean found = false;
char ch = '?';
int position = 0;
while ( !found && position < str.length( ) ) {
    ch = str.charAt( position );
    if ( ch == ' ') { found = true; }
    else { position++; }
}
```

149. Describe common loop algorithms: prompting until a match is found.

```
boolean valid = false;
double input = 0;
while ( !valid ) {
    System.out.println( "Enter: " );
    input = in.nextDouble( );
    if ( 0 < input && input < 100 ) { valid = true; }
    else { System.out.println( "Invalid"); }
}
```

150. Describe common loop algorithms: maximum and minimum.

```
double largest = in.nextDouble( );
while ( in.hasNextDouble( ) ) {
    double input = in.nextDouble( );
    if ( input > largest ) {
        largest = input;
    }
}
```

151. Describe common loop algorithms: comparing adjacent values.

```
double input = in.nextDouble( );
while ( in.hasNextDouble( ) ) {
    double previous = input;
    input = in.nextDouble( );
    if ( input == previous ) {
```

```
            System.out.println( "duplicate." );
        }
    }
```

152. How do you find the position of the last space in a string?

```
boolean found = false;
int I = str.length( ) - 1;
while ( !found && I >= 0 ) {
    char ch = str.charAt( I );
    if ( ch == ' ' ) { found = true; }
    else { i—; }
}
```

153. How do you simulate a coin toss with the *Random* class?

- Compute generator.nextInt(2) and use 0 for heads, 1 for tails, or the other way around.

154. How do you simulate the picking of a random playing card?

- Compute generator.nextInt(4) and associate the numbers 0, 1, 2, 3 with the 4 suits. Then compute generator.nextInt(13) and associate the numbers 0, …, 12 with the cards.

155. How you generate a random floating-point number >= 0 and < 100?

- Generator.nextDouble( ) * 100.0;

156. Instead of using a debugger, could you simply trace a program by hand?

- For short solutions.