ANSWER KEY (Section I)

1. D
2. B
3. C
4. A
5. B
6. E
7. B
8. C
9. B
10. A
11. C
12. B
13. A
14. A
15. E
16. C
17. A
18. E
19. A
20. E
21. B
22. E
23. D
24. E
25. A
26. C
27. D
28. B
29. A
30. A
31. A
32. C
33. E
34. D
35. E
36. C
37. B
38. A
39. E
40. D

DIAGNOSTIC CHART FOR PRACTICE EXAM

Each multiple-choice question has a complete explanation (p. 47).

The following table relates each question to sections that you should review. For any given question, the topic(s) in the chart represent the concept(s) tested in the question. These topics are explained on the corresponding page(s) in the chart and should provide further insight into answering that question.

Question	Topic	Page
1	Inheritance	135
2	Implementing classes	212
3	Storage of integers	61
4	Constructors	95
5	The `toString` method	176
	`ClassCastException`	142
6	`Integer.MIN_VALUE` and `Integer.MAX_VALUE`	61
7	`for` loop	71
8	Program specification	208
9	Recursion	291
10	Boolean expressions	65
11	Hexadecimal	62
12	`IndexOutOfBoundsException` for `ArrayList`	244
13	Passing parameters	236
14	Passing parameters	236
15	Abstract classes	142
16	Subclass constructors and super keyword	135
17	Polymorphism	138
18	`swap` method	237
19	Rounding real numbers	61
20	Recursion	293
21	Selection and insertion sort	324
22	Subclass method calls	141
23	Compound boolean expressions	65
24	`String` class `equals` method	178
	`String` class `substring` method	180
25	Round-off error	62
26	Array processing	235
27	Assertions about algorithms	219
	Binary search	329
28	Binary search	329
29	Random integers	185
30	`String` class `substring` method	180
31	Two-dimensional arrays	249
32	Relationships between classes	216
33	Array of objects	239
	`ArrayList`	244
34	`NullPointerException`	103
35	Traversing an array	235
	The `if` statement	69
36	Processing a 2-D array	251
	Mirror images	357
37	Using `ArrayList`	245
38	Using `ArrayList`	245
39	Using super in a subclass	139
40	One-dimensional arrays	233

ANSWERS EXPLAINED

Section I

1. (D) Constructors are never inherited. If a subclass has no constructor, the default constructor for the superclass is generated. If the superclass does not have a default constructor, a compile-time error will occur.

2. (B) The programmer is using an object-oriented approach to writing the program and plans to test the simplest classes first. This is bottom-up development. In *top-down* development (choice A), high-level classes are broken down into subsidiary classes. Procedural abstraction (choice C) is the use of helper methods in a class. Information hiding (choice D) is restriction of access to private data and methods in a class. Choice E is wrong because a driver program is one whose sole purpose is to test a given method or class. Implementing the simplest classes first may involve driver programs that test the various methods, but the overall plan is not an example of a driver program.

3. (C) 8 bits (1 byte) are required to represent the values from 0 to 255. The base 2 number 11111111 represents $1+2+4+8+16+32+64+128=255$. Since there are 3 such values in an RGB representation, $(8)(3)=24$ bits are needed.

4. (A) In the constructor, the private instance variables `suit` and `value` must be initialized to the appropriate parameter values. Choice A is the only choice that does this.

5. (B) Implementation II invokes the `toString` method of the `Card` class. Implementation I fails because there is no default `toString` method for arrays. Implementation III will cause a `ClassCastException`: You cannot cast a `Card` to a `String`.

6. (E) Since the values in `arr` cannot be greater than `Integer.MAX_VALUE`, the test in the `while` loop will be true at least once and will lead to the smallest element being stored in `min`. (If *all* the elements of the array are `Integer.MAX_VALUE`, the code still works.) Similarly, initializing `min` to `arr[0]`, the first element in the array, ensures that all elements in `arr` will be examined and the smallest will be found. Choice I, `Integer.MIN_VALUE`, fails because the test in the loop will always be false! There is no array element that will be less than the smallest possible integer. The method will (incorrectly) return `Integer.MIN_VALUE`.

7. (B) The maximum number will be achieved if `/* test */` is true in each pass through the loop. So the question boils down to: How many times is the loop executed? Try one odd and one even value of `n`:

$$\text{If } n=7,\quad i=0,2,4,6 \qquad \text{Ans}=4$$
$$\text{If } n=8,\quad i=0,2,4,6 \qquad \text{Ans}=4$$

   Notice that choice B is the only expression that works for both `n` = 7 and `n` = 8.

8. (C) Here is one of the golden rules of programming: Don't start planning the program until every aspect of the specification is crystal clear. A programmer should never make unilateral decisions about ambiguities in a specification.

9. (B) When $x \le y$, a recursive call is made to `whatIsIt(x-1, y)`. If `x` decreases at every recursive call, there is no way to reach a successful base case. Thus, the method never terminates and eventually exhausts all available memory.

10. (A) The expression `!(max != a[i])` is equivalent to `max == a[i]`, so the given expression is equivalent to `a[i] == max || max == a[i]`, which is equivalent to `a[i] == max`.

11. (C) A base-`b` number can be represented with `b` characters. Thus, base-2 uses 0,1 for example, and base-10 uses $0,1,\ldots,8,9$. A hexadecimal (base-16) number is represented with 16 characters: $0,1,\ldots,8,9,A,B,C,D,E,F$, where $A=10, B=11,\ldots,F=15$. The largest two-place base-2 integer is

$$11 = 1\times 2^0 + 1\times 2^1 = 3$$

The largest two-place base-10 integer is

$$99 = 9\times 10^0 + 9\times 10^1$$

The largest two-place base-16 integer is

$$FF = F\times 16^0 + F\times 16^1$$

The character F represents 15, so

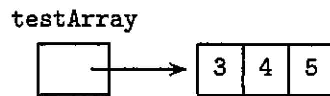
$$FF = 15\times 16^0 + 15\times 16^1 = 255$$

Here's another way to think about this problem: Each hex digit is 4 binary digits (bits), since $16 = 2^4$. Therefore a two-digit hex number is 8 bits. The largest base-10 number that can be represented with 8 bits is $2^8 - 1 = 255$.

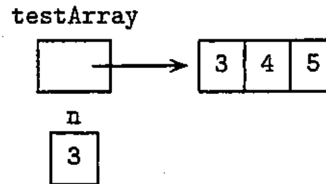
12. (B) The index range for `ArrayList` is $0 \le$ `index` $\le$ `size()-1`. Thus, for methods `get`, `remove`, and `set`, the last in-bounds index is `size()-1`. The one exception is the `add` method—to add an element to the end of the list takes an index parameter `list.size()`.

13. (A) The array will not be changed by the `increment` method. Here are the memory slots:

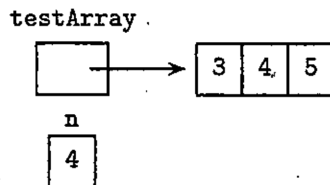
Before the first call, `increment(3)`:



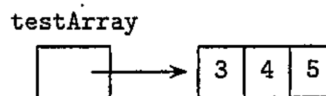
Just after the first call:



Just before exiting `increment(3)`:



Just after exiting `increment(3)`:



The same analysis applies to the method calls `increment(4)` and `increment(5)`.

14. (A) As in the previous question, the array will not be changed by the `increment` method. Nor will the local variable `element`! What *will* be changed by `increment` is the copy of the parameter during each pass through the loop.

15. (E) Subclasses of `Quadrilateral` may also be abstract, in which case they will inherit `perimeter` and/or `area` as abstract methods.

16. (C) Segment I starts correctly but fails to initialize the additional private variables of the `Rectangle` class. Segment II is wrong because by using `super` with `theTopLeft` and `theBotRight`, it implies that these values are used in the `Quadrilateral` superclass. This is false—there isn't even a constructor with three arguments in the superclass.

17. (A) During execution the appropriate `area` method for each `quad` in `quadList` will be determined (polymorphism or dynamic binding).

18. (E) The algorithm has three steps:

    1. Store the object at `i` in `temp`.
    2. Place at location `i` the object at `j`.
    3. Place `temp` at location `j`.

    This has the effect of swapping the objects at `i` and `j`. Notice that choices B and C, while incomplete, are not incorrect. The question, however, asks for the *best* description of the postcondition, which is found in choice E.

19. (A) Subtracting 0.5 from a negative real number and then truncating it produces the number correctly rounded to the nearest integer. Note that casting to an `int` truncates a real number. The expression in choice B is correct for rounding a *positive* real number. Choice C won't round correctly. For example, −3.7 will be rounded to −3 instead of −4. Choices D and E don't make sense. Why cast to `double` if you're rounding to the nearest integer?

20. (E) The method call `whatsIt(347)` puts on the stack `System.out.print(7)`.
    The method call `whatsIt(34)` puts on the stack `System.out.print(4)`.
    The method call `whatsIt(3)` is a base case and writes out `3`.
    Now the stack is popped from the top, and the `3` that was printed is followed by `4`, then `7`. The result is `347`.

21. (B) Recall that insertion sort takes each element in turn and (a) finds its insertion point and (b) moves elements to insert that element in its correct place. Thus, if the array is in reverse sorted order, the insertion point will always be at the front of the array, leading to the maximum number of comparisons and data moves—very inefficient. Therefore choices A, C, and E are false.

    Selection sort finds the smallest element in the array and swaps it with `a[0]` and then finds the smallest element in the rest of the array and swaps it with `a[1]`, and so on. Thus, the same number of comparisons and moves will occur, irrespective of the original arrangement of elements in the array. So choice B is true, and choice D is false.

22. (E) Method call I fails because `ClassOne` does not have access to the methods of its subclass. Method call II fails because `c2` needs to be cast to `ClassTwo` to be able to access `methodTwo`. Thus, the following would be OK:

```
((ClassTwo) c2).methodTwo();
```

    Method call III works because `ClassTwo` inherits `methodOne` from its superclass, `ClassOne`.

23. (D) Notice that in the original code, if `n` is 1, `k` is incremented by 1, and if `n` is 4, `k` is incremented by 4. This is equivalent to saying "if `n` is 1 or 4, `k` is incremented

by n."

24. (E) Segment I will throw a `NullPointerException` when `s.equals...` is invoked, because `s` is a null reference. Segment III looks suspect, but when the `startIndex` parameter of the `substring` method equals `s.length()`, the value returned is the empty string. If, however, `startIndex` > `s.length()`, a `StringIndexOutOfBoundsException` is thrown.

25. (A) Since results of calculations with floating-point numbers are not always represented exactly (round-off error), direct tests for equality are not reliable. Instead of the boolean expression `d == c`, a test should be done to check whether the difference of `d` and `c` is within some acceptable tolerance interval (see the Box on comparing floating-point numbers, p. 65).

26. (C) If `arr` has elements `2, 3, 5`, the values of `value` are

```
2                 //after initialization
2*10 + 3 = 23     //when i = 1
23*10 + 5 = 235   //when i = 2
```

27. (D) The point of the binary search algorithm is that the interval containing `key` is repeatedly narrowed down by splitting it in half. For each iteration of the `while` loop, if `key` is in the list, `arr[first]` ≤ `key` ≤ `arr[last]`. Note that (i) the endpoints of the interval must be included, and (ii) `key` is not necessarily in the list.

28. (B)

	first	last	mid	a[mid]
After first iteration	0	13	6	50
After second iteration	7	13	10	220
After third iteration	7	9	8	101
After fourth iteration	9	9	9	205

29. (A) The data structure is an array, not an `ArrayList`, so you cannot use the `add` method for inserting elements into the list. This eliminates choices B and D. The expression to return a random integer from `0` to `k-1` inclusive is

```
(int) (Math.random() * k)
```

Thus, to get integers from 0 to 100 requires `k` to be 101, which eliminates choice C. Choice E fails because it gets integers from 1 to 100.

30. (A) Suppose `str1` is `strawberry` and `str2` is `cat`. Then `insert(str1, str2, 5)` will return the following pieces, concatenated:

```
straw + cat + berry
```

Recall that `s.substring(k, m)` (a method of `String`) returns a substring of `s` starting at position `k` and ending at position `m-1`. String `str1` must be split into two parts, `first` and `last`. Then `str2` will be inserted between them. Since `str2` is inserted starting at position `5` (the `"b"`), `first = straw`, namely `str1.substring(0,pos)`. (Start at `0` and take all the characters up to and including location `pos-1`, namely `4`.) Notice that `last`, the second substring of `str1`, must start at the index for `"b"`, which is `pos`, the index at which `str2` was inserted. The expression `str1.substring(pos)` returns the substring of `str1` that starts at `pos` and continues to the end of the string, which was required. Note

that you don't need any "special case" tests. In the cases where `str2` is inserted at the front of `str1` (i.e., `pos` is `0`) or the back of `str1` (i.e., `pos` is `str1.length()`), the code for the general case works.

31. (A) Method `changeMatrix` examines each element and changes it to its absolute value if its row number equals its column number. The only two elements that satisfy the condition `r == c` are `mat[0][0]` and `mat[1][1]`. Thus, `-1` is changed to `1` and `-4` is changed to `4`, resulting in the matrix in choice A.

32. (C) Composition is the *has-a* relationship. A `PlayerGroup` *has-a* `Player` (several of them, in fact). Inheritance, (choice D) is the *is-a* relationship, which doesn't apply here. None of the choices A, B, or E apply in this example: An interface is a single class composed of only abstract methods (see p. 144); encapsulation is the bundling together of data fields and operations into a single unit, a class (see p. 93); and `PlayerGroup` and `Player` are clearly dependent on each other since `PlayerGroup` contains several `Player` objects (see p. 212).

33. (E) All of these data structures are reasonable. They all represent 20 bingo numbers in a convenient way and provide easy mechanisms for crossing off numbers and recognizing a winning card. Notice that data structure II provides a very quick way of searching for a number on the card. For example, if 48 is called, `bingoCard[48]` is inspected. If it is `true`, then it was one of the 20 original numbers on the card and gets crossed out. If `false`, 48 was not on that player's card. Data structures I and II require a linear search to find any given number that is called. (Note: There is no assumption that the array is sorted, which would allow a more efficient binary search.)

34. (D) A `NullPointerException` is thrown whenever an attempt is made to invoke a method with an object that hasn't been created with `new`. Choice A doesn't make sense: To test the `Caller` constructor requires a statement of the form

```
Caller c = new Caller();
```

Choice B is wrong: A missing `return` statement in a method triggers a compile-time error. Choice C doesn't make sense: In the declaration of `numbers`, its default initialization is to `null`. Choice E is bizarre. Hopefully you eliminated it immediately!

35. (E) For each element in `a`, `found` is switched to `true` if that element is found anywhere in `b`. Notice that for any element in `a`, if it is not found in `b`, the method returns `false`. Thus, to return `true`, every element in `a` must also be in `b`. Notice that this doesn't necessarily mean that `a` and `b` are permutations of each other. For example, consider the counterexample of `a=[1,1,2,3]` and `b=[1,2,2,3]`. Also, not every element in `b` needs to be in `a`. For example, if `a=[3,3,5]` and `b=[3,5,6]`, the method will return `true`.

36. (C) In the example given, `height` $=$ `3`, `height/2` $=$ `1`, and `numCols` $=$ `3`. Notice that in each pass through the loop, `row` has value `0`, while `col` goes from `0` through `2`. So here are the assignments:

```
mat[2][0] = mat[0][0]
mat[2][1] = mat[0][1]
mat[2][2] = mat[0][2]
```

From this you should see that row `2` is being replaced by row `0`.

37. (B) Eliminate choices D and E immediately, since assignment of new values in an ArrayList is done with the set method, not get. Eliminate choice C since you do not know that the TileBag class has a swap method. Choice A fails because it replaces the element at position size before storing it. Choice B works because the element at position index has been saved in temp.

38. (A) The size variable stores the number of unused tiles, which are in the tiles list from position 0 to position size. A random int is selected in this range, giving the index of the Tile that will be swapped to the end of the unused part of the tiles list. Note that the length of the tiles ArrayList stays constant. Each execution of getNewTile decreases the "unused tiles" part of the list and increases the "already used" part at the end of the list. In this way, both used and unused tiles are stored.

39. (E) When pigeon.act() is called, the act method of Dove is called. (This is an example of polymorphism.) The act method of Dove starts with super.act() which goes to the act method of Bird, the superclass. This prints fly, then calls makeNoise(). Using polymorphism, the makeNoise method in Dove is called, which starts with super.makeNoise(), which prints chirp. Completing the makeNoise method in Dove prints coo. Thus, so far we've printed fly chirp coo. But we haven't completed Dove's act method, which ends with printing out waddle! The rule of thumb is: When super is used, find the method in the superclass. But if that method calls a method that's been overridden in the subclass, go back there for the overridden method. You also mustn't forget to check that you've executed any pending lines of code in that superclass method!

40. (D) In Implementation 1, the first element assigned is prod[1], and it multiplies arr[1] by prod[0], which was initialized to 1. To fix this implementation, you need a statement preceding the loop, which correctly assigns prod[0]: prod[0]=arr[0];

Section II

1. (a)
```
public static void reverseArray(int[] arr)
{
    int mid = arr.length/2;
    for (int i = 0; i < mid; i++)
    {
        int temp = arr[i];
        arr[i] = arr[arr.length - i - 1];
        arr[arr.length - i - 1] = temp;
    }
}
```

(b)
```
public void reverseAllRows()
{
    for (int[] row: mat)
        ArrayUtil.reverseArray (row);
}
```

(c)
```
public void reverseMatrix()
{
    reverseAllRows();
    int mid = mat.length/2;
    for (int i = 0; i < mid; i++)
    {
        for (int col = 0; col < mat[0].length; col++)
        {
            int temp = mat[i][col];
            mat[i][col] = mat[mat.length - i - 1][col];
            mat[mat.length - i - 1][col] = temp;
        }
    }
}
```

Alternative solution:

```
public void reverseMatrix()
{
    reverseAllRows();
    int mid = mat.length/2;
    for (int i = 0; i < mid; i++)
    {
        int[] temp = mat[i];
        mat[i] = mat[mat.length - i - 1];
        mat[mat.length - i - 1] = temp;
    }
}
```

NOTE

- Parts (a) and the alternative solution in part (c) use the same algorithm, swapping the first and last elements, then the second and second last, etc., moving toward the middle. If there is an odd number of elements, the middle element does not move. In part (a) the elements are integers. In part (c) they are rows in the matrix.
- In the first solution of part (c), start by reversing all rows. Then for each column, swap the elements in the first and last rows, then the second and second last, and so on, moving toward the middle.
- The alternative solution in part (c) is more elegant. It is not, however, part of the AP subset to replace one row of a matrix with a different array.

2. (a)
```
public List<Integer> getBlankPositions()
{
    List<Integer> posList = new ArrayList<Integer>();
    for (int i = 0; i < sentence.length(); i++)
    {
        if (sentence.substring(i, i + 1).equals(" "))
            posList.add(i);
    }
    return posList;
}
```

Alternatively (an inferior, unnecessarily complicated solution!),

```
public List<Integer> getBlankPositions()
{
    List<Integer> posList = new ArrayList<Integer>();
    String s = sentence;
    int diff = 0;
    int index = s.indexOf(" ");
    while (index >= 0)
    {
        posList.add(index + diff);
        diff = sentence.length() - (s.substring(index + 1)).length()
        s = s.substring(index + 1);
        index = s.indexOf(" ");
    }
    return posList;
}
```

(b)
```
public int countWords()
{
    return getBlankPositions().size() + 1;
}
```

(c)
```
public String[] getWords()
{
    List<Integer> posList = getBlankPositions();
    int numWords = countWords();
    String[] wordArr = new String[numWords];
    for (int i = 0; i < numWords; i++)
    {
        if (i == 0)
        {
            if (posList.size() != 0)
                wordArr[i] = sentence.substring(0, posList.get(0));
            else
                wordArr[i] = sentence;
        }
        else if (i == posList.size())
            wordArr[i] = sentence.substring(posList.get(i - 1));
        else
            wordArr[i] = sentence.substring(posList.get(i - 1),
                           posList.get(i));
    }
    return wordArr;
}
```

NOTE

- In part (a), it would also work to have the test

```
i < sentence.length() - 1;
```

in the for loop. But you don't need the -1 because the last character is a punctuation mark, not a blank.
- In the alternative part (a), you can't just store the positions of index as you loop over the sentence. Finding s.indexOf(" ") will give a value that is too small, because you are successively taking shorter substrings of s. The local variable, diff, represents the difference between the length of the original sentence and the length of the current substring. This is what must be added to the current value of index, so that you get the position of the blank in the original sentence.
- Part (b) takes advantage of the precondition that there is one and only one blank between words. This means that the number of words will always be the number of blanks plus one.
- In part (c), you have to be careful when you get the first word. If there's only one word in the sentence, there are no blanks, which means posList is empty, and you can't use posList.get(0) (because that will throw an IndexOutOfBoundsException!).
- Also in part (c), the second test deals with getting the last word in the sentence. You have to distinguish between the cases of more than one word in the sentence and exactly one word in the sentence.

3. (a)

```
public Player requestSlot(String playerName)
{
    for (int i = 0; i < slots.length; i++)
    {
        if (slots[i] == null)
        {
            Player p = new Player(playerName, i);
            slots[i] = p;
            return p;
        }
    }
    waitingList.add(playerName);
    return null;
}
```

(b)

```
public Player cancelAndReassignSlot(Player p)
{
    int i = p.getPlayerNumber();
    if (waitingList.size() != 0)
    {
        slots[i] = new Player(waitingList.get(0), i);
        waitingList.remove(0);
    }
    else
    {
        slots[i] = null;
    }
    return slots[i];
}
```

NOTE

- In part (a), the last two lines of the method will be executed only if you are still in the method, namely no available slot was found.
- In part (b), the final line will return either a new player, or `null` if the waiting list was empty.

4. (a)
```
public void reset()
{
    if(arm.isFacingRight())
        arm.changeDirection();
    arm.moveForward(arm.getCurrentIndex());
    arm.changeDirection();
}
```

(b)
```
public int mostAcidic()
{
    reset();
    int minPH = Integer.MAX_VALUE, minIndex = 0;
    int index = 0;
    while (index < solutions.size())
    {
        Solution s = solutions.get(index);
        if (s.getPH() < minPH)
        {
            minPH = s.getPH();
            minIndex = index;
        }
        index++;
    }
    if (minPH >= 7)
        return -1;
    else
    {
        arm.moveForward(minIndex);
        return minIndex;
    }
}
```

NOTE

- In part (b), a for-each loop won't work, because you need to save an index.
- In part (b), notice that resetting the mechanical arm causes the arm to face right.
- In part (b), you could initialize `minPH` to any integer greater than or equal to 7 for this algorithm to work. You just must be careful not to set it to an "acidic" number, namely 1 to 6.