Student's Solutions Manual
To Accompany

Multiple-Choice & Free-Response Questions in Preparation for the AP Computer Science Examination

Seventh Edition

by

Leon Schram

John Paul II High School
Plano, Texas

ISBN # 978-1-934780-07-7

Printed in the U.S.A.

Preface

This manual is a detailed supplement for those students who desire to see the steps involved in working out the problems presented in the question manual. The solutions in this booklet go way beyond the presentation of a single letter answer key. For each multiple-choice question a detailed explanation is given to indicate why the given answer is the correct solution. Sometimes students understand a topic well but they are confused by the wording of the questions. The explanation here should help to clarify the logic of the questions and the reasons for the correct answers.

For each free-response question every completed method is part of a workable program environment, which enables teachers and students to view the solution in the context of a program that compiles and executes.

The biggest problem with creating a booklet of this type is to decide the degree of information that is provided with each question. It needs to be understood that the purpose of these solutions is not to teach any given topic. It is quite possible that certain fine points may be better understood by the explanations that are provided, but it not the intention of this manual. It is assumed that the teacher or student using these questions has familiarity with the computer science theory associated with each topic, including the commonly recognized vocabulary.

An effort is made to indicate a natural progression through a problem to help understand the selected answer. In many situations, diagrams and "variable traces" are used to help explain the logic of the solution.

I like to hear from students and teachers about the questions and solutions that are presented. Please let me know if you believe a solution to be in error or if some other answer can be shown to be equally valid to the selected answer.

Leon Schram

All communication concerning this book should be addressed to the publisher and distributor:

D & S Marketing Systems, Inc.
1205 38th Street
Brooklyn, New York 11218

www.dsmarketing.com

Solutions Table of Contents

01. Control Structures 1
02. Methods and Parameters 4
03. Boolean Algebra 8
04. Object Oriented Programming 11
05. Static Java 1D Arrays 14
06. Static Java 2D Arrays 18
07. The String Class 22
08. Number Systems 25
09. The ArrayList Class. 28
10. Recursion. 31
11. Implementing Methods and Redefining Methods 36
12. Java Standard Libraries 39
13. Algorithms and Informal Algorithmic Analysis 42
14. Polymorphism 45
15. Program Design. 48
16. Sample Examination I 51
17. Sample Examination II 62
18. Sample Examination III 74

Chapter 1 Solutions
Control Structures

Many of the control structure solutions in this chapter, and in future chapters, will be shown with *trace tables* that display the changes in variable values for each iteration of the loop structure. Sometimes it is possible to recognize a pattern in the value changes, which makes it unnecessary to check every step of the actual program execution. Keep in mind that some questions only ask for the *last output* of the execution. The intermediate value will still be shown in the trace tables to help establish the pattern that will display the last output.

01. (A)

p	2	3	5	8	**12**
q	1	2	3	4	**5**

02. (E)

p	0	1	2	3	4	
count	4	3	5	4	6	**6**

03. (D)

It may be tempting to conclude that it is not possible to determine what the code segment prints without knowing the value of **x**. Actually, the value of **x** is not necessary at all.

If it is **true** that **x < 1000** then it is **false** that **x > 1500**, in which case **n = 300**.

If it is **false** that **x < 1000** then **x** must be greater or equal to **1000**. That means that it is **false** that **x < 500** and once again **n = 300**.

No matter what value of **x** is used, the result is that **n** will always become **300**.

04. (D)

The loop condition is While **(x < y)** and the program output displays the value of **x** after the loop exits. The actual program statements inside the loop body have no impact. It is possible that the program statements are such that the loop condition never becomes true, and this means that the loop will never exit. None of the answers handles such a scenario. This leaves the exit condition, which occurs when **x >= y**.

05. (A)

For x values in the range [1..9] the two loops will not be identical. The **while** loop condition is **(x <= n)** and the **for** loop condition is **(int p = y; p < n; p++)**. This means that the **while** loop will repeat one more time than the **for** loop.

The moment **x >= 10** it is also true that then **x > n**. This means that neither loop is entered and the output for both loops is an identical blank screen. This is not the result of a program crash. Both segments execute properly, but without entering the loop body, execution stops without any output display.

06. (C)

Before the while loop body is entered the relationship between **p** and **q** can be expressed with the equation **q = q + p**. This odd loop equation is correct, because the initial value of **p** is **0**. Inside the loop, **p** increases by **2** each time and **q** decreases by **2** each time. This means that throughout the loop, and after the loop, the expression **q + p** will equals the original value of **q.**

07. (B)

The program states that *some integer value greater than q* is assigned to **p**.
It is now impossible that the loop condition, **(p < q)** can ever be **true**.
This means that the loop body is never entered and the initial **p** and **q** values are displayed.

08. (A)

The code segment correctly prints the letter grade for each number range.
The nested **else..if** keeps every number range in control.

09. (E)

The very first condition **(average < 100)** applies to every number, except **100**.
This means that a score of **45** will display *your grade is A*.

10. (A)

This question is intentionally similar to question **05**. The only difference is the change from the loop condition **(x <= n)** to **(x < n)**. It is easy to overlook such subtle changes.
Now that both the **while** loop and the **for** loop have the **< n** condition, output display stops when the value of **x** equals **n** or **9**.

11. (D)

For **n** greater than **0**, **segment 2** repeats one more time than **segment 1** and they are not equal. When **n** is **0** or less, the loops in **Segment 1** and **Segment 2** are not entered. The initial values for **s1** and **s2** are identical and display the same output.

12. (E)

The outer loop repeats from **0** to **n-1**, which is **n** times.
The inner loop repeats from **1** to **n – 1**, which is **n – 1** times.
Variable **count** is incremented for each iteration of the inner loop and starts with **0;**
This means that **count** is called **n** times **n – 1** times.

13. (D)

The condition **n % 2 == 0** means n is divisible by 2 or n is a multiple of 2. In the same way **n % 3 == 0** means n is divisible by 3 or n is a multiple of 3. This program will only display the word *Hello* when both conditions are true. For a number to be a multiple of 2 and a multiple of 3, it must be a multiple of 2 times 3, or in other words, a multiple of 6.

14. (E)

If we work our way from the inside out, the compound condition **n > 50 && n < 100** refers to numbers in the range of 51..99. Now that compound condition is in a set a parentheses with an ! (meaning NOT) outside them. This negates everything inside the parentheses. So instead of dealing with integers in the range of 51..99, we are dealing with integers NOT in that range.

15. (A)

If an AND symbol (&&) were use, the answer would be **(E)**, but since an OR symbol (II) is used output is always generated because when you combine all of the integers greater than or equal to **100** with all of the integers less than or equal to **100**, you have ALL of the integers.

Chapter 2 Solutions
Methods and Parameters

01. (D)

n & p	7							
q		7	8	9	10	11	12	13
count	1	2	4	8	16	32	64	128

In the table above **6** is used as the value for **n**.
Each iteration through the loop doubles the value of **count**.
The repetition continues as long as **q < 2 * p**
The result is that **count** will return 2^n.

02. (A)

The number of repetitions executed by **method0202** is not an issue.
Variable **count** starts at **0** and for each iteration through the loop **0** is added to **0**.
This method will always return **0**, regardless of the parameter value.

03. (B)

n	1
p	0
q	0
temp	1

n	2		
p	0	0	1
q	0	1	0
temp	1	2	3

n	3					
p	0	0	0	1	1	2
q	0	1	2	0	1	0
temp	1	2	3	4	5	6

n	4									
p	0	0	0	0	1	1	1	2	2	3
q	0	1	2	3	0	1	2	0	1	0
temp	1	2	3	4	5	6	7	8	9	10

It is important to realize that the inner loop has a condition that is reduced by **p**.
The four variable traces shown here show that with each increase of **n** there is also a pattern increase that adds 2, then 3, then 4 and so on until **55** is returned.

04. (D)

n	9							
p	0	3	4	5	6	7	8	9
k1	3	2	1	3	4	7	11	18
k2	2	1	3	4	7	11	18	29
k3	1	3	4	7	11	18	29	47

05. (D)

method0205b is a method to compute the *GCF* of **p** and **q**.
The *GCF* of **120** and **108** is **12**, which is returned by **method0205b**.
The result of **12** is assigned to **temp** in **method0205a**.
method0205a computes the *LCM* of **x** and **y** using the *GCF*.
120 divided by the *GCF* **12** = **10** times **108** = the *LCM* of **1080**.

06. (B)

Many students will initially get the wrong answer to this question.
Variable **n** starts with **10** and becomes larger with every iteration through the loop.
The correct output of **10** seems unlikely.
However, the output display is the value of **x**, which never changed from **10**.
In Java, parameters pass a value, and the *actual* (calling) parameter does not change.

07. (A)

This question is similar to the previous question with one major difference.
method0207 is a *return* method and the previous **method0206** is a *void* method.
Consider the value of **n** as it changes in the method.

x	10										
n	10	11	13	16	20	25	31	38	46	55	65
k		1	2	3	4	5	6	7	8	9	10

This time the value **n = 65** is returned and assigned to **x**. In this manner **x** does change.

08. (B)

Values **x** and **y** are displayed first in the main method as **10 20**.
Method **swap** exchanges the parameters values of **p** and **q**.
This is followed with a display of **p** and **q**, which is **20 10**.
This question is intentionally simple to demonstrate the difference with the next question.

09. (A)

This question, like the previous one, starts with a display of **x** and **y**, which is **10 20**.
Method **swap** is called after the first output display.
The values of **p** and **q** are exchanged, as they were in the previous question.
This time there is no output display inside the **swap** method.
The second display of **x** and **y**, in the **main** method, displays the same **10 20** values.
Remember that a change in *formal* (receiving) parameters does not alter *actual* (calling) parameters.
Three more questions follow with the **swap** method to help illustrate this point.

10. (D)

In this question a **Widget** class is declared, which contains a **swap** method.
The values of two **Widget** objects are displayed, before and after the **swap** call.
The immediate value stored for an object is a memory reference.
This memory reference is displayed, unless a **toString** method is redefined.

11. (A)

Question 11 uses the same **Widget** class as Question 10.
This time the **getWidgets** method is used, which results in the **100 200** display.

12. (B)

Finally, the call to the **swap** method brings about some exchange results.
In this case method **swap** exchanges the *deep* values of the two **Widget** objects.
The values of **w1** and **w2** are left unchanged, but the values being referenced are now swapped.

13. (B)

This program shows 4 overloaded **boohiss** methods. One has an **int** parameter; one has a **double** parametes; one has a **char** parameter; and one has an **String** parameters. Since the parameter list or *signature* is different this is fine. The main method has one statement that calls **boohiss** with a parameter of **3.0**. This matches the **boohiss** method with the **double** parameter and *Hola* is displayed.

14. (B)

Like the previous program, this program also shows 4 overloaded **boohiss** methods. This time all of the methods have **int** parameters but the signatures are still different because the first has 1 **int** parameter, the second has 2, the third has 3 and the fourth has 4 **int** parameters. In the main method, we call the 2 parameter **boohiss** method first, following by the 1 parameter, 4 parameter, and 3 parameter **boohiss** methods.

15. (E)

This program will not compile because the second and fourth **boohiss** methods have the same signature. The parameter identifiers may be different in both methods, but that is irrelevant. Both methods have 2 **int** parameters. When a call to **boohiss(2,11);** is made, the computer has no way to determine whether to use the second or fourth **boohiss** method.

Chapter 3 Solutions
Boolean Algebra

01. (A)

The expression inside **(A || B)** is irrelevant. The **&& A** expression can only result in a **true** value if **A** is **true**.

02. (B)

The expression (**A && B) || B** is **true** when either left side **(A && B)** is true or right side **B** is **true**. If **B** is **false**, then expression **A && B** is **false** and the whole expression is **false**. This means that the value of **A** is irrelevant and the whole expression is **true** when **B** is **true**.

03. (D)

The logical **&&** operator used in any expression requires that every operand is **true** for the expression to be **true**. If multiple **&&** operators are used, then all operands and expressions must be **true**. In this case this means that both **A** and **B** must be **true** at the same time.

04. (D)

The logical **||** operator used in any expression requires that only one operand is **true** for the expression to be **true**. If multiple **||** operators are used, then it is still the case that at least one operand must be **true**.

I. is correct, because it is sufficient for one operand to be **true**.
II. is correct, because the *inclusive* or is still **true** when multiple operands are **true**.
III. is incorrect, because it describes the definition of an *exclusive* (**xor**) or.

05. (E)

The Boolean expression **!(A || B)** is equivalent to **!A && !B** by *DeMorgan's Law*.

06. (D)

The Boolean expression **!(A && B)** is equivalent to **!A || !B** by *DeMorgan's Law*.

07. (B)

The Boolean expression **(A || B) || !(A || B)** is equivalent to **Q || !Q**, which guarantees a **true || false** or **false || true** expression. The result is always **true**.

08. (C)

When *DeMorgan's Law* is applied to expression **!((A < B) && (C > D))** you get **!(A < B) || !(C > D)**. Expression **!(A < B)** is equivalent to **(A >= B)** and expression **!(C > D)** is equivalent to **(C <= D)**.

09. (B)

When *DeMorgan's Law* is applied to expression **!((A < B) || (C > D))** you get **!(A < B) && !(C > D)**. Expression **!(A < B)** is equivalent to **(A >= B)** and expression **!(C > D)** is equivalent to **(C <= D)**.

10. (A)

Short-Circuiting can occur when some condition has compound expressions. If the evaluation of the first expression guarantees the result of the entire condition, then any additional expressions are not evaluated. For instance, with expression **A && (B && (C || D && E) || F)** if it is known that **A** is **false**, then the entire expression will be **false**.

This question involves condition **(p != 0 && q!= 0)**. This means that the condition will *short-circuit* if expression **P != 0** is **false**. This situation occurs whenever **p** equals **0**.

11. (A)

The Boolean expression **(A && B) && (!A || !B)** can be changed to **(A && B) && !(A && B)** using *DeMorgan's Law*. The resulting expression can be simplified to **true && !true** or **false && !false,** which is always **false**.

12. (A)

The Boolean expression **(A || B) && !(A || B)** can be simplified to **true && !true** or **false && !false**, which is always **false**.

13. (B)

This is a classic example of *Short Circuiting*. Since **A** is false, the values of all of the other variables on the other side of the AND (&&) sign are irrelevant. On a computer, they would not even be looked at. AND requires both conditions to be true. If the first condition is false, then we already know the entire expression is false, so there is no need to check anything else.

14. (E)

The **for** loop counts from 1 to 49. Each time through the loop, the value of **boolean** variable **pineapple** becomes **true** if **x** is a multiple of 5, **false** otherwise. Even though the value of **pineapple** is true about 20% of the time and false about 80% of the time, the word **false** is never displayed. This is because the value of **pineapple** is only displayed if **pineapple** is **true**. Since there are 9 multiples of 5 between 1 and 49, 9 **true**s are displayed.

15. (E)

For the first value, we need to look at what happens before the first output occurs. **guava** is initialized to the value of **1**. **guava** doubles itself (so now it is 2) and then **guava** is displayed. **2** is the first output. This eliminates 3 of your choices and now you just have to choose between **(C)** and **(E)**. The question is what is the last number output? The loop repeats as long as **pineapple** is **true**. The condition for **pineapple** being **true** is that **guava** is less than **100**, so you might think the answer is **(C)**, but be careful. When **guava** is **64**, the condition is **true** so the loop repeats. **guava** doubles again (so now it is **128**) and the **128** is displayed. Now the condition becomes **false** and the loop stops.

Chapter 4 Solutions
Object Oriented Programming

01. (B)

The program instantiates a new **Widget** object using parameter **10** for the constructor.
The constructor uses the confusing statement **numWidgets = numWidgets**.
This is interpreted by Java as the local constructor method variable **numWidgets**.
It is not assumed to be the attribute **numWidgets**.
Attribute **numWidgets** does not get any parameter information and defaults to **0.**

02. (D)

This question demonstrates a fundamental syntax error.
Constructors are neither *void* methods *nor* return methods.
The **Widget** constructor is declared as a *void* method, which results in a syntax error.

03. (A)

Implementation (A) correctly assigns the value parameter of 99 to all 10 values.
Implementation (B) has an incorrect method heading and will not compile.
Implementation (C) fails to instantiates the **list** object with the **new** operator.
Implementation (D) completely mangles the instantiation of the **list** object.
Implementation (E) creates a proper object, exceeds the index bounds of the array.

04. (B)

This is the first of three questions, which investigate the relationship between two classes.
In this question there is no relationship between **Rumba** and **Mambo**.
Two different objects are constructed, and each constructor displays its message.

05. (E)

Question 05 shows that the **Mambo** class *has-a* **Rumba** class attribute.
This is the composition relationship between two classes.
In the **main** method a **Mambo** object is first constructed.
This results in the **Mambo** constructor message.
A new **Rumba** object is constructed inside the **Mambo** constructor and displays **Rumba**.
The **main** method constructs a **Rumba** object for a second **Rumba** message.

06. (C)

Question 06 shows that the **Mambo** class *is-a* **Rumba** class with the **extends** keyword.
This is the inheritance relationship between two classes.
The **main** method first instantiates a **Mambo** object.
Since **Mambo** is a sub class, it instantiates a super class **Rumba** object first, which is displayed.
Then it continues and finishes the **Mambo** object with a **Mambo** message display.
The output display concludes with a **Rumba** message from the **main** method construction.

07. (B)

There appears only one class, **Aardvark**, in this question.
However, every class extends the **Object** class.
The **Object** class contains the original definition of the **toString** method.
Method **toString** is redefined in **Aardvark**.

08. (C)

The **super** statement must be the first statement in any sub class constructor to pass information to the super class constructor.

09. (A)

The **toString** method of the super class **Person** gives the correct output for **age**.
The statement **super.toString()** starts with **age** and then adds **gpa**.

10. (E)

The job of the **Students** constructor is to initialize **listSize**, **studentList** and **lastIndex**.
Only answers **(D)** and **(E)** address the three attributes.
Answer **(D)** incorrectly creates an array of **Students** elements.
Answer **(E)** correctly creates an array of **Student** elements.

11. (B)

The postcondition indicates that a new **student** object is added at the **lastIndex.**
This means that **lastIndex** must be incremented before a new record is added.

12. (C)

The **studentList** array is used to store **student** data.
Answer **(E)** uses **studentList**, but starts at the wrong index value.
Answer **(C)** uses **studentList** and start at the correct array index value of **0**.

13. (C)

(I) demonstrates *multi-level inheritance* because a **Fly** *is-an* **Insect**, and an **Insect** *is-an* **Animal**
(IV) demonstrates *multi-level inheritance* because a **Mouse** *is-a* **Mammal**, and a **Mammal** *is-an* **Animal**

14. (D)

Choice **(A)** is not good because we are using inheritance so we do not have to copy the information.
Choice **(B)** is not good because **public** access for attributes is bad for data security.
Choice **(C)** is not good because **super** does not give you access to **private** data.
Choice **(D)** is good. Moving the method will mean it has access to **numLegs**.
The **Insect** class will still be able to access the **getNumLegs** method via inheritance.
Choice **(E)** is not good because if the methods is copied and not moved, there will still be a method in one **class** trying to access the **private** data of another.

15. (E)

What is important to realize here is that the **Animal** constructor requires an **int** parameter. The constructor of any subclass of **Animal** must first have a **super** statement to send an **int** value to the **Animal** constructor which will be processed first. (I) and (II) do not have this. (III), (IV) and (V) do.

Chapter 5 Solutions
Static Java 1D Arrays

01. (E)

k	0	1	2	3	4	5	6	7	8	9
temp[k]	1	3	4	7	11	18	29	47	76	123

02. (B)

Method **method0202b** exchanges two adjacent **list** array elements.
Method **method0202a** compares adjacent **list** array members.
If the two array elements are not in ascending order, **method0202b** is called to perform a swap.
After one pass the largest-value **list** element is in the last array location.

03. (E)

method0503 returns the addition and subtraction of all the members of the **list** array.
Implementation 1 correctly adds the even indexed elements and subtracts the odd elements.
Implementation 2 is incorrect. It adds the even index elements and subtracts the even elements.
Implementation 3 is also correct, but starts with the first array element as the **temp** value.

04. (E)

method0504 is a sort routine, or so it seems.
The sort routine requires a **swap** method. The **swap** method is called, but alters nothing.
The values of parameters **a** and **b** are exchanged, but the values of the **list[q]** and **list[q+1]** are not exchanged, because Java parameters are passed by value.
The end result is that the elements in the **list** array are unchanged.

05. (B)

The loop structure of question 05 exchanges two **list** array elements during each iteration.
The exchange is between the first element and the last element.
This is followed by an exchange between the second element and the second-to-last element.
It may give the impression that the list will be reversed, but choice **(c)** is wrong.
When the exchange process reaches the halfway point the list is reversed, but exchanges continue.
During the second half of the loop all the elements are exchanged back to their original position.

06. (C)

The loop structure in question 06 traverses every **list** array element and divides by **list[1]**.
The value of **list[1]** is **5** and answer **(b)** may seem logical as each element is divided by **5.**
However, **list[1]** is also divided by **list[1]** and the result is that the value of **list[1]** becomes **1.**
This means that the remaining **list** elements are divided by **1** and left unchanged.

07. (B)

All three segments return the correct median for a list with an odd number of elements.
Implementation 1 does not use the two middle numbers for an even list.
Implementation 2 correctly uses the two middle numbers for an even list.
Implementation 3 uses the two middle numbers for an even list, but does not cast to a **double** value.

08. (A)

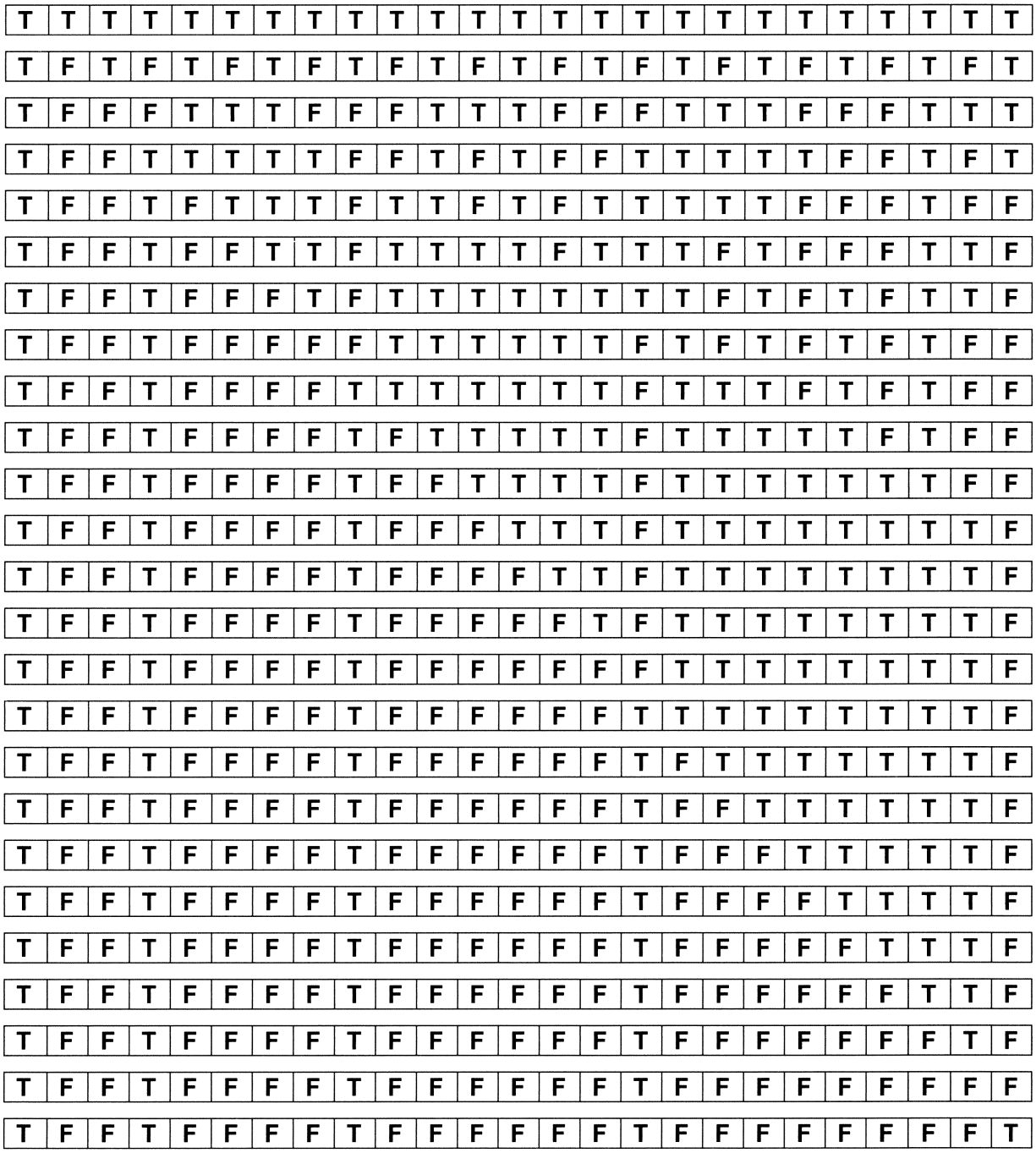
Implementation 1 correctly uses the **for..each** loop to compute the sum of the numbers.
Implementation 2 does not use the **for..each** loop correctly. **item** is an element and not an index.
Implementation 3 uses completely wrong syntax for the **for..each** loop.

09. (D)

Let us take this one step at a time:
We create an array of 10 **boolean**s.
The first one (at index 0) is initialized to **true**.
The first **for** loop starts at index 1, and traverses the array making every element the *opposite* of the previous one. Since index 0 stores **true**, Index 1 will store **false**. Index 2 will be **true**, 3 will be **false** and this continues to the end of the array.
The second **for** loop simply displays the contents of the array.

10. (A)

This is an implementation of what some mathematicians call the *Locker Problem*. Imagine a long hallway with lockers labeled 1 through 1000. One student goes through and opens all of the lockers. A second students goes through and closed all of the even numbered lockers. A third student goes through and changes the state (opens if closed, closes if open) of every third locker. This process continues until the 1000th student closes locker number 1000. Then the question, which lockers are left open? For simplicity the size of the *hallway* was reduced from 1000 to 25. Keep in mind that for this program we are not using index 0 at all. If you trace through the program you would get the following:



At the conclusion, the value of true is stored only at indexes 1, 4, 9, 16 and 25. You probably noticed that these are perfect squares and there is a reason for that. Most numbers have an even number of factors. In the case of the *Locker Problem* an even number of factors means that for every time a locker is opened, it is also closed. However, perfect squares have an *odd* number of factors. This causes the locker to stay open, or in the case of this program, the value of **true** to be stored.

11. (D)

The first **for** loop in this program traverses the array and copies the last element to the first, the second to last element to the second, the third to last element to the third and continues until the first element is copies to the last. The temptation is to think that this would reverse the list, but once you pass the halfway point, you are copying values that have already changed.

12. (D)

This program essentially is a bubble sort which sorts in descending order. There is only one change. In the outer **for** loop, the counter starts counting at 2 instead of 1. This results in one less comparison pass being made by the inner loop. So while the first 9 numbers appear sorted, it never even saw the 10^{th} number.

13. (D)

As this program traverses the array, the value at every even index is tripled and the value at every odd index is incremented by 1.

14. (B)

For this program to work, we need to store the index of the last element in the original array. The variable used for that is **k**. Implementation 1 is not good because **k** stores the length of the array, not the last index. To satisfy the postcondition, the first item of the original array should be in the new array only once in the middle index. In Implementation 3, after the reverse list is copies **k** starts at index 0 to copy forwards. This duplicates the element at index 0 which is already in the new array. Implementation 2 avoids this problem by starting at index 1 when copying the forwards values.

15. (B)

Implementations 1 and 3 do not work because neither first checks the length of the 2 arrays.

Chapter 6 Solutions
Static Java 2D Arrays

01. (A)

A 2D array is a 1D array of 1D arrays. The number of rows is the same as the number of 1D arrays in the 2D array. The **length** field will give you this information.

02. (C)

A 2D array is a 1D array of 1D arrays. For the number of columns, you need the **length** of an individual array (or row) within the 2D array, which requires the single index. Frequently index **[0]** is used. index **[r]** is also used to indicate the particular row you are working with. Index **[c]** has the potential to crash because there are more columns in this 2D array than there are rows.

03. (B)

When you trace through the **createSquare** method, you see that the value of the variable **q** is what gets stored at every index of the matrix. This value keep incrementing as each row is traversed. Before a new row is started, q starts over at 1. So why is it that every row is not the same? The key is the inner **for** loop. It does not start counting at index 0. It starts at index **r**. So when we are in row 0, we start at column 0. When we are in row 1, we start in column 1. When we are in row 2, we start in column 2 and so forth. This is why we see a "triangle" of zeroes in the bottom left corner.

04. (A)

When you trace through the **createSquare** method, you see that the variable **q** simply is storing a copy of the value in variable **r**, the row variable. At every index, what is stored is **q + c** which might as well be **r + c**. Every index stores the sum of its row and column values. This causes the symmetric appearance in the matrix. Here is an example. Both row 1 column 3 and row 3 column 1 will have the value of 4 stored because 1+3 and 3+1 both equal 4.

05. (E)

This is the implementation of the classic magic square problem. A magic square is a square where every row, every column, and both diagonals add up to the same amount. Finding the answer to this particular problem does not require knowledge of the magic square. Before the **for** loop even begins, the number **1** is place in the top row and center column. That is enough to determine that E must be the answer.

06. (D)

This shows what happens when **length** is not used properly with matrixes. In the outer **for** loop **square.length** will give us the number of rows, but in the inner **for** loop we should be using something like **square[0].length** or **square[row].length**. Since the problem uses **square.length** again for the columns the program behaves as if the number of rows and columns are the same and displays what is essentially a square matrix.

07. (B)

Unlike the previous program, this program properly displays the content of the matrix.

08. (C)

size is storing the number of rows. The largest row index therefore can be **size-1**.

09. (B)

n	3			
p		0	1	2
n-p-1		2	1	0
output		3	5	7

10. (D)

The inner **row** loop copies the value of the current column value to each row.
The first time that the inner loop executes all values in the first column are the same.
The second iteration puts all the same values in the second column.
This process continues until the values in every column are the same.

11. (C)

Consider the matrix below of three rows with four columns.

111	222	333	444
222	333	444	555
333	444	555	666

The conditional statement **if (row == r || col == c) matrix[row][col] = 0;**
is true when **row** equals **r** and **col** equals **c**.
Whenever the statement is true the current **matrix** element becomes **0**.
This will result in the following array appearance.

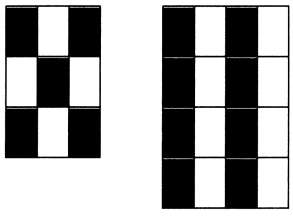
111	222	0	444
0	0	0	0
333	444	0	666

12. (B)

Segment 1 reverses the length of the rows and columns and will not works when the matrix row and col values are not the same.
Segment 2 is correct.
Segment 3 uses the same length for rows and columns. This only works for square matrices.

13. (B)

When you look at the program, the first nested **for** loop traverses the matrix and fills it up with alternating 1s and 2s. In the second nested **for** loop, the 1s cause a space to be displayed while the 2s (or any other number) will cause a solid square (**█**) to be displayed. The question is when will this display a checkerboard/chess board pattern? Let us look at a 3x3 and a 4x4 for some reference:



When the size of the matrix is odd, the next row starts with the opposite of the previous row and we get the checkerboard pattern.

14. (A)

Refer to the previous display. When the size is even, every row will start with a solid square and we will get vertical columns.

15. (E)

Horizontal bars is not a possible display with this program because the within each row alternating solid squares and spaces are printed.

Chapter 7 Solutions
Strings

01. (D)

The *immediate* value of **s1** is a memory reference, which references "Aardvark".
The *immediate* value of **s2** is also a memory reference, but different from **s1**.
The == operator compares immediate values, which works for primitive data types.
With objects == equality is not accurate, and displays "not equals" first.
After the statement **s3 = s1** the both objects now store the same memory reference.
At this point the == operator determines they are equal and displays "equals".

02. (A)

The **equals** method is redefined for the **String** class and compares the *deep* values.
In this case only "equals" is displayed.

03. (C)

The **s1.indexOf(s2)** method returns the starting index of **s2** in **s1**.
String "shin" starts at index **3** in "sunshine".

04. (A)

Method call **s1.substring(1,4)** returns the substring that starts at index **1** of **s1**.
s1 is "racecar" and index **1** is **a**.
The substring ends at the character before index **4**, which is **3.**
This means that string "ace" is returned.

05. (C)

The loop in question 05 repeats three times and concatenates three strings together.
The table below shows the substring returned for each repetition.

k	1	2	3
substring(0,k)	A	AB	ABC
s2	A	AAB	AABABC

06. (D)

Method **substring(k)** returns the substring from index **k** to the end of the string.
The table below shows the substring returned for each repetition.

k	1	2	3
substring(0,k)	BCD	CD	D
s2	BCD	BCDCD	BCDCDD

07. (E)

The **s2.indexOf(s1)** method returns the starting index of **s1** in **s2**.
String "FOX JUMPS" does not appear in **s1** and returns **-1**.
The **s3.indexOf(s1)** method returns the starting index of **s1** in **s3**.
String "FOX JUMPS" starts at index **16** of **s3**.

08. (D)

Method **method0708** uses the **substring** method to take one character at a time from string **s**.
The single character starts from the back of **s** and works forward.
Each character that is returned is concatenated to **t**.
The result is that **t** returns the reverse string of **s**.

09. (D)

Method **s.substring(s.length()-1,s.length());** returns the last character of **s**.
The loop repeats as many times as there are characters in **s**.
The result is that **t** returns a string of the last characters of **s**.

10. (E)

Method **method0710** builds a string **t** which is the reverse of parameter **s.**
Single characters are retrieved from string **s** with the **substring** method.
The **equals** method compares **s** with **t** and returns **true** if they are equal.

11. (B)

Method **method0711a** returns **s.substring(1)** which returns a string from index 1 to the end.
Method **method0711b** returns **s.substring(1,s.length())** which returns a string from 1 to the end.
Both methods will always return the same substring for all values of **s.**

12. (E)

Both methods **method0712a** and **method0712b** return an index value.

Method **method0712a** returns **s2.indexOf(s1)**, which is the starting index of **s1** in **s2**.
Method **method0712b** returns **s1.indexOf(s2)**, which is the starting index of **s2** in **s1**.

The only time that these index values can be identical is when **s1** equals **s2**.

13. (B)

When comparing "aardvark" to "vark" the computer subtracts the code for "a" from the code for "v". Since "a" comes before "v" the result will be a positive number. Technically speaking, since we are subtracting the 1st letter's code from the 21st letter's code, the result would be 20.

14. (C)

When comparing "vark" to "aardvark" the computer subtracts the code for "v" from the code for "a". Since "v" comes after "a" the result will be a negative number. Technically speaking, since we are subtracting the 21st letter's code from the 1st letter's code, the result would be -20.

15. (E)

Since "racecar" has 7 letters in it, substring(4,7) and substring(4) yield the same string. When we use **compareTo** to compare these strings, the result will be 0 because the strings are equal.

Chapter 8 Solutions
Number Systems

01. (B)

1 10 11 **100 101 110 111 1000** 1001 1010 1011

02. (E)

96 97 98 **99 9a 9b 9c 9d**

03. (A)

$2 \times 5^3 + 1 \times 5^2 + 3 \times 5^1 + 2 \times 5^0$ =
$2 \times 125 + 1 \times 25 + 3 \times 5 + 2 \times 1$ =
$250 + 25 + 15 + 2$ = **292 base-10**

04. (C)

$1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$ =
$16 + 0 + 4 + 0 + 0$ = **20 base-10**

05. (E)

$1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$ =
$1 \times 128 + 0 \times 64 + 1 \times 32 + 0 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1$ =
$128 + 0 + 32 + 0 + 8 + 0 + 2 + 0$ = **170 base-10**

06. (D)

$1 \times 16^2 + c \times 16^1 + b \times 16^0$ =
$1 \times 256 + 12 \times 16 + 11 \times 1$ =
$256 + 192 + 11$ = **459 base-10**

07. (A)

$128 + 64 + 32 + 16 + 8 + 4 + 2 + 1$ =
1 1 1 1 1 1 1 0 = **11111110 base-2**

08. (D)

512 + 256 + 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 =
1 0 1 0 1 0 1 + 0 + 1 + 1 = **1010101011 base-2**

09. (D)

1000 / 5 = 200 remainder ***0***
200 / 5 = 40 remainder ***0***
40 / 5 = 8 remainder ***0***
8 / 5 = 1 remainder ***3***
1 / 5 = 0 remainder ***1*** **= 13000 base-5**

10. (B)

505 / 16 = 31 remainder ***9***
31 / 16 = 1 remainder ***15 = f***
1 / 16 = 0 remainder ***1*** **1f9 base-16**

11. (D)

a1b2c3d4e5f6 base-16 = ??? base-2

base-16	a	1	b	2	c	3	d	4	e	5	f	6
base-2	1010	0001	1011	0010	1100	0011	1101	0100	1110	0101	1111	0110

= 1010 0001 1011 0010 1100 0011 1101 0100 1110 0101 1111 0110 base-2

12. (E)

1111 1110 1101 1100 1011 1010 base-2 = ??? base-2

base-2	1111	1110	1101	1100	1011	1010						
base-16	f	e	d	c	b	a						

= 1010 0001 1011 0010 1100 0011 1101 0100 1110 0101 1111 0110 base-2

13. (D)

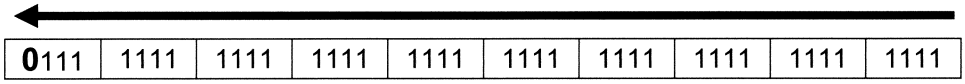
It is tempting to automatically think that the next number after 9999 is 10000, but in base-16 the next digit after 9 is A, so the next number after 9999 is 999A.

14. (D)

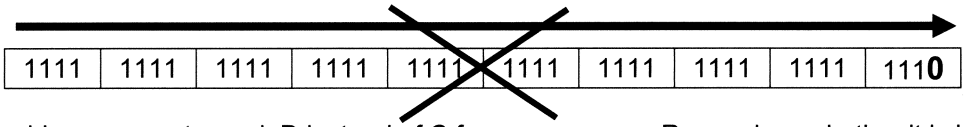
In base-16, the largest digit is F which represents 15. The largest six digit number in base-16 would be composed of 6 Fs.

15. (E)

When you separate a base-2 number into different nibbles (groups of 4 bits – half a byte) you need to remember to start on the right side and work left. If you do not have a multiple of 4 bits, you can add *leading* zeros as shown below.



A common pitfall here is to start on the left side and work right. This is natural because we read left to right. When you run out of bits, some people make the mistake of adding *trailing* zeros.



This would cause you to mark D instead of C for your answer. Remember, whether it is base-2 or base-10, adding *trailing* zeros alters the value of the number. Adding *leading* zeros does not.

Chapter 9 Solutions
The ArrayList Class

01. (E)

Segment 1, 2 and 3 use **<String>**, **<Integer>** and **<Double>** classes respectively to declare a generic **ArrayList** object.
Segment 4 uses **<int>** which is a primitive data type and not a class.

02. (C)

The **add** method stores a new element at the end of an **ArrayList** object.
The index of an array object starts at **0**.
The loop structure starts counting at **1** and the **get** method starts with "Ann".

03. (B)

The **set** method is a return method, which returns the current value before replacing the value.
This means that the first loop displays all the original array elements.
The second loop displays the replaced values, which is "Isolde" for each element.

04. (C)

The overloaded **add** method adds a new element at the indicated index and all elements, starting at the index values are shifted one element higher.
Twice an element is inserted at index **2**. Index **2** goes from "Joe" to "Bob" to "Ann".

05. (A)

The **remove** method removes an array element at the indicated index and shifts every element at a higher index number to one number lower.
This process is done twice. First "Tiger" is removed and then "Aardvark".
Be careful to use two steps. Initially, "Aardvark" is at index **4**, but becomes **3** after the first removal.

06. (B)

Question 06 combines the **get** method with the **set** method in one program statement.
With each iteration the **get** method *gets* the previous element and **sets** it to the current element.
The process starts at index **1** and the result is that every element becomes the value at index **0.**

07. (C)

The loop structure visits every **list** element.
The **Integer** object is converted to an **int** value and added to **p**, which is a growing *sum.*
After the loop, **p** - which is the sum of the values - is divided by the **list** size.
The result, which is the *mean*, is returned.

08. (D)

Method **reverseList** is meant to reverse the elements stored in the **intArray**.
Implementation 1 and **Implementation 2** both reverse **intArray** values and stop halfway.
Implementation 3 reverses elements and continues past the halfway point, which reverses the elements back into their original position.

09. (C)

Method **mystery** compares adjacent **intArray** elements.
If the **kth** element is greater than the **(k+1)th** element, three statements are processed.
These three statements exchange the element values.
The result is that the largest element is constantly exchanged until it reaches the end of the array.

10. (C)

Method **sortList** requires a *swap* segment after the **if** condition.
Implementation 1 does not use a **temp** memory location and looses one value with each swap.
Implementation 2 does not declare the **temp** variable.
Implementation 3 correctly declares and uses the **temp** variable to swap the elements.

11. (C)

Question 11 requires accessing a field in an object, which is an object of another object.
The **get** method must be used first to access the individual **students** element.
Then the **getName** and **getAge** methods are used to access the individual **Student** record.
Implementations 1 and 2 both access **Students** and then the individual record.
Implementation 3 incorrectly tries to access **name** and **age** directly.

12. (B)

Question 12 checks the redefinition of the **toString** method.
Method **toString** returns a **"["** left bracket, followed by **name, age** and a **"]"** right bracket.

13. (E)

Segment 1 is an ArrayList which can store any number of objects.
Segment 2 is a Java static Array of 1000 ArrayLists
Segment 3 is an ArrayList whose capacity is already set to 1000 instead of the default of 10.
Note: capacity and size are 2 different things. Capacity is how many elements the array can hold. Size is how many the array actually is holding. With a dynamic array, when the capacity is exceeded it recreates the array with a larger capacity.

14. (E)

Only list2 uses generics, but both lists start empty and are dynamic. list1 can store any object and list2 can only store Integer objects, so we can say both lists can store Integer objects.

15. (B)

When generics are used, class casting is not necessary.

Chapter 10 Solutions
Recursion

01. (B)

Use **method0901(4,3)** as an example to see the pattern.
The pattern you observe below is **4 + 4 + 4** or **a + a + a**, which is **a * b**.

Call #	a	b	Method Returns
1	4	3	4 + method0901(4,2)
2	4	2	4 + method0901(4,1)
3	4	1	4
method0901(4,3) == 4 + 4 + 4 == **12**			

02. (C)

Use **method0902(4,3)** as an example to see the pattern.
The pattern you observe below is **4 * 4 * 4** or **a * a * a**, which is $\mathbf{a^b}$.

Call #	a	b	Method Returns
1	4	3	4 * method0902(4,2)
2	4	2	4 * method0902(4,1)
3	4	1	4
method0902(4,3) == 4 * 4 * 4 == **64**			

03. (C)

Use **method0903(5)** as an example to see the pattern.
The pattern you observe below is **5 * 4 * 3 * 2 * 1** or **a * a-1 * a-2 * ... 2 * 1** which is **a!** or **a-factorial**.

Call #	a	Method Returns
1	5	5 * method0903(4)
2	4	4 * method0903(3)
3	3	3 * method0903(2)
4	2	2 * method0903(1)
5	1	1 * method0903(0)
6	0	1
method0903(5,3) == 5 * 4 * 3 * 2 * 1 * 1 == **120**		

04. (D)

This recursive method can very easily result in computing the wrong values.
It is tempting to think that the method will return **7 + 6 - 5 + 4 - 3 + 2 - 1 = 10**.

Call #	a	Method Returns
1	7	7 + method0905(6)
2	6	6 - method0905(5)
3	5	5 + method0905(4)
4	4	4 - method0905(3)
5	3	3 + method0905(2)
6	2	2 - method0905(1)
7	1	1

Remember that unfinished recursive calls are completed in a **LIFO** (Last In, First Out) sequence.
This means that the computation must be done as follows.

1 is returned and subtracted from **2**, which now equals **1**.
1 is returned and added to **3**, which now equals **4**.
4 is returned and subtracted from **4**, which now equals **0**.
0 is returned and added to **5**, which now equals **5**.
5 is returned and subtracted from **6**, which now equals **1**.
1 is returned and added to **7**, which now equals **8** and is the final answer.

05. (C)

Call #	x	y	Method Returns
1	7	2	method0905(6,3)
2	6	3	method0905(5,4)
3	5	4	method0905(4,5)
4	4	5	4
method0905(7,2) == **4**			

06. (C)

Method **method0906** is a recursive implementation of using *Euclid's Algorithm* to compute the **GCF**.

Call #	n1	n2	rem	Method Returns
1	150	135	15	method0906(135,15)
2	135	15	0	15
method0906(150,135) == **15**				

07. (D)

Method **method0907b** is *Euclid's Algorithm* to compute the **GCF**, described in question 06.

A call to **method0907a(120,108)** generates the call **method0907b(120,108)**, which returns the **GCF** of **12**.

Method **method0907a** computes the **LCM** with the **GCF 12**.
The result is **120 / 12 * 108 = 1080**.

08. (A)

Use **method0908(5)** as an example to see the pattern.

Call #	q	Method Returns
1	5	method0908(4) + method0908(4)
2	4	method0908(3) + method0908(3)
3	3	method0908(2) + method0908(2)
4	2	method0908(1) + method0908(1)
5	1	1
6	0	
method0905(7,2) == **(1+1) = 2, (2+2) = 4, (4+4) = 8, (8+8) = 16, (16 + 16) = 32** **The pattern that emerges is $2^5 = 32$ or in general 2^n**		

09. (D)

Call #	s	n	Method Prints	Method Call
1	TANGO	5	TANGO	method0909("TANG");
2	TANG	4	TANG	method0909("TAN");
3	TAN	3	TAN	method0909("TA");
4	TA	2	TA	method0909("T");
5	T	1	T	method0909("");
6	empty	0		

10. (E)

Question 10 is like question 09 with *pre-recursion*. This means that the recursive call is made before any output is printed. Unfinished recursive calls are stored on a temporary stack and executed later in a **LIFO** sequence.
The result is that the output is displayed in reverse order from question 09.

11. (E)

The method **linear2** is incorrect, because it initializes **k = 0;** at the start of each method execution.

The method **linear3** is correct, because it adds the **k** parameter to the method heading and increments the **k** value inside each recursive call.

The method **linear4** is correct, because it adds the **k** parameter to the method heading and increments the **k** value before each recursive call.

12. (D)

The method **binary2** is correct, because it adds the **lo** and **hi** parameters to the method heading. Each recursive call alters either **lo** or **hi** appropriately inside each recursive method call.

The method **binary3** is incorrect, because it initializes **lo** and **hi** to the same value at the start of each method execution.

The method **binary4** is correct, because it adds the **lo** and **hi** parameters to the method heading. Each recursive call alters either **lo** or **hi** appropriately before each recursive method call.

13. (B)

The **boohiss** method compares the first and last characters. If they are different **false** is returned, otherwise a recursive call is made with a substring that has the first and last characters removed. this continues until what is left has a length of 1 or 0, in which case **true** is returned.

14. (D)

A string is a palindrome if it is spelled the same way forwards and backwards. In a recursive sense, we could say a string is a palindrome if the first and last characters match and the string between them is a palindrome. This would require a base case to make it stop. A string with 1 or 0 characters is a palindrome by definition. Another way the recursive process stops is if it finds 2 characters that do not match.

15. (C)

This odd method has a recursive call as a parameter for the **concat** method.
Note: the same thing could be accomplished with **temp = temp + mystery(a,b);** or
temp += mystery(a,b); A recursive variable trace is the best way to see what is happening:

```
public static String mystery(String a, String b)
{
    String temp = "";
    int alen = a.length();
    int blen = b.length();
    if (alen < 2)
        return b;
    else if (alen > blen)
    {
        temp = a.substring(0,2);
        a = a.substring(2);
    }
    else
    {
        temp = b.substring(0,2);
        b = b.substring(2);
    }
    return temp.concat(mystery(a,b));
}
```

Call#	a	alen	b	blen	temp	Method mystery returns
1	JESSICA	7	brenda	6	JE	"JE" + mystery("SSICA","brenda")
2	SSICA	5	brenda	6	Br	"br" + mystery("SSICA","enda")
3	SSICA	5	enda	4	SS	"SS" + mystery("ICA","enda")
4	ICA	3	enda	4	en	"en" + mystery("ICA","da")
5	ICA	3	da	2	IC	"IC" + mystery("A","da")
6	A	1	da	2	N/A	"da"

"JE" + "br" + "SS" + "en" + "IC" + "da" = "JEbrSSenICda"

Chapter 11 Solutions
Implementing Methods and Redefining Methods

01. (E)

An interface declaration is completely abstract.
Any implementation of an interface must - at a minimum - define every interface method.
Optionally, the implementation may define additional methods and attributes.

02. (D)

In this question the **ArrayList** is used generically as **ArrayList<String>**.
Method **addList** accepts a parameter that is any **Object** variable.
Implementation 3 attempts to use the **add** method with an **Object** and it needs a **String**.

03. (C)

Implementation 1 uses the **for..each** loop incorrectly and generates a syntax error.
Implementation 2 uses the **remove** method incorrectly. An index is required and not an element.
Implementation 3 is correct.

04. (B)

The **toString** method is redefined for the **ArrayList** class.
The format shown by answer **(B)** represents the redefinition return of the **toString** method.

05. (C)

Choice I will display a memory reference, because **toString** has not been redefined properly.
Choice II will create a syntax error, because the **toString** method must return a **String**.
Choice III correctly returns the number of quirks stored by the object with the **getQuirks** method.

06. (E)

Choice I is correct because **toString** is redefined properly.
Choice II is correct because **toString** is defined properly. It is not necessary to use the **toString** method call in an output statement, but it is correct.
Choice III correctly returns the number of quirks stored by the object with the **getQuirks** method.

07. (D)

The **toString** method needs to be redefined to behave like it does for **ArrayList**.
Implementation 1 is correct. It starts with **"["**, concatenates each element with **", "** and concludes with concatenating **"]"**.
Implementation 2 is correct with a very poor design. The **toString** method actually displays the array elements and then returns an empty string.
Implementation 3 is incorrect. **intArray** does not return a string of elements.

08. (A)

Implementation 1 is correct. Note how **Object source** is casted to **Car** before assignment to **temp**.
Implementation 2 is incorrect. **source** is used without casting to **Car**.
Implementation 3 is incorrect. Method **compareTo** is altered to a **void** method.

09. (C)

The **Banana** class requires the heading **class Banana implements Comparable**.
Without this statement the **Comparable** interface is not implemented.

10. (D)

Class **student** redefines the **equals** method, such that equality is based on **age**.
Objects **tom** and **sue** have different ages and **false** is returned.
Objects **tom** and **ann** have identical ages and **true** is returned.

11. (D)

Question 11 needs to redefine the **equals** method, such that it tests for **gpa** equality.
Implementation 1 is correct. Note how **obj** is casted to **Student** before comparing.
Implementation 2 is correct. The solution is identical to **Implementation 1** with 2 statements.
Implementation 3 is incorrect. Now **obj** is compared before casting to **Student**.

12. (C)

Question 12 essentially states that all **Student** objects are equal.
age and **gpa** are not an issue and should not be compared.
Only **Implementation 3** uses the correct, and simple, solution to return **true** without checking.

13. (D)

The **toString** method will **return** the number of **borbs**, but not display it. This is why (III) does not work. (II) works and explicitly calls the **toString** method, but that is not necessary. (I) also works because **toString** is automatically called by **print** and **println**.

14. (A)

The **toString** method only returns the **name** of the student, so that it what is displayed.

15. (C)

The **toString** method returns the concatenation of the **name**, **age** and **gpa** with **"\n"** (new line) characters in-between to make each piece of information display on a separate line.

Chapter 12 Solutions
Java Standard Libraries

01. (A)

Math.random() returns a number **x**, such that **0 <= x < 1** or a range of **[0 . . 0.9999999]**
Multiply each number times **50** and the range becomes **[0 . . 49.9999999]**.
Add **10** and the range becomes **[10 . . 59.9999999].**
Apply the **(int)** cast and the range finalizes to **[10 . . 59]**.
Only **(A)** generates numbers in the proper range.

02. (C)

The **[400 . . 1600]** range has **1201** different numbers.
Multiply **1201** times **rndDouble** followed by an **(int)** cast gives the range **[0 . . 1200]**.
Add the smallest value **400** to the numbers gives the desired range **[400 . . 1600]**.

03. (D)

Both **Segment 1** and **Segment 2** use **Math.random() * 100** to get the **[0 . . 99.9999999]** range.
Segment 1 will display numbers in the given range, which will be **double** numbers.
Segment 2 will display numbers in the given range, but after **(int)** cast, will display integers only.

04. (B)

Question 04 has a loop that iterates **10** times with a **temp** variable initialized to **100**.
The statement **temp *= -1;** toggles **temp** between **+100** and **-100**;
Variable **temp** will be **-100** whenever loop counter **k** is an odd number.
The **if** condition is **true** whenever **k** is an odd number, when **temp** is also **-100**.
The statement **Math.sqrt(-temp)** computes the square root of **(- -100)**, which is always **10.0**.

05. (E)

The value of **temp** is not relevant.
Computing the **sqrt** and the **abs** are also not relevant.
This program includes a division by **0**, which will result in an **ArithmeticException** error.

06. (D)

Math.pow(a,b) computes a^b, which in this question is $(2.5)^{-2}$.
$(2.5)^{-2}$ equals $1/(2.5)^2$, which equals **1/6.25**, which equals **16/100** or **0.16**.

07. (C)

The table below shows the values of **k** as they are entered in the **list** array.
The second loop accesses the **list** array and adds each element to a growing **sum** variable.

k	10	20	30	40
index	0	1	2	3
sum	10	30	60	100

08. (E)

String **s1** equals **"GridWorld"** and method **s1.substring(4,5)** returns a substring, which starts at index **(4)**, which is **W** and ends before index **(5)**, which means that only **W** is returned.

09. (A)

Question 09 goes through several steps with different method calls.
First the index is returned with **s1.indexOf("W")**, which is index **4** and assigned to **start**.
Then **s1.substring(start)** returns the string from **W** to the end of the string, which is "**World"**.

10. (B)

In this program segment, four names are added to the **names** array.
Each name is added at the end of the **ArrayList** object.
The list now contains **[Tom, Sue, Joe, Ann]**.
The fifth **add** method call, indicates index **(2)** and inserts **"Bob"** , pushing **"Joe"** and **"Ann"** over.

11. (D)

Like question 10, this program segment initially creates an array with **[Tom, Sue, Joe, Ann]**.
The statement **System.out.println(names.set(2,"Bob"));** returns **"Joe"** and replaces with **"Bob"**.
The statement **System.out.println(names.get(2));** returns **"Bob"**.

12. (A)

Like question 10, this program segment initially creates an array with **[Tom, Sue, Joe, Ann]**.
The statement **System.out.println(names.remove(1));** returns **"Sue"**.
The statement **System.out.println(names,remove(3));** tries to return the element with index **(3)**.
The highest available index is **(2)** and the result is an **IndexOutOfBoundsException** error.

13. (D)

(A) and **(B)** do not work because they both check to see if one string is in itself and never look at the other string. **(C)** is backwards and returned **true** if **a** is NOT **a** substring of **b**. **(E)** only returns **true** if the strings are equal.

14. (E)

An interface can be used to declare an object, but not to construct it.

15. (D)

Implementation 1 does not work because it attempts to add up **Double** objects without using the **doubleValue** method to convert the **Double** objects to **double** primitive data types.

Chapter 13 Solutions
Algorithms and Informal Algorithmic Analysis

01. (A)

The *linear search* implementation in question 01 is very inefficient.
The loop structure will repeat for the entire length of the **list** array.
Even after the **searchNumber** is found, the comparisons will continue.

02. (C)

Implementation 1 is incorrect. The entire list is searched and **found** will probably be wrong.
Implementation 2 is incorrect. The loop is conditional, but only based on the array length.
Implementation 3 is correct. This time the loop checks the **length** variable and also **found**.
If the **searchNumber** is reached, **found** becomes **true** and the loop exits.

03. (A)

It is not required that data is sorted for the correct execution of a *linear search* implementation.
There are situations when the *search item* is not in the data list.
In such a case the entire list is searched before it is positive that the *search item* does not exist.
In a sorted list the loop can exit when the data order indicates that further searching is futile.

04. (C)

Method **whatSort** only has a single loop, which compares adjacent array elements.
If **list[q]** is greater than **list[q+1]** the elements are exchanged.
The result after one pass is that the largest array element is at the end of the list.

05. (A)

During each pass of the inner loop in the **selectionSort** method, each element is compared.
A **temp** variable stores the index of the first **intArray** element.
Each **intArray** element is checked to see if it is smaller than the element stored at **temp**.
If the current element is smaller, then **temp** stores the index of the now smallest element.
At the end of the inner pass, a swap is made and the smallest array element is placed in front.
This process repeats and the result is that the array is sorted in *ascending* order.

06. (D)

In a binary search find the two binary values that bound the number of elements.
Number **1025** is bounded by **1024** and **2048**.
These numbers are $\mathbf{2^{10}}$ **(1024)** and $\mathbf{2^{11}}$ **(2048)**. This means that at most **11** comparison are made.

07. (A)

The first number **(100)** is at index **[0]** and the last number **(200)** is at index **[10]**.
A binary search will start by using the *midpoint* at **(0 + 10)/2 or 5**.
The number at index **[5]** is **150**.
This means that only **1** comparison is made to find **150**.

08. (C)

The *merge sort* starts by dividing the list into two separate lists.
Each half of the list is stored during the *unfinished* recursive method calls.
The smaller lists are divided repeatedly until only one element is left in the list.
At that time the actual merging process can put together two lists of 1 and than two lists of 2.
This merging process continues until all the *halved* lists are back in place, but now in sorted order.

09. (C)

When a list has **1** element, the *binary search* and *linear search* have identical executing times.
When a list has **64** elements the *binary search* makes at most **8** comparisons.
Now with **64** elements the *linear search* can make up to **64** comparisons.
By the time the list grows to **1,000,000** the *binary search* makes at most **20** comparisons.
The *linear search* must traverse each element of the list and may require **1,000,000** comparisons.
As the list grows bigger and bigger, the *binary search* grows faster compared to the *linear search*.

10. (D)

Question 10 tests to see if you can count the number of time that nested loop iterate.
The outer loop starts at **1** and ends at **n-1**.
The inner loop starts at **1** and also ends at **n-1**.
This means that **count++** is executed **(n-1) x (n-1)** times.
With the parentheses removed that becomes $\mathbf{n^2 - 2n + 1}$.

11. (B)

Question 11 has a set of nested loops that is harder to determine. Using an example helps.
The outer loop is still simple and repeats **n** times.
Using **n = 5**, the inner loop repeats **4**, **3**, **2** and **1** times for a total of **10** times.
This problem shows a pattern with several examples that becomes $\mathbf{(n^2-n)/2}$.

12. (C)

The *selection sort* uses a nested loop, which creates quadratic behavior.
As a list doubles in size, the nested loops quadruples in execution time.
Actual observation of execution times also confirms the quadratic behavior.

13. (A)

This is a search that looks for a particular item and returns its index when found – which is just what you need when deleting a particular item from an array. A search for inserting would not look for a particular item, but rather the index of where that item was to be inserted.

14. (E)

The **for** loop moves over the array elements to make room for the element added in the last statement.

15. (A)

The **for** loop moves array elements back one starting at index. This means whatever was at location **index** is now no longer in the array.

Chapter 14 Solutions
Polymorphism

01. (A)

Choice I is correct. It is not possible to instantiate an object with an interface.
Choice II is incorrect. It is true that **Bank** is not implemented, but this is not a compile error.
Choice III is incorrect. It is true that there are no **print** statements, but this is not a compile issue.
Choice IV is incorrect. Semicolons are required at the end of abstract method headings.

02. (B)

Dance1 is incorrect. Attributes cannot be declared without an assignment in an interface.
Dance 2 is correct. Only abstract methods are declared with method headings.
Dance 3 is incorrect. There cannot be a constructor and there are no semi-colons.

03. (E)

Correct implementation of an interface requires the class heading **class Waltz1 implements Dance**.
It also requires that every abstract method is implemented.
It is allowed to define additional methods and attributes that were not declared in the interface.
All three implementations satisfy the minimum requirements.

04. (E)

This question involves the declaration of abstract classes.
All interface methods are abstract, even if the *optional reserved* word **abstract** is not used.
An abstract class must declare at least one method abstract.
The **abstract** keyword must be used. It is not optional.
Each one of the **Dance** classes correctkly uses the word **abstract** in the class heading.
None of the **Dance** class declarations indicates even a single method as **abstract**.
Therefore none of the declarations are correct.

05. (D)

Question 05 cures many of the problems that were evident in the previous question.
Declarations **Dance1** and **Dance2** satisfy the requirements of an *abstract class*.
Dance3 is not correct. It is not possible to declare an **abstract** constructor method.

06. (A)

Any sub class requires a statement like **class Waltz extends Dance**.
The **Dance** super class includes a constructor, which requires information from a sub class.
All three **Waltz** classes properly use the **extends** class heading.

Waltz1 is correct.	The **Waltz1** constructor uses **super** to pass information to the super class.
Waltz2 is incorrect.	There is no **Waltz2** constructor and the program does not compile.
Waltz3 is incorrect.	There is no **Waltz3** constructor and the program does not compile.

07. (E)

Correct implementation of an interface requires the class heading **class Waltz implements Dance**.
It also requires that every abstract method is implemented.
It is allowed to define additional methods and attributes that were not declared in the interface.
All three implementations satisfy the minimum requirements.

08. (D)

Question 08 shows an example of *overloaded methods.*
Such methods have the same method identifier, but a different *parameter signature*.
This is the case with the two **getData** methods.

09. (A)

Implementation 1 is correct.	It uses the **Animal** identifier to implement *polymorphism* correctly.
Implementation 2 is incorrect.	It uses **Object**, which does not have **getType** and **getMove**.
Implementation 3 is incorrect.	It compares an object identifier with a class identifier.

10. (A)

Declaration 1 is correct.	It uses the correct abstract heading and correct method declarations.
Declaration 2 is incorrect.	Nothing is declared as abstract.
Declaration 3 is incorrect.	It uses the correct abstract heading but has no abstract methods.

11. (D)

Segment 1 is correct.	The interface **Shape** identifier can be used for any implementing class.
Segment 2 is correct.	The class **Object** can be used for any implementing class.
Segment 3 is incorrect.	The **Class** class does not exist.

12. (D)

This question is identical to question 11, except that now an abstract class is used.

13. (A)

In this program, the **MyBank class** implements the **Bank interface** in s straight forward and intuitive manner.

14. (C)

When implementing an interface, every method must be implemented, even if it is not used.

15. (A)

All interfaces are abstract by their very nature. All interface methods are also abstract. Therefore, the keyword **abstract** is not required in an **interface**. For the same reason, the word **final** is optional as well because the only attributes allowed in an interface are **final** attributes. When working with abstract classes, you must specify **abstract**. This is because classes exist which are not **abstract**. An **abstract class** can have both **abstract** methods and *concrete* methods. An **abstract class** can also contain any kind of attribute, **final** or otherwise.

Chapter 15 Solutions
Program Design

01. (D)

A method can only work correctly if the conditions of a program are proper for method execution.
When you take the APCS Examination you will be asked to complete methods.
The *preconditions* of a method must be considered and accepted in writing the method.

02. (C)

A method exists for a purpose. This purpose is usually stated in comments.
The comments that indicate what must be true when the method finishes are called *postconditions*.
Students loose many points on the APCS Examination, because they are not careful in satisfying all the requirements of the stated *postconditions*.

03. (A)

A program exists for a purpose, which is indicated by a client.
The client knows what is necessary.
The software company knows programming.
Nothing can start with the software development until the program specifications are known.

04. (D)

Programs are meant to simulate real life.
Object Oriented Programming takes the objects observed in life and uses them in a program.
In a grade book program the real life object, *a student*, becomes an object in a program.

05. (A)

Program design does not start with final details.
It is possible to talk about the design of program stages without worrying about implementation.
It is also possible to use existing program implementations without understanding them.
This is known as *information hiding*.

06. (E)

Question 06 discusses the fundamental steps of designing a program with classes.
The design of a class requires *data storage*, *method signatures*, *preconditions* and *postconditions.*

07. (E)

The fundamental rock of Object Orient Programming is the emphasis on reliability.
Everything else, like speed, friendliness, cost, etc. is secondary to reliability.

08. (C)

The question indicates that two processes are required.
It is also states that quick access to student records occurs frequently.
A second requirement is an end-of-semester access GPA list.
Method 2 sorts the data, such that students are quickly accessed for updating.
Sorting once per semester is infrequent enough to make that a second priority.

09. (A)

It is easy to go crazy with designing separate classes for every conceivable task.
Unnecessary classes generate unnecessary processing.
Place related data and all the related operations that process the data in one class.
The **Survey** class can easily handle all the program requirements.

10. (E)

This question may be easiest explained by eliminating wrong answers first.
Several classes have some important features that result in immediate elimination.
Answers **(A)** and **(C)** are instantly removed, because they declare the data fields as **public**, allowing some serious side effects to enter the program execution.
Choice **(B)** has **private** data, but no get methods to gain indirect access to the data fields.
Choice **(D)** is fine and has no problems.
Choice **(E)** is almost the same as **(D)**, but it has the improvement of calling **shuffleCards** in the constructor, which makes the use of **CardDeck** objects more reliable.

11. (B)

Object **list1** stores a memory reference to a set of values that are located elsewhere.
Object **list2** does not use the **new** operator, but is assigned the same reference as object **list1**.
The results are that two different objects are referencing the same information and any processing of the referenced data by either object will impact the other object.

12. (B)

Choices **(B)** and **(E)** each store information about the total **donut** statistics.
Choice **(B)** is the better design because of the improved readability that is gained by the different data fields.

13. (A)

A class can contain several methods just like a toolbox can contain several tools.

14. (B)

Like a tool, a method is used for one specific task. Several methods can exist in one class just like several tools can be in your toolbox.

15. (C)

Constructors must always be declared **public**, otherwise constructing new objects would not be possible.

Chapter 16 Solutions
Sample AP Examination I

Section I, Multiple Choice

01. (A)

The outer loop repeats from **1** to **m-1** or **9** times.
The inner loop repeats from **0** to **m-p** times. This loop decreases from **9** times to **0** times.
The inner loop averages **5** times. **counter** is called **9 x 5 or 45** times.

02. (B)

Question 02 is an example of *short-circuiting*. If **A is false**, the entire expression is **false**.

03. (D)

There are **9** array elements, which is the value of **list.length**.
n = list.length - 1 = 8.
The table below shows the **list** values.
When **k >= 5**, **list** elements are replaced by **list[k-n]** values.

k	n - k	list[k]	list[k-n]
0		11	
1		22	
2		33	
3		44	
4		55	
5	3	66	44
6	2	77	33
7	1	88	22
8	0	99	11

04. (E)

Implementation 1 is not correct. It fails to cast **source** to a **Student** object.
Implementation 2 is correct. It casts to a **Student** object and uses nested **if..else** statements.
Implementation 3 is correct. It casts to a **Student** object and returns the difference between the two student ages.

05. (D)

Implementation 1 is correct. All information is returned in one statement.
Implementation 2 is correct. **temp** concatenates and then returns all the information.
Implementation 3 is incorrect. There is no **return** statement.

06. (A)

With the **str** value of "COMPUTER" , the following string manipulation executes.

k	k % 2 != 0	substring(k,k+1)	temp
0	false	Q	Q
1	true	O	QO
2	false	Q	QOQ
3	true	P	QOQP
4	false	Q	QOQPQ
5	true	T	QOQPQT
6	false	Q	QOQPQTQ
7	true	R	QOQPQTQR

07. (C)

Math.random() generates numbers in the range **[0 . . 0.9999999]**
(Math.random() * n) generates numbers in the range **[0 . . (n-1).9999999]**
(int) (Math.random() * n) generates numbers in the range **[0 . . n-1]**
(int) (Math.random() * n) + n generates numbers in the range **[n . . 2n-1]**

08. (C)

This is a question about *information hiding*.
The *preconditions* and *postconditions* of a method describe its functionality.
The *signature* of a method provides the method identifier and its required parameters.

09. (E)

Method **belgium** computes the greatest common factor **(GCF)** of two numbers.
Method **holland** computes the least common multiple **(LCM)** of two numbers with the **GCF**.

10. (A)

This question tests knowledge of *aliasing*.
Both **list1** and **list2** reference the same object.
The changes made to **list1** and to **list2** are changes to the same object.

11. (D)

Question 11 involves both inheritance and composition relationships between classes.
Before a **CamperVan** object is constructed, a super class **Van** object must be constructed.
Before a **Van** object is constructed the attribute **Engine** object must be constructed.
This makes a sequence of **Engine**, **Van** and then **CamperVan** constructions.

12. (C)

The inner **for** loop makes the computer go through a row and swap the first item with the last, the second item with the second-to-last, the third item with the third-to-last and so forth until that row is reversed. The outer **for** loop causes to process to happen for every row. Since every row has its *contents* reversed, we can say that the *columns* are reversed in the matrix.

13. (C)

The **Math.random** method does not use a starting seed.
Each execution starts a new, random number.
The numbers will be different, but they will be random within the **[100 . . 999]** range.

14. (A)

Implementation 1 is correct. The **ArrayList<Vehicle>** declaration handles all three classes.
Implementation 2 is incorrect. The **ArrayList<Car,Ambulance,Truck>** is totally wrong syntax.
Implementation 3 is incorrect. There is supposed to be one **addObjects** method, not three.

15. (E)

Class **Stuff** implements method **compareTo** according to **stuff** values.
Object **s2** is greater than object **s1**.
System.out.println(s1.compareTo(s2)); prints **1**.

16. (E)

Selection **I**, **II** and **III** are the three corner stones of Object Oriented Programming.

17. (E)

An airport is not using OOP.
The same airline agent and the same luggage conveyor belt handles luggage for many destinations.

18. (D)

This problem can easily trick you.
The first method call of **ifElse(4)** goes to **else if (n == 4)** and prints **4**.
This is followed by a recursive call to **ifElse(n - 4)** or **ifElse(0)**
The **ifElse(0)** prints **0** and stops the recursive process.
You may think that display **4 0** is correct.
However, there is unfinished business left from the first recursive call and another **4** is displayed.

19. (C)

Method **toString** is redefined for the **Apple** class.
This means that the quantity and type of apples is displayed.

20. (C)

Method **monkey** performs repeated divisions by **2** of its **nr** parameter.
The remainder of dividing by **2** is concatenated on a **temp** string.
These remainders generate the base-2 representation of the **nr** parameter.

21. (E)

Consider an example using the call **frog(4,7)**
Note that this method count **n** until it is equal to **q** and then returns this value.
This will always be **n** or as the answers reads, it will always be the value of **b**.

p	q	n
4	7	1
		2
		3
		4
		5
		6
		7

22. (E)

p	q	n	n x q + n
11	10	1	1 x 10 + 1 = 11
55	10	5	5 x 10 + 5 = 55
27	8	3	3 x 8 + 3 = 27
98	13	7	7 x 13 + 7 = 98

23. (D)

Method **remove(index)** deletes an **ArrayList** element if it is located at the indicated **index**.
In method **removeNr** the **remove** method is called whenever the loop counter is even.

24. (B)

Local variable **temp** starts by concatenating **s1** and **s2**.
The substring that is returned starts at **s1.length(),** which is the first character of **s2** and finishes at the last character of **temp**.
This means that **sillyString** will always return **string2**.

25. (A)

The **Math.random** method returns a value **x**, such that **0 <= x < 1**.
In this segment an **ArrayList** object stores **n** random numbers returned by **Math.random()**.
A second loop adds all the values to a **temp** variable and divides by **n**.
The result is actually the **mean** of the numbers, but it will be in the same **0 <= x < 1** range.

26. (B)

If **p % q > 0** then **q** or **b** is returned.
If **p** is a multiple of **q** then **p % q == 0** and the recursive call is made.
The recursive call includes parameter **p % q**, which is this case is **0**.
This will cause an **ArithmeticExceptionError** due to division by zero.

27. (E)

Class **CCC** extends abstract class **BBB.**
Abstract class **BBB** declares method **setData**, which is not implemented by class **CCC**.

28. (B)

Implementation 1 is incorrect. Method heading requires **ArrayList<Integer>**.
Implementation 2 is correct.
Implementation 3 is incorrect. The static array is limited to 100 elements.

29. (C)

Without the **extends** keyword, there cannot be an inheritance relationship.
The private data of the **Eye** class is a **Person** object, which is composition.
It sounds illogical to state that "an Eye has-a Person", but composition is based on syntax.

30. (C)

Simplify in stages to get:

(3 + 5 * 10 * 2 + 7) equals (3 + 50 * 2 + 7) equals (3 + 100 + 7) equals 110.

31. (E)

This question involves Java's *short-circuiting*.
This means that a conditional statement is not completely checked when the condition is known.
With **(method1(n) && method2(n))** *short-circuiting* occurs when **method1** returns **false**.

The first time in the loop with **n == 9** both methods return **true** and skip a blank line.
The second time in the loop with **n == 10 method1** returns **false** and **method2** is not called.

32. (A)

The starting value of bool does not matter. The expression **(bool == true || bool == false**) is guaranteed always to be **true**.

33. (B)

Method **salsa** is an implementation of the *selection sort,* which sorts in descending order.

34. (E)

The linear search does not benefit from a sorted list if the search item exists. The linear algorithm will still compare every element along the way. However, if the search item is not in the list than the search can stop at a certain point, if it is known that the list is sorted.

35. (A)

Class **OrpsA** is correct.	It has the correct heading and implements the **Orps** methods.
Class **OrpsB** is incorrect.	The class heading is backwards.
Class **OrpsC** is incorrect.	It has the correct heading, but wrong **Orps** method implementations.

36. (B)

The best-case for a *linear search* in a list of **n** elements is only one comparison.
The worst-case for a *linear search* in a list of **n** elements is to compare all elements.
On average that means there are **n/2** comparisons.

37. (D)

Segment1 is correct. It defines **students** with **MyList<String>**.
Segment2 is correct. It defines **students** with **MyList<Integer>**.
Segment3 is incorrect. It defines **students** with **<E>**, which is not any type of a class.

You do see **<E>** and **E** used in the **MyList** declaration.
The variable **E** for Element belongs in the generic declaration.
It does not belong in a statement that instantiates a new object of a generic class.

38. (A)

According to the Law of Absorption, **a && (a || b)** is equivalent to **a**.
By the same law, **a || (a && b)** is also equivalent to **a**.
Therefore the 2 expression are always equal, for all values of **a** and **b**.
The following truth table may also be helpful:

A	B	A || B	A && B	A && (A || B)	A || (A && B)
T	T	T	T	T	T
T	F	T	F	T	T
F	T	T	F	F	F
F	F	F	F	F	F

39. (D)

As the loop repeats the following changes occur.
John becomes Greg.
Greg becomes Maria.
Maria becomes Heidi.

This creates the list Greg Maria Heidi Heidi

40. (D)

There are 19 elements in the list.
The **1st** comparison is made at **[(0 + 19)/2]** or **[9]** with element **59**.
The **2nd** comparison is made at **[(10 + 19)/2]** or **[14]** with element **87**.
The **3rd** comparison is made at **[(10 + 13)/2]** or **[11]** with element **67**.
The **4th** comparison is made at **[(12 + 13)/2]** or **[12]** with element **74**.
The **5th** comparison is made at **[(13 + 13)/2]** or **[13]** with element **83**.

Section II, Free Response

Question 1.

Part (a).

```
public static ArrayList<Integer> createList(int n)
{
    ArrayList<Integer> temp = new ArrayList<Integer>();
    Random rnd = new Random();
    for (int k = 1; k <= n; k++)
    {
        int rndInt = rnd.nextInt(90) + 10;
        temp.add(new Integer(rndInt));
    }
    return temp;
}
```

Part (b).

```
public static double getMean(ArrayList<Integer> list)
{
    int sum = 0;
    for (int k = 0; k < list.size(); k++)
    {
        Integer intObj = list.get(k);
        sum += intObj.intValue();
    }
    return (double) sum/list.size();
}
```

Part (c).

```
public static double getMedian(ArrayList<Integer> list)
{
    sortList(list);
    double temp = 0;
    int n = list.size();
    if (n % 2 == 0)
    {
        Integer intObj1 =  list.get(n/2 - 1);
        Integer intObj2 =  list.get(n/2);
        temp =  (intObj1.intValue() + intObj2.intValue() ) / 2.0;
    }
    else
    {
        Integer intObj = list.get(n/2);
        temp = intObj.intValue();
    }
    return temp;
}
```

Question 2.

Part (a).

```
public String toString()
{
    String temp = "";
    temp += "String:       " + str + "\n";
    temp += "Palindrome:   " + palindrome + "\n";
    return temp;
}
```

Part (b).

```
private boolean isPalindrome()
{
    String forward = str.toUpperCase();  // provided method
    int n = forward.length();
    String backward = "";
    for (int k = n-1; k >= 0; k--)
        backward += forward.substring(k,k+1);
    if (forward.equals(backward))
        return true;
    else
        return false;
}
```

Question 3.

Part (a).

```
public void addNewProduct(String name, double cost, int amount)
{
    Product temp = new Product(name,cost,amount);
    if (max == 0)
    {
        products.add(temp);
        max++;
    }
    else
    {
        int index = 0;
        while (index < max && name.compareTo((products.get(index)).getItem()) >= 0)
            index++;
        products.add(index,temp);
        max++;
    }
}
```

Part (b).

```
public ArrayList<String> mustOrder()
{
    ArrayList<String> orderList = new ArrayList<String>();
    for (int k = 0; k < max; k++)
    {
        Product temp = products.get(k);
        if (temp.getQuantity() < 20)
            orderList.add(temp.getItem());
    }
    return orderList;
}
```

Question 4.

Part (a).

```
private int contains (Object item)
{
    boolean found = false;
    int index = 0;
    while (!found && index < numItems)
    {
        if (item.equals(set.get(index)))
            found = true;
        else
            index++;
    }
    if (found)
        return index;
    else
        return -1;
}
```

Part (b).

```
public void add(Object item)
{
    int temp = contains(item);
    if (temp == -1)
    {
        set.add(item);
        numItems++;
    }
}
```

Part (c).

```
public void remove (Object item)
{
    int temp = contains(item);
    if (temp >= 0)
        set.remove(temp);
    numItems--;
}
```

Chapter 17 Solutions
Sample AP Examination II

01. (B)

The final value of **n** with value 13 is returned. The inner while loop is never entered, because p is never less than **q**.

p	q	n
15	7	0
		1
		2
		3
		4
		5
		6
		7
		8
		9
		10
		11
		12
		13

02. (E)

Method **swing** will always return true.
This is because the mathematical expression **p - (q * p / q)** is identical to **p % q**.
Both perform modulus (remainder) division.
By hierarchy rules in the first expression the multiplication will be performed first, the integer division second, and the subtractions last.

Method **polka** requires some special conditions. Anytime that **p % q == 0**, **p > q** in all cases, except when **p == q**.

03. (A)

In method **austria** the entire list **ArrayList** object is traversed. The **int** value of each list element is checked and if the value is an odd integer, the element is removed.

04. (C)

Variable **temp** starts by concatenating **s1** and **s2**.
The substring that is printed starts at **s1.length(),** which is the first character of **s2** and finishes at the last character of **temp**. This means that **s2** is printed or "WestPoint".

05. (D)

Implementation 1 is correct. Each element is added to **sum** with **for..each** and divided by **length**.
Implementation 2 is correct. Each element is added to **sum** with **[k]** and divided by **length**.
Implementation 3 is incorrect. Loop is incorrect causing **IndexOutOfBoundsException** error.

06. (A)

p	q	return Value
45	30	columbia(44,30)
44	30	columbia(43,30)
43	30	columbia(42,30)
42	30	columbia(41,30)
41	30	columbia(40,30)
.....	.....	.....
31	30	columbia(30,30)
30	30	30

07. (D)

The result of calling method **egypt** is identical to calling method **congo**.
The two methods look very different, but they perform the exact same process.
Method **egypt** is a recursive implementation of the iterative method **congo** .

08. (E)

The **Plips** class does not redefine the **toString** method.
The statement **System.out.println(plip1);** will use the **toString** method inherited from **Object**.
In that case the result is the class name followed by its hexadecimal memory value.

09. (C)

Cat extends Mammal
Mammal extends Animal
There are two examples of inheritance used.

10. (C)

8 + 8 / 4 + 3 * 6 = 8 + 2 + 18 = 28

11. (A)

This question involves Java's *short-circuiting*.
This means that a conditional statement is not completely checked when the condition is known.
With **(method1(n) || method2(n))** *short-circuiting* occurs when **method1** returns **true**.

The first time in the loop with **n == 4**, **method1** returns **true** and **method2** is not called.
The loop skips a line.
The second time in the loop with **n == 5**, both methods return **false**.

12. (C)

If the initial value of **trueFalse** is **true**, then **trueFalse == true** is **true.**
Furthermore, **!trueFalse == false** is also **true.**

If the initial value of **trueFalse** is **false**, then **trueFalse == true** is **false.**
Furthermore, **!trueFalse == false** is also **false.**

13. (E)

p	list1[p]	q	list1[q]	temp	list2
0	66	0	66	66	
		1	22	22	
		2	11	11	
		3	44	11	
		4	55	11	
		5	33	11	
		6	77	11	11
1	22	1	22	22	
		2	11	11	
		3	44	11	
		4	55	11	
		5	33	11	
		6	77	11	11, 11
2	11	2	11	11	
		3	44	11	
		4	55	11	
		5	33	11	
		6	77	11	11, 11, 11
3	44	3	44	44	
		4	55	44	
		5	33	33	

This pattern continues with the end result that **list2** becomes **[11, 11, 11, 33, 33, 33, 77]**.

14. (D)

The *binary search* is only possible with an ordered list of data.
The *linear search* functions correctly with any type of list.

15. (D)

Class **System** is a subclass of **Object**, which is inheritance.
out is an attribute of System, which is composition.

16. (E)

The worst-case scenario in a search occurs when the search item is not present.
The *linear search* will not realize that the search item is not present until every element is compared.
This requires 5000 comparisons.

17. (A)

The two **sortList** methods can co-exist in the same class if they have different signatures.
This means that the method identifiers might be identical, but the parameter list is different in data type and/or parameter quantity.
The two **sortList** methods in this question have different code in the method body, but the method headings show two identical signatures.

18. (E)

According to the *distributive law* for the operation **and** over the operation **or** the two statements are equivalent. You can also use a truth table to check the equivalence.

P	A	B	A || B	P && (A || B)	P && A	P && B	P && A || P && B
T	T	T	T	**T**	T	T	**T**
T	T	F	T	**T**	T	F	**T**
T	F	T	T	**T**	F	T	**T**
T	F	F	F	**F**	F	F	**F**
F	T	T	T	**F**	F	F	**F**
F	T	F	T	**F**	F	F	**F**
F	F	T	T	**F**	F	F	**F**
F	F	F	F	**F**	F	F	**F**

19. (A)

Math.random()	generates numbers **x**, such that	**0 <= x < 0**
(int) (Math.random() * 9000)	generates numbers **x**, such that	**0 <= x < 8999**
(int) (Math.random() * 9000) + 1000	generates numbers **x**, such that	**1000 <= x <= 9999**

20. (B)

Since $2^{12} = 4096$ and $2^{13} = 8192$, any number **x**, such that **4096 < x <= 8192** takes at most 13 comparisons with a list 5000 elements using a *binary search*.

21. (D)

The value of **x** is copied to parameter **m**.
The outer **p** loop iterates (**1 to m-1**) times, which is **m-1** times.
The inner **q** loop iterates (**0 to m-1**) times, which is **m** times.
Variable counter is incremented **m * (m - 1)** or $m^2 - m$ times.
Since the parameter is **x**, the final answer is $x^2 - x$ times.

22. (D)

This is an example of De Morgan's rule and the Double Negative rule.
Remove the outer parentheses from answer **(D)** and apply DeMorgan's rule.

23. (C)

Method **mystery** exchanges the first element with the last element in **list**.
This is followed by exchanging the second element with the second-to-last element in **list**.
The process continues until the midpoint of **list** is reached.
At that point the entire **list** contents are reversed.

24. (A)

Implementation 1 is correct. It casts to a **Student** object and it returns a boolean value.
Implementation 2 is incorrect. It does not cast to a **Student** object.
Implementation 3 is incorrect. It returns an **int** value, and it must return a boolean value.

25. (C)

Implementation 1 is incorrect. It does not have a line feed at the end of the string.
Implementation 2 is incorrect. It returns a boolean value in place of a string.
Implementation 3 is correct.

26. (A)

k	k % 2 != 0	substring(k)	temp
0	false		
1	true	OLLAND	OLLAND
2	false		
3	true	LAND	OLLAND LAND
4	false		
5	true	ND	OLLAND LAND ND
6	false		

27. (E)

The **random** method of the **Math** class does not take a parameter like **Math.random(100)**.

28. (B)

The airport has dedicated one area or container for the specific needs of one destination.
This is *encapsulation*, where the data and all the methods processing the data are together.

29. (B)

Method **ghana** computes the greatest common factor (GCF) of two numbers.
Method **botswana** computes the least common multiple (LCM) of two numbers with the help of the GCF.

30. (C)

This is an example of *aliasing*.
list2 starts with all the elements of **list1** and then adds four additional names.
However, both **list1** and **list2** store the same reference to the memory of the array elements.
The result is that **list1** displays the same contents as **list2**.

31. (A)

Question 31 involves both inheritance and composition relationships between classes.
The construction of an **Employee** object involves first constructing the super class object of **Person**.
During the construction of a **Person** object, an **Education** object is constructed as the last statement in the **Person** constructor.

32. (C)

Method **rearrange** traverses the entire matrix and checks for cells who **row** value equals **c** and whose **col**umn value equals **r**. These cells get reassigned with a value of **0**. If the matrix were to be displayed it would show a plus (+) sign of zeros in most cases. The examples below is for a 4 x 5 matrix where **r** = **1** and **c** = **2**. The left example is correct. The right example is a common pitfall if you misread the **if** statement. It says `if (row == c || col == r)`, NOT `if (row == r || col == c)`.

1	**0**	3	4	5
6	**0**	8	9	10
0	**0**	**0**	**0**	**0**
16	**0**	18	19	20

1	2	**0**	4	5
0	**0**	**0**	**0**	**0**
11	12	**0**	14	15
16	17	**0**	19	20

33. (A)

Math.random() generates a random **x**, such that **0 <= x < 1**.
The **(int)** casting turns any number generated by **Math.random** into **0.**
The multiplication times **10** occurs after the casting is already done.

34. (B)

This question checks the knowledge that a class can only extend one super class.
However a class can implement multiple interfaces.

35. (D)

The **toString** method is implemented so that each iteration adds an array element and a line-feed.
This means that all the elements display on a separate line.

36. (B)

The syntax **System.out** implies that **out** is a member of the **System** class.
out has no parentheses, which implies it is an attribute or *instance variable* and not a method.

37. (D)

To understand what is happening, look at the 3 matrixes below:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

1	5	9	13
5	6	10	14
9	10	11	15
13	14	15	16

The left matrix shows a sample of what a matrix might look like BEFORE the call to **reArrage** is made. A common pitfall when students see the statement `matrix[row][col] = matrix[col][row];` they erroneously that the rows and columns are being swapped and might think the resulting matrix would be the one in the middle. What actually happens is every row is *copied* from every column. This makes the matrix symmetrical and the resulting matrix would be the one on the right.

38. (C)

Method **ifElse** performs recursive calls before displaying the value of **n.**
The call **ifElse(4)** generates four recursive calls. Each call stores the value of **n**.
The stored values are displayed in a **LIFO** sequence of **1 2 3 4**.

39. (A)

During the construction of the **Flower** object the **q** parameter with value **5000** is properly passed to the constructor of the **Plant** super class with the **super(q);** statement.

In the constructor of the **Plant** class **q** is assigned to quantity.

The **toString** redefinition of the **Flower** class displays the output **5000 Tulips.**

40. (E)

The parameter of method **counter**, **n** is a *binary* number, like 2, 4, 8, 16, 32, etc.
Variable **q** is initialized to **2** and continuously multiplied by **2**.
The loop stops when **q** equals **n**.
The results is that method counter returns a number **v**, such that $\mathbf{2^v == n}$.

Section II, Free Response

Question 1.

Part (a).

```
public static double getMean(int[] list)
{
    int sum = 0;
    for (int k = 0; k < list.length; k++)
        sum += list[k];
    return (double) sum/list.length;
}
```

Part (b).

```
public static int getMode(int[] list)
{
    sortList(list);
    int mode = list[0];
    int modeCount = 1;
    int tempMode = mode;
    int tempCount = 1;
    for (int k = 1; k < list.length; k++)
    {
        if (tempMode == list[k])
        {
            tempCount++;
            if (tempCount > modeCount)
            {
                mode = tempMode;
                modeCount = tempCount;
            }
        }
        else
        {
                tempMode = list[k];
                tempCount = 1;
        }
    }
    return mode;
}
```

Question 2.

Part (a).

```
public int nextInteger(int n)
{
    double temp = getRandom();
    temp = temp * n;
    return (int) temp;
}
```

Part (b).

```
public Random2(double seed)
{
    super(seed);
}
```

Part (c).

```
public int nextInteger(int start, int end)
{
    int range = end - start + 1;
    int temp = nextInteger(range);
    return temp + start;
}
```

Question 3.
Part (a).

```
public String enCrypt(String source, String key)
{
    String codedString = "";
    int q = 0;
    for (int p = 0; p < source.length(); p++)
    {
        String sourceChar = source.substring(p,p+1);
        String keyChar =key.substring(q,q+1);
        q++;
        if (q == key.length())
            q = 0;
        codedString += getCode(sourceChar,keyChar);
    }
    return codedString;
}
```

Part (b).

```
public String deCrypt(String coded, String key)
{
    String sourceString = "";
    int q = 0;
    for (int p = 0; p < coded.length(); p++)
    {
        String codedChar = coded.substring(p,p+1);
        String keyChar =key.substring(q,q+1);
        q++;
        if (q == key.length())
            q = 0;
        sourceString += getSource(codedChar,keyChar);
    }
    return sourceString;
}
```

Question 4.

Part (a).

```
public String toString()
{
    String temp = "";
    if (num == 0)
        temp += 0;
    else
        if (den == 1)
            temp += num;
        else
            temp += num + "/" + den;
    return temp;
}
```

Part (b).

```
private int getGCF (int n, int d)
{
    int rem = n % d;
    if (rem == 0)
        return d;
    else
        return getGCF(d,rem);
}
```

Part (c).

```
public void add(Rational r1, Rational r2)
{
    int tempNum = r1.num * r2.den + r2.num * r1.den;
    int tempDen = r1.den * r2.den;
    int gcf = getGCF(tempNum,tempDen);
    num = tempNum / gcf;
    den = tempDen / gcf;
}
```

Chapter 18 Solutions
Sample AP Examination III

01. (B)

Method mystery has an outer loop counter **p** that repeats 7 times.
This means that the inner loop counts to 7 with counter **q** 7 times.
However, the inner loop counter **q** starts with 1, then 2, then 3, etc.
This means that the inner loop repeats **7 + 6 + 5 + 4 + 3 + 2 + 1 = 28** times.

Since **counter** starts at **0** and increments by one for each iteration of the **q** loop, **counter** returns **28**.

m	counter	p	q
8	0	1	1
	1		2
	2		3
	3		4
	4		5
	5		6
	6		7
	7	2	2
	8		3
	9		4
	10		5
	11		6
	12		7
	13	3	3
	14		4
	...	...	...

02. (B)

Boolean logic with *De Morgan's Law* states that **(!A || !B)** is equivalent to **!(A && B)**

This means that now the expression is **(A && B) && !(A && B)**, which is always **false**.

03. (C)

In method **Mystery** the statement **list[k] = list[k-1]** assigns the previous array element to the current array position.

However, this assignment only occurs when the array index value of **k** is even.
Consider the following table below and notice how every other value is shifted.

Index	0	1	2	3	4	5	6	7	8
Starting values	99	88	77	66	55	44	33	22	11
Shifting values	99	88	88	66	66	44	44	22	22

04. (E)

Assume that **p = 100** and **q = 200** to analyze this question.
The **Math.random()** statement generates a number in the **[0 . . 1)** range.
r = q - p + 1 which means that **r = 200 - 100 + 1 = 101**.
This means that every number generated by **Math.random()** is multiplied times **101**.
The range of numbers is now **[0 . . 101)**
The value of p, which is 100, is added to the numbers to get a range of [100 . .201).
Finally to cast to an **(int)** eliminates any fractions value to make the range **[100 . . 200]**.
This is precisely what happens with the variables **[p . . q]**.

05. (C)

Method **getIntCount** is mathematically correct.
The problem is that an **int** value is returned with a limited range of values.
Integer.MAX_VALUE equals 2,147,483,647.
Integer.MIN_VALUE equals -2,147,483,648.
The total range of integers is 4,294,967,296, which doubles the maximum **int** value.

06. (C)

The only way to make method **getIntCount** work correctly is to return a larger integer type.
In this case a **long** is returned and a cast to **long** is made.

Note: data type **long** is not in the AP subset for testing. However, it needs to be understood that an item, which is not in the AP subset, can be used in a question if such an item is explained in the question.

07. (B)

Implementation 1 is wrong; it does not cast to **Student**.
Implementation 2 is correct.
Implementation 3 is wrong; it dereferences a **double**.

08. (C)

numWidgets is a static variable and has global scope.
The value of **numWidgets** is not replaced by the scores of **w1**, **w2** and **w3**, because they are lower than **1000**.

09. (B)

The preconditions of both methods states that the argument is greater than zero.
This eliminates any concerns about dividing by zero.

Both **temp1** and **temp2** make the identical **italy(x)** method call.
Regardless of the **italy(x)** value, the value of **temp1/temp2** will always be **1**.

10. (D)

All three implementations check if **obj** is already contained in the **MyList** object.
If there is no duplicate element all three implementation call the **super.add** method.

Implementation 1 is correct using **this.contains(obj)**.
Implementation 2 is correct using **contains(obj)** which implies **this**.
Implementation 3 is incorrect. **contains** is treated like a class method with **MyList.contains(obj)**.

11. (D)

This question involves composition relationships.
The **House** constructor is called with the statement **House house = new House();**
The sequence of the three statements in the **House** constructor dictate the output.

12. (A)

Question 12 has a set of nested loops.
The statements in the inner loop swap elements in a **temp** array.
Note that the **q** index, which is the column index is always **q**.
The **p** element uses **p** and **p-1** for swapping, which swaps row values.
With each completion of the inner loop a set of rows are swapped.
This results in the top row being swapped repeatedly until it finishes at the bottom position.

13. (C)

The **Math.random()** statement generates a number in the **[0 . . 1)** range.
Every number generated by **Math.random()** is multiplied times **900**.
The range of numbers is now **[0 . . 900)**
A cast to an **(int)** eliminates any fractions value to make the range **[0 . . 899]**.

The value of **k** is added to **[0 . . 899]** ten times and **k** changes from **1** to **10**.

This means that first loop generates numbers in the **[1 . . 900]** range.
The last loop iteration generates numbers in the **[10 . . 909]** range.
The result of ten loop will then generate numbers in the **[1 . . 909]** range.

14. (A)

The outer loop of the program segment visits every element of the **list1** array.
The inner loop compares each element of **list1** with elements of **list2**.
If the element in **list1** does not appear in **list2**, then it is added to **list2**.
This means that all elements from **list1** are added to **list2**, if they are not duplicate values.

15. (C)

This question presents a *bubble sort* in disguise. Normally, a traditional **for** loop is used for the outer loop, which is used to reduce the comparisons of the inner loop.
With the **for..each** loop the *bubble sort* makes many unnecessary comparisons, but the **names** array is still sorted correctly in ascending order.

16. (B)

All three choice are features of program reliability.
(I) describes encapsulation.
(II) describes polymorphism.
(III) describes inheritance.

17. (A)

The future airport keep the data (passengers and luggage) contained in the same area as anything that processes the passengers and luggage, which is encapsulation.

18. (A)

There are many conditions in method **ifElse** and none apply with parameter **-5**.
This means that only the final **else** applies, which displaus the parameter **-5**.

19. (E)

The **toString** method of the **Apple** class is redefined to first call the **toString** method of the super class, which returns the **quantity**. This is followed by the **name** of the **Fruit** object.

20. (C)

Method **monkey** takes the **nr** argument processes a loop that repeatedly divides **nr** by **3**. The remainder of each division is added to **temp**. This is the algorithm to convert a number from base 10 to any base, and in this case it becomes a base-3 number.

21. (A)

Attribute **highScore** needs to store the highest scores from multiple instances of the **Game** objects. This process requires a static variable.

Declaration I is correct by declaring a static variable.
Declaration II is wrong; the instance variable is not static.
Declaration III is wrong; the instance variable is not static

22. (B)

A method that alters a static variable must be a public static method.

Declaration I is wrong; the method is declared private.
Declaration II is correct; the method is declared public and static.
Declaration III is wrong; the method is not declared static.

23. (B)

The program segment starts with a loop that that adds 15 random integers to **numbers1**.

The second loop checks each integer in **numbers1**. If the number is odd, then the number is added to the **numbers2** array.

24. (A)

This question requires careful scrutiny.
The statements are not complicated, but they can be misleading.
Method **sillyString** returns **temp2 + temp1**, which can give the impression that it is **string1 + string2**.

However, **temp1** stores a **s2** substring and **temp2** stores a **s1** substring.
This reverses the strings and the result is a return of **string1 + string2**.

25. (A)

Follow the statements of method **jack** in sequence.

seed = seed + Math.PI;	generates some random real number with a fractional portion.
seed = Math.sqrt(seed);	alters **seed**, but it is still a real number.
(int) seed;	cast a real number to an integer; the fraction portion is removed.
seed = seed - (int) seed	subtracts the integer number from the real number, leaving a fractrion.

26. (C)

Consider the call **canada(8,3)** to analyze the behavior of the **canada** method.

recursive call	p > q	p	q
canada (8,3)	true	7	4
canada (7,4)	true	6	5
canada (6,5)	true	5	6
canada (5,6)	false		

Method **canada** returns **5 + 6 = 11** in this example, which is the same as **8 + 3 = 11**.

The result will always be **a + b** because one argument is increased and the second one is decreased, which always leaves the same sum of the original arguments.

27. (B)

Interface **XYZ** is correct and only contains abstract methods. It is not necessary to use the reserved word **abstract** in an interface. All members of an interface are assumed to be abstract.

Abstract class **PQR** is correct and implements some abstract methods. It is not required that all the methods of the interface are implemented as long as the class is an abstract class.

Concrete class **ABC** is not correct. Since this method is not abstract, it is required to implement any abstract methods that have not yet been implemented. Method **getData** was never implemented.

28. (C)

Implementation 1 is correct by using the **add** method, which allocates space for the **ArrayList** object.

Implementation 2 is correct by using the **add** method, which allocates space for the **ArrayList** object.

Implementation 3 is wrong.by using the **set** method. The **set** method does not allocate new memory space. Method **set** can only replace a value of an element at a specified location.

29. (D)

Class **Eye** is neither a superclass nor a subclass. Inheritance is not used.

Class **Person** uses an **Eye** object as a private attribute, which is composition.
A person has-an eye.

Class **Student** extends **Person**, which makes **Student** a subclass or the superclass **Person.**
A student is-a person.

30. (C)

Use the rules of mathematical precedence.
First divide 1 / 4, which is integer division and becomes 0.
Add **4 + 3 + 2 = 9**
Cast integer **9** to a double to end with **9.0**

A result of **9.25** would have resulted if the statement was changed to

```
double result = 4 + 3 + 2 + (double) 1/4;
```

31. (A)

This is a question about *short-circuiting*.

The call to **method1(9)** return **true**.
This means that the call to **method(9)** is ignored, since it is known that the entire condition is **true**.

The second time in the loop **method1(10)** is called, which returns **false**.
The result is that **method2(10)** is called.

32. (B)

It is always **true** that **bool equals true or bool equals false**.
The negetation of the previous statement makes the expession always false.

33. (B)

. Method **swing** is a sort routine.
When the statement **(list[q] < list[q+1])** is **true**, the values are swapped.
This result in an descending array order.

34. (A)

When a requested element is not present, the binary search cannot positively state that the element does not exist. Areas that should be searched may be omitted.

All remaining statements are true.

35. (C)

The functionality may be identical, but the program will be different.
An interface must be implemented.
An abstract class must be extended.

36. (C)

Normally, a linear search requires that every elment is compared before concluding that a search item is not present in a search list.

When the search list is sorted, the search can stop when the linear search is past the point where it is possible for the search item to exist.

Both implementation 1 and implementation 2 do not check if the search is past the possible point.
Only implementation 3 check for this possibility.

37. (C)

The only time that the row **[r]** index and the col **[c]** index are equal is on the *top-left to bottom-right* diagonal of the matrix.

Since the matrix values are only printed when **if (r == c) is true** and **00** otherwise,
answer **(C)** is correct.

38. (A)

The Boolean expression **A and B** is always **false** when **A equals false**.

The Boolean expression **A and B** is always **false** when **A equals false**.

It is not necessary to consider the second Boolean operand in the example above.
This is known as *short circuiting*.

39. (D)

At the start of the loop structure the **names** array holds values **[John, Greg, Maria, Heidi]**

The loop is repeated three times and each time there are two changes.

1st Iteration:	Greg is set into John's position leaving The second statement brings no change	**[Greg, Greg, Maria, Heidi]**
2nd Iteration:	Maria is set into Greg's position leaving The second statement brings no change	**[Greg, Maria, Maria, Heidi]**
3rd Iteration:	Heidi is set into Maria's position leaving The second statement brings no change	**[Greg, Maria, Heidi, Heidi]**

40. (D)

Method **panda** is a simple *linear search* method.
The problem is that the method continues to check each item, even after **number** is found.

This means that **found** is changed to **false** immediately after **number** is found.
This method only works correctly if **number** is the last element or **number** does not exist.

Section II, Free Response

Question 1.

Part (a).

```
public static String fromHexToBin(String hexNum)
{
        String binNum = "";
        for (int k = 0; k < hexNum.length(); k++)
        {
            String temp = hexNum.substring(k,k+1);
            if (temp.equals("0")) binNum += "0000";
            else if (temp.equals("1")) binNum += "0001";
            else if (temp.equals("2")) binNum += "0010";
            else if (temp.equals("3")) binNum += "0011";
            else if (temp.equals("4")) binNum += "0100";
            else if (temp.equals("5")) binNum += "0101";
            else if (temp.equals("6")) binNum += "0110";
            else if (temp.equals("7")) binNum += "0111";
            else if (temp.equals("8")) binNum += "1000";
            else if (temp.equals("9")) binNum += "1001";
            else if (temp.equals("A")) binNum += "1010";
            else if (temp.equals("B")) binNum += "1011";
            else if (temp.equals("C")) binNum += "1100";
            else if (temp.equals("D")) binNum += "1101";
            else if (temp.equals("E")) binNum += "1110";
            else if (temp.equals("F")) binNum += "1111";
        }
        return binNum;
    }
```

Part (b).

```
public static int fromBinToDec(String binNum)
{
    int decNum = 0;
    int pow2 = 1;
    for (int j = binNum.length()-1; j >= 0; j--)
    {
        if (binNum.charAt(j) == '1')
                decNum += pow2;
        pow2 *= 2;
    }
    return decNum;
}
```

Part (c).

```
public static int fromHexToDec(String hexNum)
    {
    String binNumber = fromHexToBin(hexNum);
    int decNum = fromBinToDec(binNumber);
        return decNum;
    }
```

Question 2.

Part (a).

```
public int linearSearch(int sn)
{
    boolean found = false;
    int k = 0;
    while (k < size && !found)
    {
        if (intArray[k] == sn)
            found = true;
        else
            k++;
    }
    if (found)
        return k;
    else
        return -1;
}
```

Part (b).

```
public boolean isSorted()
{
    for (int q = 0; q < size-1; q++)
        if (intArray[q] > intArray[q+1])
            sorted = false;
    return sorted;
}
```

Part (c).

```
public int binarySearch(int sn)
{
    boolean found = false;
    int lo = 0;
    int hi = size-1;
    int mid = 0;
    while (lo <= hi && !found)
    {
        mid = (lo + hi) / 2;
        if (intArray[mid] == sn)
            found = true;
        else
        {
            if (sn > intArray[mid])
                lo = mid + 1;
            else
                hi = mid - 1;
        }
    }
    if (found)
        return mid;
    else
        return -1;
}
```

Question 3.

Part (a).

```
public void add(String str)
{
    strings.add(str);
    size++;
}
```

Part (b).

```
private void sortStrings()
{
    for (int p = 1; p < size; p++)
        for (int q = 0; q < size-p; q++)
        {
            if ( ((strings.get(q)).compareTo(strings.get(q+1))) > 0)
            {
                String temp = strings.get(q);
                strings.set(q,strings.get(q+1));
                strings.set(q+1,temp);
            }
        }
}
```

Part (c).

```
public String toString()
{
    sortStrings();
    String temp = "[";
    for (int k = 0; k < size-1; k++)
        temp = temp + strings.get(k) + ", ";
    temp = temp + strings.get(size-1) + "]";
    return temp;
}
```

Question 4.

Part (a).

```
public String toString()
{
    String temp = "";
    for (int row = 0; row < numRows; row++)
    {
        for (int col = 0; col < numCols; col++)
            temp += twoD[row][col] + "  ";
        temp += "\n";
    }
    return temp;
}
```

Part (b).

```
private boolean okSize(Matrix m1, Matrix m2)
{
    return m1.numCols == m2.numRows && numRows == m1.numRows && numCols == m2.numCols;
}
```

Part (c).

```
public void multiply(Matrix m1, Matrix m2)
{
   if (okSize(m1,m2))
   {
      for (int r = 0; r < numRows; r++)
      {
         for (int c = 0; c < numCols; c++)
         {
            int temp = 0;
            for (int k = 0; k < m1.numCols; k++)
               temp += m1.twoD[r][k] * m2.twoD[k][c];
            twoD[r][c] = temp;
         }
      }
   }
}
```