

**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 # 978-1-934780-06-0

Copyright © 2010 by D&S Marketing Systems, Inc.

All rights reserved.

No part of this book may be represented or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, or by any information storage or retrieval systems, without written permission from the publisher.

Printed in the U.S.A.

Preface

This book has been prepared with one purpose in mind: to help students to prepare for the Advanced Placement Computer Science Examination. It should be made clear from the start that students will have to respond on the AP Examination to both multiple-choice and free-response type questions. Our focus here is on the multiple-choice area, a very important one. Students of Computer Science frequently get the kind of practice they need in developing algorithms and writing programs through programming assignments during the course of the year. But all too often there just is not the amount of time teachers would like to be able to spend practicing multiple-choice type questions. This book seeks to serve that need better. There are 345 multiple-choice questions. Additionally, there are four free-response questions with each practice examination for a total of 357 questions in the seventh edition.

The questions which you will find within, have been specifically chosen to give students a review experience over a wide range of topics covered on the AP Computer Science Examination. The difficulty level of the questions aims to be as varied and challenging as those on the exam itself, and at times intentionally more challenging. Continuing with this edition are groups of questions in the same topic area that will help teachers and students identify strengths and weaknesses in AP Computer Science Examination preparation. New to the seventh edition is the removal of "AB-Level" questions that will no longer be tested and the inclusion of topics that are part of the new AP Computer Science curriculum starting with the 2010 AP Computer Science Examination.

The seventh edition consists of two major parts. The first part contains 15 individual chapters with fifteen questions in each chapter on the same topic. The second part consists of three sample AP Computer Science Examinations. Each examination is divided into two sections. Section I contains 40 multiple choice questions and Section II continues with four free response questions.

For the teacher of Advanced Placement Computer Science, it is hoped that this book will provide you and your students with additional practice materials in an area where there is not as much available as we might like. For the student of computer science, I hope that this book might in some small way, help you achieve continued success in an exciting and important field.

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

Foreword to the 7th Edition

The creation of the seventh edition presented the same challenges that were present during the writing of the previous editions. All editions are created with a close eye on the course description for AP Computer Science. Additionally, review books of this type rely heavily on a growing number of released examinations, which provide information about question style, topic emphasis, and question difficulty level.

The most important requirement for review materials, in preparation for any test, is that the questions cover the range of topics that will be tested, so that strength and weaknesses in test preparation can be identified. A second test preparation requirement is to gain familiarity with question style. This second requirement used to be easily satisfied with the aid of released examinations. For the last three editions, the issue of question style is more challenging. The switch to Java 5.0 has not altered a certain generic style of questioning that is language independent. On the other hand, the switch to Java 5.0 has brought the inclusion of many new Java language features that are not yet available on any released examination.

In creating this seventh edition I have relied heavily on my experience with the AP Test Development Committee in the past, the test philosophy of the College Board, the AP Course Description in Computer Science for May, 2010 and May 2011, workshops I have attended in preparation for the Java change and communication with many teachers who are active with the College Board and AP Computer Science.

The seventh edition would not be possible without the tremendous help of my son, John Schram, who now has 19 years of computer science teaching experience. He contributed many new questions and assisted with many, many hours of proofreading, editing and suggestions for completion the latest edition.

The AP Computer Science Examination includes questions on a case study. Since the case study topic changes every two or three years, any inclusion of case study questions would make this edition rapidly obsolete. Information about the current case study can be obtained from the College Board.

Advanced Placement teachers face a classic *no-win situation* as they prepare students for an AP examination. A teacher can give students a sample examination that is very rigorous and hope that students will be relieved that the real exam is not as difficult as the practice test. The problem with this approach is that some students will be discouraged and select not to take the examination. A teacher can also give a relatively easy examination with the aim to motivate students and give them a sense of success. This approach also has a problem since it can result in a serious *cold shower* during the real AP examination.

Many teachers prefer a compromise and like to give a practice examination that is neither too hard nor too easy and represents the difficulty level of the actual examination. Such is a lofty goal that is easily mentioned but difficult to attain. Multiple choice examinations are not released on a yearly basis, and the most recently released multiple choice computer science examination was given in a previous Java language version.

The main purpose of the four sample examinations in this book is to help prepare students for the real thing. I have decided to fault on the side of difficulty. Students often find many questions on the AP examination that are long and time-consuming to read. Normally, students have not received many *full-page* questions during the school year. The three sample examinations in this review book have a heavy dose of long questions. The long questions include many preconditions and postconditions that are time-consuming to read, but will also show throughout the actual AP Computer Science Examination.

Students must budget their time wisely as they take difficult, time-controlled tests. The sample examinations in this book intentionally contain many time-consuming questions to alert students to the reality of time budgeting. There are a variety of questions that involve selecting the correct solution for some incomplete method. These *complete-the-method* questions are intended not only to prepare students for the multiple choice part of the examination, but also for the free response section. I have noticed with my own students that the biggest problem with the long questions is that students skip them. Students think these questions are too long and too complicated. In reality, the long *complete-the-method* style questions are quite manageable, and experience has shown that students handle such questions very well with sufficient exposure. The spirit of the sample examination is to give students sufficient exposure and prepare them to succeed on the real test. I hope these questions will satisfy that goal.

Leon Schram
Royse City, Texas

Table of Contents

01. Control Structures	1
02. Methods and Parameters	16
03. Boolean Algebra	35
04. Object Oriented Programming	56
05. Static Java 1D Arrays	77
06. Static Java 2D Arrays	91
07. The String Class	107
08. Number Systems	121
09. The ArrayList Class.	130
10. Recursion.	146
11. Implementing Methods and Redefining Methods	162
12. Java Standard Libraries	185
13. Algorithms and Informal Algorithmic Analysis	170
14. Polymorphism	196
15. Program Design.	211
15. The Java Collection Hierarchy.	231
16. Sample Examination I.	241
17. Sample Examination II	278
18. Sample Examination III	316

Chapter 1 Review

Control Structures

Before the actual review of control structures starts, you need to understand clearly what to expect from this review and the reviews of future chapters. This book is designed to prepare you for the upcoming AP Computer Science Examination, which will be abbreviated as APCS Exam from this point forward. This manual is not a substitute for a text book, which introduces and explains each one of the computer science topics. Each review section consists of three parts.

First, there is a review of the significant points that you must clearly understand before you approach any one of the questions. Keep in mind that the topics are presented not as a first introduction, but as a review, which assumes that you have seen this information earlier. If in fact you come to a chapter where the review makes it very clear that you have never studied the given topic, then you must seek help from your teacher, a text book or whichever resource can provide you with a good first introduction. Computer Science topics that have proven to present few understanding problems, like control structures, provide very little review. Topics, which have a record of poor test performance will be presented in far greater detail.

*Second, you will receive information that will be tested on the APCS Exam. The College Board has established an AP Subset, which is a set of topics that may be tested. This preparation manual only focuses on those topics that will be tested and the questions will likewise only focus on those topics. For example, the Java post-condition **do...while** loop is not in the AP Subset. This means that do...while will neither be reviewed in this section, nor will there be sample questions on the post-condition loop in the question section. It is very possible that the do...while loop is in your textbook and you may have seen questions on this topic in a chapter test. However, this manual will strictly focus on those topics and sub-topics that are listed in the College Board Course Description.*

Third, you need to be aware how easily somebody with a clear understanding on a given topic can nevertheless answer questions incorrectly. This means you need to understand test-taking strategies by topic. Each topic has its own subtle little pitfalls that can be overlooked. Students, who are not aware of these problems, may feel good about their performance on the APCS Exam and later are quite unpleasantly surprised when an unexpectedly low score arrives in the mail.

If you are using this test preparation manual as part of the course curriculum, your teacher will specify what the class needs to do and when to do it. On the other hand, it is possible that you, and perhaps a small number of fellow students, are given this manual and told to study and prepare for the test. How should you proceed in such a scenario? I am now assuming that you have access to the solutions manual. It really is pretty much impossible to work on your own if you do not know the accuracy of your solutions.

I suggest the following approach. After the chapter review do the fifteen questions that follow and check your answers. It is very important that you do not guess on any questions. Leave a question blank if you are confused. It is possible that you answer a question correctly with lucky guessing. In such a case you may erroneously conclude that you understand the material. Therefore, it is very important that you carefully identify questions into several categories.

- You know the topic and you have the question correct.
- You know the topic, but you selected the wrong answer. When you see the answer you know that you made a careless error.
- You know the topic, but you selected the wrong answer. When you check the explanation of the solution you realize your mistake and you understand this type of question better.
- You thought you understood the question, but you have the wrong answer. The solution manual does not help to clarify the confusion. This means that you must seek help on this topic.
- You skipped the question and you know that you must seek help for this topic.

There are very few problems that test control structures directly on the AP CS Exam. The scope of topics in AP Computer Science guarantees that there cannot be many questions on any one topic. Control structures can also be overlooked as a fundamental topic, which does not compare in complexity to many other topics in computer science. This may all be true, but the simple reality is that control structures appear everywhere in any program and they will appear in many AP CS Exam questions. A question may focus on inheritance, sorting, searching, parameter passing, algorithmic analysis or other topics, but a quick examination of the program code includes one or more control structures. This can easily bring about the following problem.

APCS Examination Alert

Students frequently answer questions incorrectly about a variety of topics, even though they have solid knowledge on these topics. The wrong answer is selected due to handling control structures incorrectly.

The problem described here is analogous to mathematics. Students may have a good understanding of advanced mathematical concepts and yet answer many questions incorrectly because of arithmetic errors.

It was mentioned earlier that not all Java features are tested on the AP CS Exam. You will see evidence of this in this introductory chapter since not all control structures are tested. The rationale for inclusion or exclusion of topics is not a concern in this AP CS Exam review book. The AP Computer Science Test Development Committee makes decisions about the inclusion of topics and makes questions based on included topics. The focus of this review book will be strictly on those topics that are going to be tested. This pattern is already evident in this first chapter. The review questions presented do not use all of the Java primitive data types.

APCS Examination Alert

The College Board has published an **AP Java Subset** of Java features that may be tested on the AP Computer Science Examination. Whenever appropriate, you will receive alerts to those items that are tested.

A *simple* or *primitive* data type holds a single value. Java's primitive data types that are in the AP Subset are:

int double boolean

The three primary control structures are *simple sequence*, *selection* and *repetition*. Program logic, in any language, dictates that a program executes in the sequence that program statements are presented. Line after line the program is executed unless a selection structure or repetition structure causes the program flow to jump to another sequence of instructions. During program execution, conditional statements are encountered. These conditional statements will always be true or false, and they are used in selection control structures and repetition control structures. The Java syntax of the different control structures will be reviewed in various program examples that will be shown later in this chapter. Not all of the Java control structures will be tested.

AP Examination Alert

One-way selection with **if** and two-way selection with **if...else** are part of the AP subset.

Multi-way selection with **switch**, **case** and **break** is not part of the AP subset.

Multi-way selection logic will be tested with nested **if...else** statements.

Fixed iteration with **for** and pre-condition iteration with **while** are part of the AP subset.

Post-condition iteration with **do...while** is not part of the AP subset.

Many APCS Exam questions require determining the output of a program segment. Students are used to working on their computers. The compiler detects syntax errors, and the program execution displays the actual program output. It is a different matter when the only program available is printed on paper, and you are both the compiler and program executor. Playing computer requires that you keep track of the variable values as they change. You need to do a *variable trace*. Many students think that it is possible to perform variable tracing in their heads. Please take the time to write down the columns of variable values. The best students, with solid computer science knowledge, can easily lose track when there are multiple variable values, which change through a large variety of repetitions.

AP Examination Alert

There are no computers available during the APCS Exam. You must perform all program executions on paper. Scratch paper is not allowed during the examination. All writing, including the multiple choice segment, must be done on the test paper. It is vital to use *variable traces* whenever questions require the program segment output or variable values.

The program segment in **Example 1.1** shows a typical problem that requires tracing. The segment combines a loop control structure with a selection structure. Two variables change during program execution. Some students may have the mental concentration to handle this problem. The anxiety people feel on an important test makes it more difficult to keep track of multiple details. The program segment is followed by a *variable trace table*.

Example 1.1

```
int count = 0;
for (int k = 1; k < 5; k++)
{
    if (k % 2 == 0)
        count += 2;
    else
        count += 3;
}
System.out.println(count);
```

k	count	Output
	0	
1	3	
2	5	
3	8	
4	10	10

Control Structure Pitfalls

The review sections are placed at the start of each chapter before you tackle fifteen sample questions during each one of the topics questions. After concluding the topics, you can then test yourself with three sample APCS Exams. Some chapters present less review, like this chapter, because of its fundamental nature. It can be argued that any student who does not comprehend control structures may encounter serious difficulties during the challenging APCS Exam. On the other hand, even the brightest students may have difficulties with subtleties that are easily overlooked and result in selecting the wrong answer. Keep in mind that the committee who creates the APCS Exam intentionally includes *distracters* (wrong answers), which seem correct if the subtleties are ignored. Perhaps the biggest benefit of each review section is a summary of pitfalls that are common for the given topic.

The most common control structure error is the OBOB. OBOB means *Off By One Bug*. It may appear that OBOBs are only a problem with control structure questions. Since there are not that many control structure questions, you might argue that this is not a big deal. The reality is that control structures are found everywhere in a program. It does not matter what the focus of a certain topic might be, control structures are used. During the multiple choice segment of the APCS Exam, you will need to evaluate many types of methods and program segments. This allows many OBOB opportunities. Likewise, during the free response section of the APCS Exam, you need to write your own code. The code you write may be almost correct because you overlook a boundary case.

AP Examination Alert

An OBOB is an **Off By One Bug**.

OBOBs occur frequently because the program code may appear simple and may be considered unworthy of serious attention.

Failure to consider borderline conditions causes many OBOBs.

This review will conclude with a set of programs that demonstrate examples of OBOB situations. The first program segment in **Example1.2** is an example of program that determines how many times the loop structure executes. You need to determine the value of `count` by checking how many times the loop repeats. A quick look at the loop condition `k < 5` makes a surprising number of students conclude that the loop repeats five times. Keep in mind that a loop body executes only if the loop condition (`k < 5` in this case) is true. This means that the loop will not execute the fifth time. The next program segment, in **Example1.3** makes a small change and now the loop repeats five times.

Example 1.2

```
int count = 0;
for (int k = 1; k < 5; k++)
{
    count++;
}
System.out.println(count);
```

The program examples in **Example1.2** and **Example1.3** use almost identical loop control structures. In both cases the loop control variable (LCV) is initialized to **1**. In both cases the LCV increases by one after each execution of the loop body. The difference is the loop condition, which determines if the loop body will be executed. As long as the loop condition is true, the loop will be executed. Make sure you watch out for the `<=` operator. In this case it means that the segment in **Example1.3** will repeat once more than the segment in **Example1.2**.

Example 1.3

```
int count = 0;
for (int k = 1; k <= 5; k++)
{
    count++;
}
System.out.println(count);
```

The two previous examples are subtle enough to be missed by many students. The common mistake presented in **Example 1.2** does happen often enough, but it tends to be a mistake of students new to computer science or students who do not pay attention. Frequently, students who are very confident or perhaps they are impatient quickly look at a problem and pick the answer. The simpler the problem, the less concentration that is presented and the likelihood of picking the wrong answer increases.

Sometimes program output occurs during the loop execution. This presents a different type of OBOB. **Example1.4** starts the loop execution by incrementing count, followed by an output display. This results in an output display of 1 2 3 4 5. The loop repeats five times and five times the value of count is displayed.

Example 1.4

```
int count = 0;
for (int k = 1; k <= 5; k++)
{
    count++;
    System.out.print(count + " ");
}
```

The output in **Example1.5** is also displayed five times. The exact same loop structure is used, which repeats five times. There are five numbers that will be displayed, but they are not the same numbers. In this example the value of count is first displayed and then incremented. You now have a classic OBOB opportunity. The first value displayed will be 0 and the last value is 4. The complete output display is 0 1 2 3 4. You can be sure that any multiple choice question will include distracters such that the answer is *off by one*

Example 1.5

```
int count = 0;
for (int k = 1; k <= 5; k++)
{
    System.out.print(count + " ");
    count++;
}
```

You now come to the main chapter section. It is your turn to demonstrate your understanding. Look at each one of the fifteen questions that follow and create a variable trace. It is possible that these questions are assigned for homework. It is also possible that you are working independently preparing for the AP CS Exam. In either case, it is very simple to place the question code in an actual computer program to determine the correct output or value. If you do that, congratulations, you have proven your word processing skills of copy and paste. Please have the maturity to realize that getting the answer by any means should not be your goal. You need to realize what you know and what you do not know.

Chapter 1 Questions

Control Structures

01. Consider the following code segment.

```
int p = 2;

while (p < 7)
{
    int q = 0;
    while (q < 5)
    {
        p += q;
        q++;
        System.out.println(p + " " + q);
    }
}
```

What is the last output when the segment executes?

- (A) 12 5
- (B) 5 12
- (C) 8 4
- (D) 4 5
- (E) 3 4

02. Consider the following code segment.

```
int count = 2;

for (int p = 0; p < 5; p++)
{
    if (p % 2 != 0)
        count--;
    else
        count += 2;
}
System.out.println(count);
```

What is printed as a result of executing the code segment?

- (A) 8
- (B) 4
- (C) 3
- (D) 5
- (E) 6

03. Consider the following code segment.

```
int x = <some integer greater than zero> ;
int n = 0;

if (x < 1000)
{
    if (x > 1500)
        n = 200;
    else
        n = 300;
}
else
{
    if (x < 500)
        n = 400;
    else
        n = 300;
}
System.out.println(n);
```

What is printed as a result of executing the code segment?

- (A) Unknown without the value of x
- (B) 0
- (C) 200
- (D) 300
- (E) 400

04. Consider the following code segment.

```
int x = 0;
int y = <some integer value greater than 0> ;
while (x < y)
{
    if (x % 2 == 0)
        x += 2;
    else
        y -= 2;
}
System.out.println(x);
```

In the answers the original value of y is used.

What is output when the segment executes?

- (A) 0
- (B) 1
- (C) An integer value equal to $y / 2$
- (D) An integer value greater than or equal to y
- (E) An integer value greater than y

05. Consider the following two code segments.

```
int n = 9;
int x = <some integer value greater than zero> ;
int y = x;

// Segment-1
while (x <= n)
{
    x++;
    System.out.print(x + " ");
}

// Segment-2
for (int p = y; p < n; p++)
{
    y++;
    System.out.print(y + " ");
}
```

For which integer values of x will **Segment-1** and **Segment-2** have the same output?

- (A) Only whenever $x \geq 10$
- (B) Only when $x = 10$
- (C) Only whenever $1 < x < 10$
- (D) For no values of x
- (E) For all values of x

06. Consider the following code segment.

```
int p = 0;
int q = <some integer value greater than 0> ;

while (p < q)
{
    p += 2;
    q -= 2;
}
System.out.println(p + q);
```

In the answers the original values of p and q are used.

What is output when the segment executes?

- (A) 0
- (B) A value equal to $-q$
- (C) A value equal to q
- (D) A value equal to $q - 2$
- (E) A value equal to $q + 2$

07. Consider the following code segment.

```
int q = <some integer value greater than 0> ;
int p = <some integer value greater than q> ;
while (p < q)
{
    p++;
    while (p < q)
    {
        q--;
    }
}
System.out.println(p + " " + q);
```

In the answers the original values of p and q are used.

What kind of values are printed when the segment executes?

- (A) two positive integers, such that p equals q
- (B) two positive integers, which are the initial values of p and q
- (C) two positive integers, such that p is greater than q
- (D) two positive integers, such that p is lesser than q
- (E) two positive integers, such that p equals q + 1

08. Consider the following code segment.

The code segment is meant to execute in such a manner that grades are displayed as follows:

0	-	74	F
75	-	83	C
84	-	92	B
93	-	100	A

```
int average = <int value such that 0 <= average <= 100> ;
if (average > 92)
    System.out.println("Your grade is A");
else if (average > 83)
    System.out.println("Your grade is B");
else if (average > 74)
    System.out.println("Your grade is C");
else
    System.out.println("Your grade is F");
```

For which values of average does the code segment execute correctly?

- (A) For all values in the [0 .. 100] range
- (B) Only for values of average < 75
- (C) Only for values of average < 84
- (D) Only for values of average < 93
- (E) There are no values of average that will execute properly.

09. Consider the following code segment.

The code segment is meant to execute in such a manner that grades are displayed as follows:

0	-	69	F
70	-	79	C
80	-	89	B
90	-	100	A

```
int average = <int value such that 0 <= average <= 100> ;
if (average < 100)
    System.out.println("Your grade is A");
else if (average < 90)
    System.out.println("Your grade is B");
else if (average < 80)
    System.out.println("Your grade is C");
else if (average < 70)
    System.out.println("Your grade is F");
```

For which values of average does the code segment execute correctly?

- (A) For all values in the proper [0..100] range
- (B) Only for values in the [0..69] range
- (C) Only for values in the [70..100] range
- (D) Only for values in the [90..100] range
- (E) Only for values in the [90..99] range

10. Consider the following two code segments.

```
int n = 9;
int x = <some integer value greater than zero> ;
int y = 10;
```

Segment-1

```
while (x < n)
{
    x++;
    System.out.print(x + " ");
}
System.out.println();
```

Segment-2

```
for (int p = y; p < n; p++)
{
    y++;
    System.out.print(y + " ");
}
```

For which integer values of x will **Segment-1** and **Segment-2** have the same output?

- (A) Only whenever $x \geq 9$
- (B) Only when $x == 10$
- (C) Only whenever $1 < x < 10$
- (D) Only whenever $1 \leq x \leq 10$
- (E) For all values of x

11. Consider the following two code segments.

```
int n = <some integer> ;
```

Segment-1

```
String s1 = "#";
for (int k = 1; k < n; k++)
    s1 += "#";
System.out.println(s1);
```

Segment-2

```
String s2 = "#";
for (int k = n; k > 0; k--)
    s2 += "#";
System.out.println(s2);
```

For which integer values of n will **Segment-1** and **Segment-2** have the same output?

- (A) Only whenever $n > 0$
- (B) Only whenever $n \geq 0$
- (C) Only whenever $n < 0$
- (D) Only whenever $n < 1$
- (E) For all values of n

12. Consider the following code segment.

```
int n = <some integer greater than zero> ;
int count = 0;
for (int p = 0; p < n; p++)
    for (int q = 1; q < n; q++)
        count++;
System.out.println(count);
```

What is the value of $count$ when the code segment finishes executing?

- (A) n^2
- (B) $(n - 1)^2$
- (C) $(n - 2)^2$
- (D) $p * (q - 1)$
- (E) $n * (n - 1)$

13. Consider the following code segment.

```
int n = <some integer greater than zero> ;
if ( n % 2 == 0 && n % 3 == 0)
    System.out.println("Hello");
```

For what values of n will this code segment generate output?

- (A) For all values of n
- (B) For all even values of n
- (C) For all values of n that are multiples of 3
- (D) For all values of n that are multiples of 6
- (E) For all values of n that is either a multiple of 2 or a multiple of 3.

14. Consider the following code segment.

```
int n = <some integer greater than zero> ;
if ( !(n > 50 && n < 100) )
    System.out.println("Hello");
```

For what values of n will this code segment generate output?

- (A) For all values of n
- (B) For all values of n in the range of [50..100]
- (C) For all values of n in the range of [51..99]
- (D) For all values of n in NOT the range of [50..100]
- (E) For all values of n in NOT the range of [51..99]

15. Consider the following code segment.

```
int n = <some integer greater than zero> ;
if (n >= 100 || n <= 100)
    System.out.println("Hello");
```

For what values of n will this code segment generate output?

- (A) For all values of n
- (B) For all values of n which are 100 or greater
- (C) For all values of n which are 100 or lesser
- (D) For all values of n NOT equal to 100
- (E) For n == 100 only

Chapter 2 Review

Methods and Parameters

Reviewing methods presents a *chicken or egg* problem. Methods are members of a class and a class has members that include methods. Should methods be presented in the context of an Object Oriented Programming (**OOP**) review, which includes classes and methods? In a later chapter, Object Oriented Programming will be reviewed, and at that time classes and their members will get closer scrutiny. At this stage, methods and parameters will be considered without the bigger OOP picture. With a better understanding of the building blocks used by classes, it will be easier to handle questions about classes and program design.

A method is nothing more than a group of program statements placed between a set of opening and closing braces. The method has a heading with a name that is used to execute the program statements in the method body. Up to this point it can be argued that working with methods is pretty simple if you understand the control structures that are likely used inside the method body. The essence of understanding methods depends on clearly comprehending how information is passed to and from methods.

Methods can be categorized in various types. Most fundamentally, especially for information passing, is the distinction between a *void* method and a *return* method. Method **sum** in **Example 2.1** is a return method. Method **sum** computes and then returns the sum of the two calling or actual parameters, **n1** and **n2**. The method heading parameter or formal parameter **x** receives the value **100**, which is passed by method call parameter **n1** and likewise method heading parameter **y** receives the value **200**, which is passed by method call parameter **n2**. It is easy to identify a *return* method. The conspicuous keyword *return* takes care of passing information back to the "calling" program statement, which in this case will display **300**. Note that variable **result** is an **int** and that is also the return type of method **sum**.

Example 2.1

```
public static void main(String args[])
{
    int n1 = 100;
    int n2 = 200;
    System.out.println(sum(n1,n2));
}

public static int sum(int x, int y)
{
    int result = x + y;
    return result;
}
```

Java makes method distinctions easy. Return methods include the keyword **return** and void methods include the keyword **void**. In **Example 2.2**, method **displayMax** compares parameters **x** and **y** for equality and then displays the result. There is no value returned, and in this example the parameter information is used to display the result of some desired process. The return type of a return method is required in the heading. If the method does not return a value, the return type is not necessary and becomes **void**, which is placed in the same location where the data type normally is indicated.

Example 2.2

```
public static void main(String args[])
{
    int n1 = 100;
    int n2 = 200;
    displayMax(n1,n2);
}

public static void displayMax(int x, int y)
{
    if (x == y)
        System.out.println(x + " equals " + y);
    else if (x > y)
        System.out.println(x + " > " + y);
    else
        System.out.println(x + " < " + y);
}
```

Calling Return Methods and Void Methods

A **return** method is called by using the *returned value* in a program statement. This can be output, assignment or comparison, like ...

```
System.out.println(computeSum(n1,n2));
int result = computeSum(n1,n2);
if (computeSum(n1,n2) > 100)
```

A **void** method call becomes a *stand-alone* program statement, like ...

```
displaySum(n1,n2);
centerString(title);
```

Methods and Parameters Pitfalls

It is easy to make mistakes in determining the values processed by a method. Such mistakes are not method errors. They are control structure errors, which were discussed in the previous section. There are problems, which are unique to method handling. First, it is important to remember the fundamental parameter rules.

Fundamental Rules About Using Methods With Parameters

The number of parameters in the method call must match the number of parameters in the method heading.

The corresponding actual parameters in the method call must be the same type as the formal parameters in the heading.

The sequence of the actual parameters in the method call must match the sequence of the parameters in the heading.

The actual parameter identifiers in the method call may be the same identifiers or different identifiers as the formal parameters in the heading.

There exists a subtle misunderstanding about parameter passing that can have serious logic error consequences in a program. Primitive data types, like **int**, **double** and **boolean**, store actual values. Objects store references to memory locations. Methods use both primitive data types and objects for parameter passing. This means that passing primitive data type values and object values seem to be different. The problem is that there is both a similarity and a difference, which is frequently misunderstood.

The problem at hand applies to void methods. What happens to the calling parameter values? Return methods do not share this confusion. With return methods, parameters provide necessary values for the processing of the return method and the requested result is returned.

Consider a **swap** method. Two parameters are passed, and the goal is to exchange the parameter values. A return method can only return a single value and is no help for a swapping routine. Perhaps a void method will do the job. The program in **Example 2.3** passes two integer values to method **swap** where the parameter values are exchanged. The variable values are displayed before the method call, inside the method body and after the method call.

Example 2.3

```
public class Example23
{
    public static void main(String args[])
    {
        int x = 10;
        int y = 20;
        System.out.println(x + " " + y);
        swap(x,y);
        System.out.println(x + " " + y);
    }

    public static void swap(int p, int q)
    {
        int temp = p;
        p = q;
        q = temp;
        System.out.println(p + " " + q);
    }
}
```

Example23 Output

```
10 20
20 10
10 20
```

The program output displays the original **10 20** values in the **main** method before the **swap** method call. The output statement inside the **swap** body proves that the parameter values are swapped. However, when the **main** method variables are displayed after the **swap** call, there is no change in the values. The **swap** method in **Example 2.3** is proof that the values of primitive data type variables cannot be changed by a method.

The logic of the swapping or lack of swapping, is caused by the fact that the **main** method variables **x** and **y** each have memory locations, which are separate from method variables. The values of these memory locations are passed to the **swap** method and copied to **swap** parameters **p** and **q**. Both **p** and **q** have memory locations different from **x** and **y**. When method **swap** exchanges values in **p** and **q**, it has no impact on **x** and **y** values.

In **Example 2.4** the **swap** method still exchanges values, but these values are now objects. For simplicity they are objects of the **String** class. The output results are identical to the previous example with integers. Values appear only to be exchanged inside the **swap** method. There exists a simple rule in Java, *actual parameters cannot be altered*. Actual parameters are the parameters in the calling method and they are fixed. The value of the actual parameters is passed to the formal parameters in the method heading. In the case of object parameters, references are passed, but back in the actual parameter's location those references will not change.

Example 2.4

```
public class Example24
{
    public static void main(String args[])
    {
        String name1 = "Tom";
        String name2 = "Sue";
        System.out.println(name1 + " " + name2);
        swap(name1, name2);
        System.out.println(name1 + " " + name2);
    }

    public static void swap(String s1, String s2)
    {
        String temp = s1;
        s1 = s2;
        s2 = temp;
        System.out.println(s1 + " " + s2);
    }
}
```

Example24 Output

```
Tom  Sue
Sue  Tom
Tom  Sue
```

At first glance it may appear that it does not matter what the parameters are. It seems that both primitive data values and object values behave in the same manner. Look at the method body and changes that occur with the copies of the actual parameters. Now return to the calling program segment, and the values of the actual parameters remain unchanged.

The program in **Example 2.5** demonstrates that it is possible to alter the values stored in an actual parameter. This is accomplished by changing method **swap** drastically. The earlier **swap** method stated something like: *temp = object1; object1 = object2; object2 = temp;* That approach does not touch the actual values that are stored. It only swaps the immediate values, which are stored, and those are memory addresses.

Now in the method of **Example 2.5**, the **swap** routine digs deeper and goes straight to the attributes of each object. Two exchanges are performed. First the name attributes are swapped, and then the age attributes follow. When method **swap** is finished, the actual parameters of the **Student** objects are not altered, even if that were possible. However the values, which are stored at the locations referenced by the objects certainly have been altered.

Example 2.5

```
public class Example25
{
    public static void main(String args[])
    {
        Student student1 = new Student("Tom",16);
        Student student2 = new Student("Sue",15);
        student1.showData(); student2.showData();
        System.out.println("\n\n");
        swap(student1,student2);
        student1.showData(); student2.showData();
        System.out.println("\n\n");
    }

    public static void swap(Student s1, Student s2)
    {
        int tempAge = s1.getAge();
        s1.setAge(s2.getAge());
        s2.setAge(tempAge);
        String tempName = s1.getName();
        s1.setName(s2.getName());
        s2.setName(tempName);
    }
}

class Student
{
    private String name;
    private int age;

    public Student (String n, int a)
    {
        name = n;
        age = a;
    }

    public int getAge()          { return age; }
    public String getName()      { return name; }
    public void setAge(int a)    { age = a; }
    public void setName(String n){ name = n; }

    public void showData()
    {
        System.out.print(name + " " + age + " ");
    }
}
```

Example25 Output

```
Tom 16    Sue 15
Sue 15    Tom 16
```

Java Parameter Notes

Java passes information to a method with parameters.

A copy of the calling, actual parameters are assigned to the receiving, formal parameters in the method.

In the case of a simple/primitive data type, a copy of the variable's value, like 23, 'A', or true, is sent by the parameter. Any changes are changes to a copy, which makes no changes to the original.

The same thing actually happens in the case of an object; however, with an object the value is the memory address of the object itself. If this is changed, it does not alter the memory address of the original object.

The values of actual parameters cannot be altered.

The only way changes made in a method impact the original calling object, is if changes are made to the attributes of the object.

A Note For Students/Teachers With C++ Knowledge

In C++ parameters can be *passed by value* or *passed by reference*.

In C++ when parameters are *passed by value*, a copy is made, and any change to the copy will not alter the value of the actual parameter.

In C++ when parameters are *passed by reference*, both the actual parameter variable and the formal parameter variable share the same memory location. Passing the reference of the actual parameter indicates where the values are stored. Any change to the formal parameter will change the actual parameter.

In Java parameters are only *passed by value*. It is true that objects store references, but the value of the reference is passed, and variables do not share the same memory location. The confusion comes from the fact that Java uses references for objects, even though parameters are passed by value only.

Chapter 2 Questions

Methods and Parameters

01. Consider the following method.

```
public static int method0201(int p)
{
    int count = 1;
    for (int q = p; q < 2*p; q++)
        count += count;
    return count;
}
```

What value is returned as a result of the call `method0201(n)`, if $n > 0$?

- (A) n
- (B) $2n$
- (C) n^2
- (D) 2^n
- (E) $2^n - 1$

02. Consider the following method.

```
public static int method0202(int p)
{
    int count = 0;
    for (int q = 1; q < 2*p; q++)
        count += count;
    return count;
}
```

What value is returned as a result of the call `method0202(10)`?

- (A) 0
- (B) 8192
- (C) 32768
- (D) 131072
- (E) 524288

03. Consider the following program.

```
public class Question0203
{
    public static void main(String args[])
    {
        for (int n = 1; n <= 10; n++)
            System.out.print(method0203(n) + "   ");
    }

    public static int method0203(int n)
    {
        int temp = 0;
        for (int p = 0; p < n; p++)
            for (int q = 0; q < n-p; q++)
                temp++;
        return temp;
    }
}
```

What is printed as a result of executing the program?

- (A) 0 2 5 9 14 20 27 35 44 54
- (B) 1 3 6 10 15 21 28 36 45 55
- (C) 2 4 8 16 32 64 128 256 512 1024
- (D) 3 6 12 24 48 96 192 384 768 536
- (E) 0 1 1 2 3 5 8 13 21 34

04. Consider the following method.

```
public static int method0204(int n)
{
    int k1 = 3;
    int k2 = 2;
    int k3 = 1;
    for (int p = 3; p <= n; p++)
    {
        k1 = k2;
        k2 = k3;
        k3 = k1 + k2;
    }
    return k3;
}
```

What value is returned as a result of the call `method0204(9)` ?

- (A) 11
- (B) 18
- (C) 29
- (D) 47
- (E) 76

05. Consider the following two methods.

```
public static int method0205a(int x, int y)
{
    int temp = method0205b(x,y);
    return x / temp * y;
}

public static int method0205b(int p, int q)
{
    int rem = 1;
    int k = 0;
    while (rem != 0)
    {
        rem = p % q;
        if (rem == 0)
        {
            k = q;
        }
        else
        {
            p = q;
            q = rem;
        }
    }
    return k;
}
```

What value is returned as a result of the call `method0205a(120,108)` ?

- (A) 12
- (B) 120
- (C) 108
- (D) 1080
- (E) 1200

06. Consider the following program.

```
public class DS0206
{
    public static void main(String args[])
    {
        int x = 10;
        method0206(x);
        System.out.println(x);
    }

    public static void method0206(int n)
    {
        for (int k = 1; k <= 10; k++)
            n += k;
    }
}
```

What is printed as a result of executing the program?

- (A) 65
- (B) 10
- (C) 20
- (D) 56
- (E) 29

07. Consider the following two methods.

```
public static void main(String args[])
{
    int x = 10;
    x = method0207(x);
    System.out.println(x);
}

public static int method0207(int n)
{
    for (int k = 1; k <= 10; k++)
        n += k;
    return n;
}
```

What is printed as a result of executing the `main` method ?

- (A) 65
- (B) 10
- (C) 20
- (D) 56
- (E) 29

08. Consider the following program.

```
public class DS0208
{
    public static void main(String args[])
    {
        int x = 10;
        int y = 20;
        System.out.println(x + " " + y);
        swap(x,y);
    }

    public static void swap(int p, int q)
    {
        int temp = p;
        p = q;
        q = temp;
        System.out.println(p + " " + q);
    }
}
```

What is printed as a result of executing the program?

- (A) 10 20
10 20
- (B) 10 20
20 10
- (C) 10 10
20 20
- (D) 20 20
20 20
- (E) There is no output displayed

09. Consider the following program.

```
public class DS0209
{
    public static void main(String args[])
    {
        int x = 10;
        int y = 20;
        System.out.println(x + " " + y);
        swap(x,y);
        System.out.println(x + " " + y);
    }

    public static void swap(int p, int q)
    {
        int temp = p;
        p = q;
        q = temp;
    }
}
```

What is printed as a result of executing the program?

- (A) 10 20
10 20
- (B) 10 20
20 10
- (C) 10 10
20 20
- (D) 20 20
20 20
- (E) There is no output displayed.

10. Consider the following two classes.

```
public class DS0210
{
    public static void main(String args[])
    {
        Widget w1 = new Widget(100);
        Widget w2 = new Widget(200);
        System.out.println(w1 + " " + w2);
        swap(w1,w2);
        System.out.println(w1 + " " + w2);
    }

    public static void swap(Widget p, Widget q)
    {
        Widget temp = p;
        p = q;
        q = temp;
    }
}

class Widget
{
    int numWidgets;
    public Widget(int w) { numWidgets = w; }
    public int getWidgets() { return numWidgets; }
    public void setWidgets(int w) { numWidgets = w; }
}
```

What is printed as a result of executing the program?

- (A) 100 200
100 200
- (B) 100 200
200 100
- (C) 100 200
200 200
- (D) Widget@18d107f Widget@360be0
Widget@18d107f Widget@360be0
- (E) Widget@18d107f Widget@360be0
Widget@360be0 Widget@18d107f

11. Consider the following two classes.

```
public class DS0211
{
    public static void main(String args[])
    {
        Widget w1 = new Widget(100);
        Widget w2 = new Widget(200);
        System.out.println(w1.getWidgets() + " " + w2.getWidgets());
        swap(w1,w2);
        System.out.println(w1.getWidgets() + " " + w2.getWidgets());
    }

    public static void swap(Widget p, Widget q)
    {
        Widget temp = p;
        p = q;
        q = temp;
    }
}

class Widget
{
    int numWidgets;
    public Widget(int w) { numWidgets = w; }
    public int getWidgets() { return numWidgets; }
    public void setWidgets(int w) { numWidgets = w; }
}
```

What is printed as a result of executing the program?

- (A) 100 200
100 200
- (B) 100 200
200 100
- (C) 100 200
200 200
- (D) Widget@18d107f Widget@360be0
Widget@18d107f Widget@360be0
- (E) Widget@18d107f Widget@360be0
Widget@360be0 Widget@18d107f

12. Consider the following two classes.

```
public class DS0212
{
    public static void main(String args[])
    {
        Widget w1 = new Widget(100);
        Widget w2 = new Widget(200);
        System.out.println(w1.getWidgets() + " " + w2.getWidgets());
        swap(w1,w2);
        System.out.println(w1.getWidgets() + " " + w2.getWidgets());
    }

    public static void swap(Widget p, Widget q)
    {
        int temp = p.getWidgets();
        p.setWidgets(q.getWidgets());
        q.setWidgets(temp);
    }
}

class Widget
{
    int numWidgets;
    public Widget(int w) { numWidgets = w; }
    public int getWidgets() { return numWidgets; }
    public void setWidgets(int w) { numWidgets = w; }
}
```

What is printed as a result of executing the program?

- (A) 100 200
100 200
- (B) 100 200
200 100
- (C) 100 200
200 200
- (D) Widget@18d107f Widget@360be0
Widget@18d107f Widget@360be0
- (E) Widget@18d107f Widget@360be0
Widget@360be0 Widget@18d107f

13. Consider the following two classes.

```
public class DS0213
{
    public static void main(String args[])
    {
        boohiss(3.0);
    }

    public static void boohiss(int n)
    {
        System.out.println("Hello");
    }

    public static void boohiss(double x)
    {
        System.out.println("Hola");
    }

    public static void boohiss(char q)
    {
        System.out.println("Aloha");
    }

    public static void boohiss(String s)
    {
        System.out.println("Goeden dag");
    }
}
```

What is printed as a result of executing the program?

- (A) Hello
- (B) Hola
- (C) Aloha
- (D) Goeden dag
- (E) Duplicate method identifier error

14. Consider the following two classes.

```
public class DS0214
{
    public static void main(String args[])
    {
        boohiss(1,2);
        boohiss(1);
        boohiss(1,2,3,4);
        boohiss(1,2,3);
    }

    public static void boohiss(int n)
    {
        System.out.println("Hello");
    }

    public static void boohiss(int p, int q)
    {
        System.out.println("Hola");
    }

    public static void boohiss(int x, int y, int z)
    {
        System.out.println("Aloha");
    }

    public static void boohiss(int a, int b, int c, int d)
    {
        System.out.println("Goeden dag");
    }
}
```

What is printed as a result of executing the program?

- (A) Hello
Hola
Aloha
Goeden dag
- (B) Hola
Hello
Goeden dag
Aloha
- (C) Hola
Aloha
Goeden dag
Hello
- (D) Goeden dag
Hello
Hola
Aloha
- (E) Duplicate method identifier error

15. Consider the following two classes.

```
public class DS0215
{
    public static void main(String args[])
    {
        boohiss(true);
        boohiss(7);
        boohiss(2,11);
    }

    public static void boohiss(int n)
    {
        System.out.println("Hello");
    }

    public static void boohiss(int p, int q)
    {
        System.out.println("Hola");
    }

    public static void boohiss(boolean z)
    {
        System.out.println("Aloha");
    }

    public static void boohiss(int x, int y)
    {
        System.out.println("Goeden dag");
    }
}
```

What is printed as a result of executing the program?

- (A) Aloha
Hello
Hola
- (B) Aloha
Hello
Goeden dag
- (C) Hola
Aloha
Goeden dag
- (D) Goeden dag
Hello
Hola
- (E) Duplicate method identifier error

Chapter 3 Review

Boolean Algebra

The review for the Boolean Algebra chapter will be longer than the other chapters. There are two reasons for this approach. First, most computer science text books do not include a Boolean Algebra chapter. Second, statistics performed on AP Computer Science Examination results have demonstrated that students who perform poorly on Boolean Algebra questions are more likely to perform poorly on the test as a whole. The logic explained by Boolean Algebra is used regularly in program segments that involve control structures with compound conditions.

APCS Examination Alert

The APCS Examination includes a variety of Boolean logic questions. Many questions require indirect knowledge of Boolean logic, and other questions are directly focused on testing a student's understanding of Boolean concepts.

Test results have shown that many students score quite poorly on this part of the APCS Examination.

Statistical Analysis of these test results have also shown that the students who perform poorly on Boolean logic questions, perform poorly on the AP Exam as a whole; and the students who perform well on the Boolean Logic questions, perform well on the AP Exam as a whole.

A good starting point is to look at a variety of English sentences and determine if these sentences are Boolean statements or not. So, what are the criteria for a Boolean statement? The sentence, statement, condition, whatever, must be **true** or **false**. Questions, ambiguities, and arguments are not Boolean statements. You can see why this branch of mathematics has a major impact on computer science. The basis of processing data in a computer is the binary system of **on** and **off**, which is identical to the Boolean **true** or **false**. Each of the following five English statements is a Boolean statement.

A mile is longer than a kilometer.

July and August both have the same number of days.

A pound of feathers is lighter than a pound of lead.

The Moon is larger than the Sun.

New York City has more people than Baltimore.

Boolean Operators

The arithmetic operators (+ - * /) *addition, subtraction, multiplication* and *division* are performed according to the rules for each operator. There are also a set of Boolean operators with their own set of rules. The APCS Exam tests three different Boolean operators. The rules of Boolean operators can be conveniently displayed in a **truth table**. This is a table, which shows the possible combinations of a Boolean statement and indicates the value (**true** or **false**) of each Boolean expression.

In the truth tables that follow, a single letter indicates a single, simple Boolean condition. Such a condition is either **true** or **false**. Boolean statement **A** is true or false. Likewise Boolean statement **B** is true or false. The truth tables will show the results of Boolean statements that use both **A** and **B** with a variety of Boolean operators. Employment requirements will be used to explain the logic of each truth table. In each case imagine that an accountant needs to be hired. Condition **A** determines if the applicant has a **Degree** and condition **B** determines if the applicant has at least five years experience.

Boolean Or Operator

The or Operator		
A	B	A or B
T	T	T
T	F	T
F	T	T
F	F	F

Notice that two conditions have four possible combinations. It is important that you know the result for each type of combination.

In this case the employment analogy requires a **Degree or Experience**. This requirement is quite relaxed. You only have a Degree, fine. You only have Experience, that's also fine. You have both, definitely fine. You have neither, that's a problem.

Boolean And Operator

The and Operator		
A	B	A and B
T	T	T
T	F	F
F	T	F
F	F	F

Now employment requires a **Degree and Experience**. This requirement is much more demanding than the **or** operator. You have a Degree, that may be fine, provided you also have Experience. If you have only one qualification, or the other qualification, that is not good enough. If you have neither qualification, forget showing up.

Boolean Not

The not Operator	
A	not A
T	F
F	T

The **not** operator is quite simple. Any statement is negated. Be careful not to assume that **not** implies **false**. The statement **not A** when **A equals false** becomes **true**. It is the concept of the double negative that is discouraged by teachers. A statement like *I don't know nothing* logically means that the person *knows something*.

Truth Tables

Truth tables provide a convenient way to decide when a Boolean expression is **true**, and if an expression is equivalent to another Boolean expression. The last section introduced simple truth tables to explain the Boolean operators. Now we are going to look at more complex Boolean statements with the help of more complex truth tables. The Boolean statements will not only be more complex, they will also include a larger number of different conditions.

Consider the Boolean expression **(A and B) or B**, shown in Truth Table #1

Truth Table #1			
A	B	A and B	(A and B) or B
T	T	T	T
T	F	F	F
F	T	F	T
F	F	F	F

What can be learned from this first truth table? There may be many **Ts** and **Fs**, but something very significant is demonstrated. The compound Boolean statement of **(A and B) or B** has the same truth table as **B**. The value of A is irrelevant. If **B** is true the entire expression is **true**.

The expression **(A and B) or C** is similar to the previous problem, but now a third Boolean operand is introduced. This third operand suddenly changes the truth table considerably. With two operands (A and B) there are four possible combinations to consider. Now that there are three operands (A, B and C), Truth Table #2 will need to consider eight different possibilities.

Truth Table #2				
A	B	C	A or B	(A or B) and C
T	T	T	T	T
T	T	F	T	F
T	F	T	T	T
T	F	F	T	F
F	T	T	T	T
F	T	F	T	F
F	F	T	F	F
F	F	F	F	F

Does Truth Table #2 demonstrate anything interesting? Notice that **C** must be true, otherwise the entire statement cannot be true. Perhaps this makes sense or maybe it seems very bizarre. So what exactly is the point of these truth tables? The point is somewhat obscure. There are Boolean rules that most people would not expect. In fact, a lot of people find some of the rules of Boolean Algebra pretty weird and do not believe them. Boolean Algebra is not very intuitive at the first introduction. Some Boolean Algebra rules seem to contradict well established regular high school Algebra facts. With the use of truth tables these rules can be proven. How? Because the truth tables of **equivalent** Boolean expressions are identical.

Truth Table Fact

The truth tables of **equivalent** Boolean expressions are identical.

Boolean expressions that use the **not** operator often create the most confusion and mistakes in computer science. In Boolean Algebra the **tilde** (\sim) is used for the **not** operator. Consider the following expression:

$$\sim(A \text{ or } B)$$

For reasons unknown it may be interesting to remove the parentheses and still maintain a Boolean expression with the same value. Armed with high school Algebra, the clever use of the *distributive property* will create the following:

$$\sim A \text{ or } \sim B$$

It may seem quite logical to use the *distributive property* to make the decision that the Boolean expression $\sim(A \text{ or } B)$ is equivalent to $\sim A \text{ or } \sim B$. This type of logic is correct in high school **Algebra**. Will this same law also apply to **Boolean Algebra**? In this situation the trusty truth tables come to the rescue. Truth Table #3 creates a table for each one of the expressions and then compares the column values.

Truth Table #3						
A	B	A or B	$\sim(A \text{ or } B)$	$\sim A$	$\sim B$	$\sim A \text{ or } \sim B$
T	T	T	>F<	F	F	>F<
T	F	T	>F<	F	T	>T<
F	T	T	>F<	T	F	>T<
F	F	F	>T<	T	T	>T<

Truth Table #3 shows that the *distributive property* logic of regular Algebra does not apply. The truth tables of the two expressions are not equivalent. The final truth tables have been highlighted with arrows. The table {F F F T} is quite different from the table {F T T T}.

The truth table demonstrated that the distributive property does not work with Boolean Algebra. So what does exactly happen the parentheses are removed? Regular Algebra is no help, the truth tables confirmed that. How about considering if the Expression $\sim (A \text{ or } B)$ is equivalent to $\sim A \text{ and } \sim B$ with Truth Table #4.

Truth Table #4						
A	B	A or B	$\sim(A \text{ or } B)$	$\sim A$	$\sim B$	$\sim A \text{ and } \sim B$
T	T	T	>F<	F	F	>F<
T	F	T	>F<	F	T	>F<
F	T	T	>F<	T	F	>F<
F	F	F	>T<	T	T	>T<

Now the two expressions are equivalent. This is entirely too weird, but the facts are staring you straight in the face. Perhaps you can appreciate now why this chapter is needed. Armed with only the rudimentary truth tables of the previous section, you would not simply conclude what was just proven. Lack of this knowledge has negative consequences on your programs and it does not help your AP Examination score much either.

In Truth Table #5 a similar problem is presented. It is a matter of altering the Boolean operands and checking to see if the expression $\sim (A \text{ and } B)$ is equivalent to $\sim A \text{ or } \sim B$.

Truth Table #5						
A	B	A and B	$\sim(A \text{ and } B)$	$\sim A$	$\sim B$	$\sim A \text{ or } \sim B$
T	T	T	>F<	F	F	>F<
T	F	F	>T<	F	T	>T<
F	T	F	>T<	T	F	>T<
F	F	F	>T<	T	T	>T<

Once again, the unexpected (or perhaps by now expected) expressions are equivalent to each other. You have actually been observing one of the more common, and more important laws of Boolean Algebra. There are many laws in Boolean Algebra, which have names. This review book is not the place to list all these named laws. The APCS Exam will not require that detailed type of knowledge. However, you do need to know *DeMorgan's Law*, which has been demonstrated by the last two truth tables.

Venn Diagrams and Boolean Algebra

Venn diagrams are useful tools for teaching *Set Theory*. With a rectangle to represent the *universal set* and a group of circles, representing individual sets inside the rectangle, it is possible to visually demonstrate many Set Theory concepts. You probably learned about Venn diagrams in one or more mathematics classes. Boolean Algebra would not have been mentioned in any of these classes and you learned terms like *union* and *intersection*. The relationship between Boolean Algebra and Set Theory is very close, and the same visual benefits of Venn diagrams can apply to Boolean Algebra, as well as Set Theory.

Set Intersection behaves like the logical and

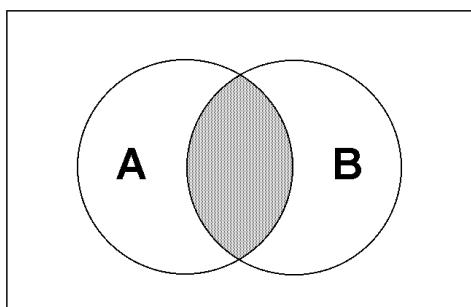
The Boolean Algebra logical **and** (*) can be demonstrated with Venn Diagrams, using **intersection**.

Venn Diagram 1 shows the intersection sets A and B. Consider one of the Boolean examples of the past. You have an employment requirement that applicants must have a college degree and they must also have five years of work experience. Now imagine that **Set A** represents people with a college degree, and **Set B** represents people with at least five years of work experience. In this case the shaded section is the result of saying **A and B** in Boolean Algebra or the *intersection* of **Set A** and **Set B** in Set Theory.

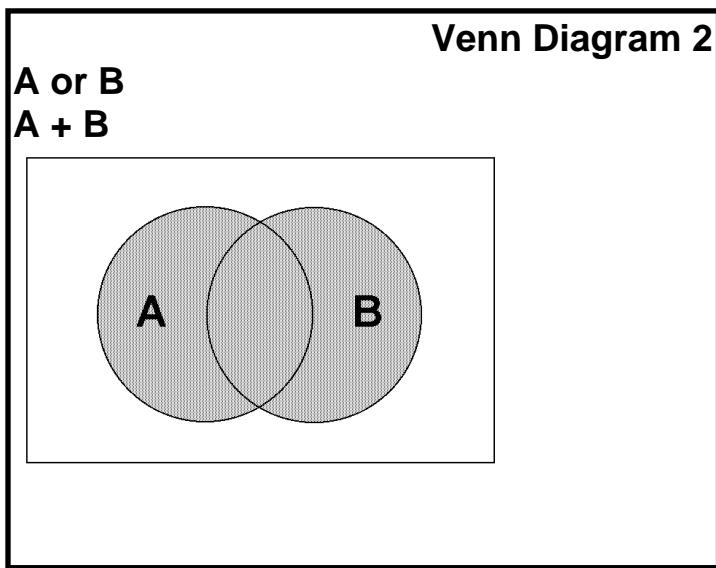
Venn Diagram 1

A and B

A * B



This time our business will hire people that have either a college degree or at least five years of experience. With the logical **or** there are now many more people qualified to apply for the job. We now can accept everybody in **Set A**, which are the college degree people as well as everybody in **Set B**, which are the experience people. In Boolean Algebra this is stated with the expression **A or B**. In set theory this is called the *union* of **Set A** and **Set B**, which is illustrated with Venn Diagram 2.



The reason for using Venn diagrams is to help illustrate Boolean Algebra concepts. With small expressions like **A and B** and **A or B** you may not see much need for using Venn diagrams.

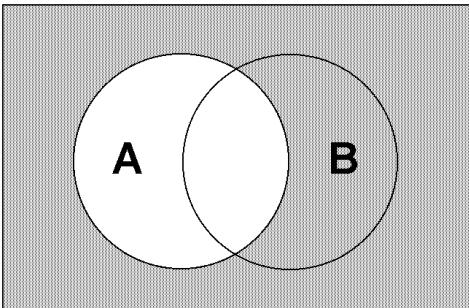
However, more complex expressions can be assisted with the help of set theory. Venn diagrams are particularly helpful to demonstrate that two different expressions are equivalent. You have seen truth tables used for proving equivalent expressions. You can now do the same with the aid of Set Theory and Venn diagrams.

Additional Venn diagram examples will be shown. A few examples will be given that come from the laws of Boolean Algebra, but not every law is going to be illustrated with Set Theory.

Venn diagrams 3 and 4 illustrate the **not** operator, which is identical in Boolean Algebra, as well as in Set Theory. Expression **not(A)** is shown in Venn Diagram 3. Expression **not(B)** is shown in Venn Diagram 4.

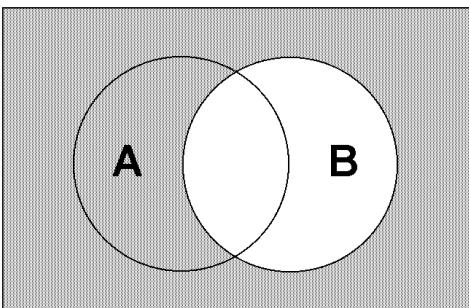
Venn Diagram 3

not(A)
 $\sim A$

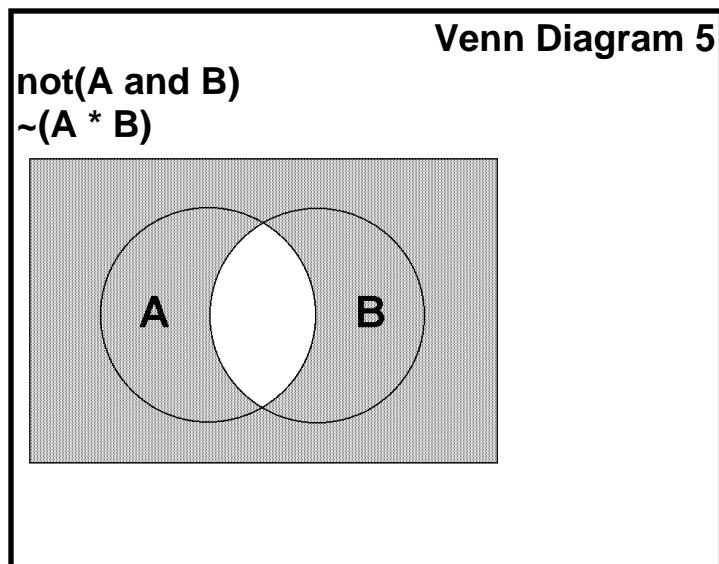


Venn Diagram 4

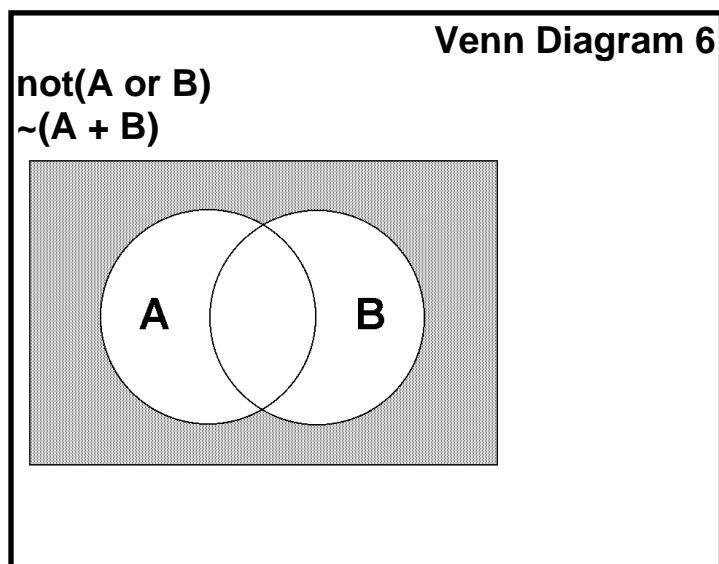
not(B)
 $\sim B$



The remaining Venn diagrams will show a variety of compound Boolean expressions. Venn diagram 5 illustrates the negation of the *intersection* of **A** and **B**, which is **not(A and B)** in Boolean Algebra.



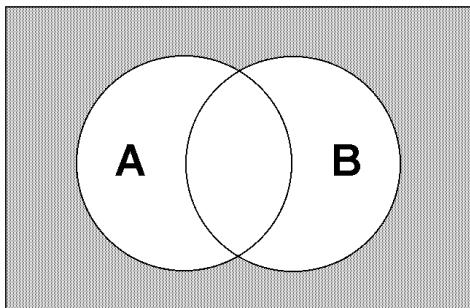
Venn diagram 6 is very similar to the previous example, but this illustrates the negation of the *union* of **Set A** and **Set B**. You can compare diagrams 5 and 6 with the earlier diagrams 1 and 2 to notice the pattern of negation.



One of the most important Boolean Algebra Laws for computer science is called *DeMorgan's Law*, which was demonstrated earlier with truth tables. This law states that the Boolean expression **not(A or B)** is equivalent to the Boolean expression **not(A) and not(B)**. The correctness of this law was proven earlier with truth tables. DeMorgan's Law can also be illustrated with Venn diagrams. Note that Venn diagram 6 shows **not(A or B)** and Venn diagram 7 shows **not(A) and not(B)**. Most importantly, note that both Venn diagrams 6 and 7 illustrate the exact same result.

Venn Diagram 7

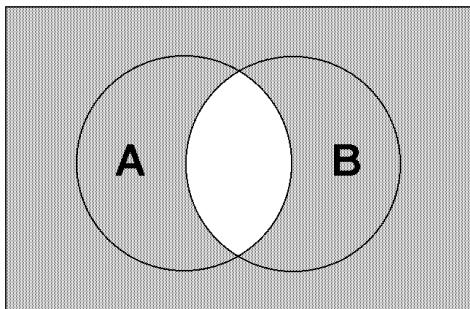
not(A) and not(B)
 $\sim A * \sim B$



The second part of *DeMorgan's Law* states that Boolean expression **not(A and B)** is equivalent to Boolean expression **not(A) or not(B)**. This law is proven by Venn diagrams 5 and 8.

Venn Diagram 8

not(A) or not(B)
 $\sim A + \sim B$



Boolean Algebra Pitfalls

The pitfalls with Boolean Algebra are usually caused by a lack of understanding of the rules of Boolean Algebra. A keen understanding of logic in general will solve many problems and truth tables, while tedious, assist in determining equivalent expressions. Five problems follow that show a variety of problems with an explanation of their solutions.

Sample Problem 1. Answer: (D)

The Boolean expression

not(A and B and C)

is equivalent to which of the following expressions?

- (A) $A \neq B \neq C$
- (B) $A \text{ and } B \text{ and } C$
- (C) $A \text{ or } B \text{ or } C$
- (D) $\text{not } A \text{ or not } B \text{ or not } C$
- (E) $\text{not } A \text{ and not } B \text{ and not } C$

Problem #1 checks to see if you can apply *DeMorgan's Law* to an expression with three operands. The same logic and the same rules apply. The Boolean operator is changed from **and** to **or**. The use of three operands does not alter the logic of DeMorgan's Law.

Sample Problem 2. Answer: (A)

The Boolean expression

(A and B) and not(A and B)

evaluates to

- (A) false in all cases.
- (B) true in all cases.
- (C) true whenever only A is true or only B is true.
- (D) true whenever both A is true and B is true.
- (E) false only when both A is false and B is false.

In problem #2 the Boolean expression **(A and B)** can only be true or false. Using **not** in front of the expression will change true to false or false to true. It is not possible for the Boolean expression **(A and B)** and the Boolean expression **not(A and B)** to have the same value. Two expressions that are always different with an **and** operator will always evaluate to **false**.

Sample Problem 3. Answer: (C)

The Boolean expression

not((A < B) and (C > D))

is equivalent to which of the following expressions?

- (A) (A < B) **or** (C > D)
- (B) (A >= B) **and** (C <= D)
- (C) (A >= B) **or** (C <= D)
- (D) (A > B) **and** (C < D)
- (E) (A > B) **or** (C < D)

Problem #3 requires two steps and it may help to use some substitution. Consider the expression **not(X and Y)**. Removal of the parentheses in that expression, using *DeMorgan's Law*, results in **not X or not Y**. In problem #3 substitute (A < B) with X and (C > D) with Y.

With that substitution the results is **not(A < B) or not (C > D)**. After removal of the **not** operator you simply the expression to **(A >= B) or (C <= D)**.

Sample Problem 4. Answer: (E)

The Boolean expression

(A > B) or (A <= B)

can be simplified to which of the following expressions?

- (A) A **or** B
- (B) A **and** B
- (C) A **and not** B
- (D) **false**
- (E) **true**

It is important to realize that **true** and **false** are Boolean expressions. This means that the Boolean expression **not(A > B)** equals **(A <= B)**. It then follows that **true or false** always becomes **true**.

Sample Problem 5. Answer: (B)

The Boolean expression

$$(A \text{ and } B) \text{ and } (\text{not } A \text{ or not } B)$$

evaluates to

- (A) true in all cases.
- (B) false in all cases.
- (C) true only whenever both A is true and B is true.
- (D) false only whenever both A is false and B is false.
- (E) true only whenever A is true or B is true.

This problem may benefit from using DeMorgan's Law in reverse. The second Boolean expression (**not A or not B**) is equivalent to **not(A and B)**. Now the entire expression with a substitution of x equals **(A and B)** results in **(x or not x)**, which is always **false**. Perhaps the "logical" approach to deduce the correct answer is obscure. Anytime a Boolean problem is difficult to determine, truth tables can always be used. Truth table #6, below, handles the same problem and demonstrates in the final column that the expression is always false.

Truth Table #6

A	B	A and B	~A	~B	~A or ~B	(A and B) and (~A or ~B)
T	T	T	F	F	F	F
T	F	F	F	T	T	F
F	T	F	T	F	T	F
F	F	F	T	T	T	F

Sample Problem 6. Answer: (E)

If **a** equals **false**, what is the value of the following expression?

$$a \text{ and } ((b \text{ or } c) \text{ and } (a \text{ or } c) \text{ or } (a \text{ and } b) \text{ and } (\text{not } b \text{ and not } c))$$

- (A) The expression cannot be evaluated without the values of **b** and **c**.
- (B) The expression is **false**, if both **b** and **c** are **false**, and **true** otherwise.
- (C) The expression is **true**, if both **b** and **c** are **true**, and **false** otherwise.
- (D) **true**
- (E) **false**

This problem looks much tougher than it is. The expression looks complex with many Boolean operands and lots of parentheses. The reality is that the very first **a** operand is immediately followed by an **and** operator. Boolean logic dictates that one **false** operand followed by an **and** operator makes the entire expression **false**. It is not necessary to create an involved truth table or logically digest the large expression following the first **and**. With **a** equals to **false**, the entire expression is **false**. This concept is called *short circuiting*.

Sample Problem 7. Answer: (D)

If **a** equals **true**, what is the value of the following expression?

a or ((b or c) and (a or c) or (a and b) and (not b and not c))

- (A) The expression cannot be evaluated without the values of **b** and **c**.
- (B) The expression is **false**, if both **b** and **c** are **false**, and **true** otherwise.
- (C) The expression is **true**, if both **b** and **c** are **true**, and **false** otherwise.
- (D) **true**
- (E) **false**

This problem is another example of *short circuiting*. Boolean logic dictates that one **true** operand followed by an **or** operator makes the entire expression **true**. Just like the previous problem, it is not necessary to create an involved truth table or logically digest the large expression following the first **or**. With **a** equals to **true**, the entire expression is **true**.

Solving Boolean Logic Programs

There are four ways to solve Boolean Logic problems:

1. Use known Boolean Algebra laws, like DeMorgan's law.
2. Use logical steps, such as shown by the previous problems.
3. Create truth tables to determine the results.
4. Use short circuiting where applicable.

Note: Java uses short circuiting with compound conditions.

Chapter 3 Questions

Boolean Algebra

01. The Boolean expression

$(A \mid\mid B) \And A$

is true

- (A) whenever A is true.
- (B) whenever B is true.
- (C) whenever either A is true or B is true.
- (D) whenever both A is true and B is true.
- (E) in all cases.

02. The Boolean expression

$(A \And B) \mid\mid B$

is true

- (A) whenever A is true.
- (B) whenever B is true.
- (C) whenever either A is true or B is true.
- (D) whenever both A is true and B is true.
- (E) in all cases.

03. The Boolean expression

$(A \And B) \And A$

is true

- (A) whenever A is true.
- (B) whenever B is true.
- (C) whenever either A is true or B is true.
- (D) whenever both A is true and B is true.
- (E) in all cases.

04. Consider the Boolean expression

$$(A \mid\mid B) \mid\mid B$$

For which of the following conditions is the expression above true?

- I. whenever either A is true or B is true.
- II. whenever both A is true and B is true.
- III. whenever both A is not true and B is not true.

- (A) I only
- (B) II only
- (C) III only
- (D) I and II only
- (E) II and III only

05. The Boolean expression

$$!(A \mid\mid B)$$

is equivalent to which of the following expressions?

- (A) $A \neq B$
- (B) $\neg A \mid\mid B$
- (C) $A \&\& B$
- (D) $\neg A \mid\mid \neg B$
- (E) $\neg A \&\& \neg B$

06. The Boolean expression

$$\neg(A \&\& B)$$

is equivalent to which of the following expressions?

- (A) $A \neq B$
- (B) $\neg A \mid\mid B$
- (C) $A \&\& B$
- (D) $\neg A \mid\mid \neg B$
- (E) $\neg A \&\& \neg B$

07. The Boolean expression

$$(A \mid\mid B) \mid\mid ! (A \mid\mid B)$$

evaluates to

- (A) false in all cases.
- (B) true in all cases.
- (C) true whenever only A is true or only B is true.
- (D) true whenever A is true or B is true.
- (E) true whenever both A is true and B is true.

08. The Boolean expression

$$! ((A < B) \&\& (C > D))$$

is equivalent to which of the following expressions?

- (A) $(A < B) \mid\mid (C > D)$
- (B) $(A \geq B) \&\& (C \leq D)$
- (C) $(A \geq B) \mid\mid (C \leq D)$
- (D) $(A > B) \&\& (C < D)$
- (E) $(A > B) \mid\mid (C < D)$

09. The Boolean expression

$$! ((A < B) \mid\mid (C > D))$$

is equivalent to which of the following expressions?

- (A) $(A < B) \&\& (C > D)$
- (B) $(A \geq B) \&\& (C \leq D)$
- (C) $(A \geq B) \parallel (C \leq D)$
- (D) $(A > B) \&\& (C < D)$
- (E) $(A > B) \parallel (C < D)$

10. Consider the following program segment.

```
int p = <some integer greater than zero>
int q = <some integer greater than zero>
while (p != 0 && q != 0)
{
    p--;
    q--;
    niftyMethod();
}
```

For which values of **p** and **q** will the loop structure condition *short-circuit*?

- (A) whenever **p** equals 0
- (B) whenever **q** equals 0
- (C) only whenever both **p** equals 0 and **q** equals 0
- (D) only whenever **p** equals 0 or **q** equals 0
- (E) only whenever **p** equals 0 or only **q** equals 0, but not when both are equal to zero.

11. The Boolean expression

$(A \&\& B) \&\& (\neg A \mid\mid \neg B)$

evaluates to

- (A) false in all cases.
- (B) true in all cases.
- (C) true whenever only A is true or only B is true.
- (D) true whenever A is true or B is true.
- (E) true whenever both A is true and B is true.

12. The Boolean expression

$(A \mid\mid B) \&\& \neg(A \mid\mid B)$

evaluates to

- (A) false in all cases.
- (B) true in all cases.
- (C) true whenever only A is true or only B is true.
- (D) true whenever A is true or B is true.
- (E) true whenever both A is true and B is true.

13. Consider the expression below.

```
A && ( (B || !C) && (!D && F) || !(G && H) && !(J || !K) )
```

If all you know is that A is false, what can you determine about the value of the expression above?

- (A) The expression evaluates to true.
- (B) The expression evaluates to false.
- (C) The expression will evaluate to true if B is true, false otherwise.
- (D) The expression will evaluate to true if B is false, true otherwise.
- (E) Nothing can be determined without knowing the values of B, C, D, E, F, G, H, J and K.

14. Consider the following code segment.

```
for (int x = 1; x < 50; x++)  
{  
    boolean pineapple = (x % 5 == 0);  
    if (pineapple)  
        System.out.println(pineapple);  
}
```

What is printed as a result of executing the code segment?

- (A) 10 trues and 40 falses
- (B) 9 trues and 40 falses
- (C) 9 trues and 41 falses
- (D) 10 trues
- (E) 9 trues

15. Consider the following code segment.

```
boolean pineapple = true;
int guava = 1;
while(pineapple)
{
    guava += guava;
    System.out.println(guava);
    pineapple = (guava < 100);
}
```

What are the first and last numbers displayed by this code segment?

- (A) 1 and 99
- (B) 1 and 64
- (C) 2 and 64
- (D) 1 and 128
- (E) 2 and 128

Chapter 4 Review

Object Oriented Programming

Object Oriented Programming (**OOP**) and Object Oriented Design (**OOD**) exist for the primary purpose of reliability. Programs can look attractive. Programs can execute rapidly. Programs may do many amazing things, but every fancy feature is irrelevant if the program does not perform in a reliable manner. How is this reliability achieved? This is no easy question. A thorough answer can be written in an entire text book, and this is only a brief review prior to a short set of questions. Perhaps the definition of OOP, shown below, tells the story.

Object Oriented Programming (**OOP**) is a style of programming that incorporates the three features of **encapsulation**, **polymorphism** and **inheritance**.

If you are a stranger to *encapsulation*, *inheritance* and *polymorphism*, then it is time to get some serious help. Hopefully, the APCS Exam is not any time soon. This section concerns itself with *encapsulation* and *inheritance*. A later chapter will cover *polymorphism*.

Encapsulation implies that something is placed in a capsule or container. In computer science this means that the data and the methods, which process the data, are placed in the same container. Furthermore, only local methods that are in the same container, may access the data.

Consider a typical airport. You arrive and check in your luggage for a flight to New York. The suit cases are placed on a conveyer belt. The next person needs to go to Seattle and his suit cases go on the same conveyer belt. There may easily be 50 to 100 different destinations used by that same check-in counter and all the luggage goes on the same conveyer belt. This is a terrific example of a process that is the opposite of encapsulation. Data - the passengers and luggage - is handled by processes that are not dedicated to one mission. The same conveyer belt and the same check-in agent handle many totally different missions.

Now imagine an airport build with OOP features, such as encapsulation. You arrive and find an entrance for New York. You enter a corridor without intersections. Other passengers for different destinations can be seen through glass walls but cannot be reached. You arrive at a check-in counter. The agent and the conveyer belt are exclusively used for passengers flying to New York. The conveyer belt has a single purpose and all its luggage goes to the New York gate. You continue and after security you arrive at a single gate with a plane waiting to fly you to New York. This is encapsulation. You have arrived at a **New York** class and all the data and methods are dedicated to getting the data - passengers and luggage - to New York.

This special airport is an example of reliability. Airport designers do need to insure that it is not possible for any *non-New York* processors to handle the *New York* data.

Object Oriented Programming Feature Encapsulation

Encapsulation places the data, called attributes and the processes, called methods, which access the attributes, in the same container.

Another OOP feature is inheritance. Inheritance helps tremendously with reliability. Consider some practical class, called **Dance**, which has been thoroughly tested. The bugs are cleaned up and the class performs as expected. You wish to create a new type-of-dance class, which is more specialized, called **Tango**. It is certainly possible to create a brand-new class, but then your new class needs to be tested from scratch. You have the **Dance** class ready to go and by making your **Tango** class a subclass of the **Dance** class, many established methods are available. You only need to concentrate on new features.

Consider that you want to build camper vans. It is certainly possible to start from scratch and build the entire vehicle. You can also save lots of time by starting with a van that has proven quality. Now that you have obtained this excellent van, you put in special seats, a small bathroom and kitchen and create special fold-out beds. With surprising little effort you have a high quality camper van, without all the effort to start with all the basic details of van building.

Object Oriented Programming Feature Inheritance

Inheritance allows new classes, called sub classes, to use the methods of established classes, called super classes.

Remember that the third feature of Object Oriented Programming, *polymorphism*, is presented in a later chapter on interfaces and abstract classes.

Encapsulation and Inheritance Pitfalls

The program in **Example 4.1** compiles and it also executes. In this simple program there appears no problem. The reality is that encapsulation is severely violated. The attribute **numWidgets** is accessed twice from another class. The value of **numWidgets** is altered and then the value of **numWidgets** is displayed.

Example 4.1

```
public class Example41
{
    public static void main(String args[])
    {
        Widget widget = new Widget(10);
        widget.numWidgets = 25;
        System.out.println("Widget Count: " + widget.numWidgets);

    }
}

class Widget
{
    int numWidgets;

    public Widget(int n)
    {
        numWidgets = n;
    }
}
```

The program shown in **Example 4.1** is solved by declaring the **numWidgets** attribute **private**. Any attempt to access a private attribute will stop the compile and you will be greeted with an error message. **Example 4.2** shows the improved **Widget** class. Keyword **private** is now used to declare the **numWidgets** attribute. Any attempt to access the private attribute from outside the **Widget** class will generate a compile error.

Example 4.2

```
class Widget
{
    private int numWidgets;

    public Widget(int n)
    {
        numWidgets = n;
    }
}
```

In **Example 4.3** a subtle mistake is shown. The program compiles without problems. The program also executes, but the output displays a Widget Count of 0. What is happening? The arrow shows the problem statement. The constructor parameter in **numWidget** is the same identifier as the attribute. Java treats **numWidgets** in the **Widget** constructor as a local variable. The result is that the **numWidget** attribute becomes a default 0, but not 10.

Example 4.3

```
public class Example43
{
    public static void main(String args[])
    {
        Widget widget = new Widget(10);
        System.out.println("Widget Count: " + widget.getWidgets());
    }
}

class Widget
{
    private int numWidgets;

    public Widget(int numWidgets)
    {
        numWidgets = numWidgets; ←
    }

    public int getWidgets()
    {
        return numWidgets;
    }
}
```

Example 4.4 and **Example 4.5** each show a solution to this problem.

Example 4.4

```
public Widget(int n)
{
    numWidgets = n; ←
```

Example 4.5

```
public Widget(int numWidgets)
{
    this.numWidgets = numWidgets; ←
```

The program in **Example 4.6** looks innocent enough. There is a super class, called **Morf**, and there is a sub class, called **Dorf**. A **Dorf** object is constructed with parameter **25**. This program will not compile. The problem is that all inheritance cases start by calling the constructor of the super class, followed by the constructor of the sub class. This mistake is easily made, because the **main** method is only concerned with an object of the **Dorf** class. Parameter information is provided for the constructor of **Dorf**. The **Morf** constructor receives no value and complains.

Example 4.6

```
public class Example46
{
    public static void main(String args[])
    {
        Dorf dorf = new Dorf(25);
        System.out.println("Dorf Count: " + dorf.getDorfs());
    }
}

class Morf
{
    private int numMorphs;

    public Morf(int m) { numMorphs = m; }

}

class Dorf extends Morf
{
    private int numDorfs;

    public Dorf(int d) { numDorfs = d; }
    public int getDorfs() { return numDorfs; }
}
```

Inheritance requires considering the needs of the super class as well as the sub class. When an object of the sub class, **Dorf** in this case, is constructed, information must be passed for all the classes that are used. You must remember that the super class is involved whether you actually use the super class name or not.

Consider the example of making a camper van. A camper can **is-a** van. The van is used for conversion to a camper van. The client who wants a camper van will have request about the camper features. At the same time there must be information provided for the initial van. Is the engine a V-8 or a V-6? Is all-wheel drive required? Surely you can think of many other requirements.

In Java this is handled with the keyword **super**, which is placed at the start of the sub class constructor. The parameter in the **super** call is used to construct the super class object. In **Example 4.7** you will see the correct way to provide information for both classes.

Example 4.7

```
public class Example47
{
    public static void main(String args[])
    {
        Dorf dorf = new Dorf(10,25);
        System.out.println("Morf Count: " + dorf.getMorfs());
        System.out.println("Dorf Count: " + dorf.getDorfs());

    }
}

class Morf
{
    private int numMorfs;

    public Morf(int m)
    {
        numMorfs = m;
    }

    public int getMorfs()
    {
        return numMorfs;
    }
}

class Dorf extends Morf
{
    private int numDorfs;

    public Dorf(int m, int d)
    {
        super(m);
        numDorfs = d;
    }

    public int getDorfs()
    {
        return numDorfs;
    }
}
```

Program Output

```
Morf Count: 10
Dorf Count: 25
```

Chapter 4 Questions

Object Oriented Programming

01. Consider the following Widget class.

```
class Widget
{
    private int numWidgets;

    public Widget(int numWidgets)
    {
        numWidgets = numWidgets;
    }

    public int getWidgets()
    {
        return numWidgets;
    }
}
```

Assume that the following code segment appears in a client program.
What is printed by the code segment?

```
Widget wid = new Widget(10);
System.out.println(wid.getWidgets());
```

- (A) 10
- (B) 0
- (C) The program compiles, but there is no output.
- (D) The program does not compile.
- (E) The program compiles, but executes with a runtime error message.

02. Consider the following Widget class.

```
class Widget
{
    private int numWidgets;

    public void Widget(int widgets)
    {
        numWidgets = widgets;
    }

    public int getWidgets()
    {
        return numWidgets;
    }
}
```

Assume that the following code segment appears in a client program.
What is printed by the code segment?

```
Widget wid = new Widget(10);
System.out.println(wid.getWidgets());
```

- (A) 10
- (B) 0
- (C) The program compiles, but there is no output.
- (D) The program does not compile.
- (E) The program compiles, but executes with a runtime error message.

03. Consider the following List class.

```
class List
{
    private int[] list;

    /* missing constructor */

    public void showList()
    {
        for (int k = 0; k < list.length; k++)
            System.out.print(list[k] + " ");
    }
}
```

Assume that the following code segment appears in a client program. The output of executing the code segment is a row of ten 99s. Which of the following constructor implementations of **/* missing constructor */** will make the List class work as intended?

```
List list = new List(10,99);
list.showList();
System.out.println();
```

- (A)

```
public List(int size, int value)
{
    list = new int[size];
    for (int k = 0; k < size; k++)
        list[k] = value;
}
```
- (B)

```
public List()
{
    list = new int[10];
    for (int k = 0; k < size; k++)
        list[k] = 99;
}
```
- (C)

```
public List(int size, int value)
{
    for (int k = 0; k < size; k++)
        list[k] = value;
}
```
- (D)

```
public List(int size, int value)
{
    list = new int[size,value];
}
```
- (E)

```
public List(int size, int value)
{
    list = new int[size];
    for (int k = 0; k <= size; k++)
        list[k] = value;
}
```

04. Consider the following two classes.

```
class Rumba
{
    public Rumba()
    {
        System.out.println("Executing the Rumba constructor");
    }
}

class Mambo
{
    public Mambo()
    {
        System.out.println("Executing the Mambo constructor");
    }
}
```

Assume that the following code segment appears in a client program.
What is printed by the code segment?

```
Mambo m = new Mambo();
Rumba r = new Rumba();
```

- (A) Executing the Rumba constructor
Executing the Mambo constructor
- (B) Executing the Mambo constructor
Executing the Rumba constructor
- (C) Executing the Rumba constructor
Executing the Mambo constructor
Executing the Rumba constructor
- (D) Executing the Mambo constructor
Executing the Rumba constructor
Executing the Mambo constructor
- (E) Executing the Mambo constructor
Executing the Mambo constructor
Executing the Rumba constructor

05. Consider the following two classes.

```
class Rumba
{
    public Rumba()
    {
        System.out.println("Executing the Rumba constructor");
    }
}

class Mambo
{
    private Rumba dance;
    public Mambo()
    {
        System.out.println("Executing the Mambo constructor");
        dance = new Rumba();
    }
}
```

Assume that the following code segment appears in a client program.
What is printed by the code segment?

```
Mambo m = new Mambo();
Rumba r = new Rumba();
```

- (A) Executing the Rumba constructor
Executing the Mambo constructor
- (B) Executing the Mambo constructor
Executing the Rumba constructor
- (C) Executing the Rumba constructor
Executing the Mambo constructor
Executing the Rumba constructor
- (D) Executing the Mambo constructor
Executing the Rumba constructor
Executing the Mambo constructor
- (E) Executing the Mambo constructor
Executing the Rumba constructor
Executing the Rumba constructor

06. Consider the following two classes.

```
class Rumba
{
    public Rumba()
    {
        System.out.println("Executing the Rumba constructor");
    }
}

class Mambo extends Rumba
{
    public Mambo()
    {
        System.out.println("Executing the Mambo constructor");
    }
}
```

Assume that the following code segment appears in a client program.
What is printed by the code segment?

```
Mambo m = new Mambo();
Rumba r = new Rumba();
```

- (A) Executing the Rumba constructor
Executing the Mambo constructor
- (B) Executing the Mambo constructor
Executing the Rumba constructor
- (C) Executing the Rumba constructor
Executing the Mambo constructor
Executing the Rumba constructor
- (D) Executing the Mambo constructor
Executing the Rumba constructor
Executing the Mambo constructor
- (E) Executing the Mambo constructor
Executing the Mambo constructor
Executing the Rumba constructor

07. Consider the following class.

```
class Aardvark
{
    private int count;

    public Aardvark(int n)
    {
        count = n;
    }

    public String toString()
    {
        return "Aardvarks " + count;
    }
}
```

What type of inheritance concepts is demonstrated by the Aardvark class declaration?

- (A) multiple inheritance
- (B) redefining an inherited method
- (C) implementing an abstract interface
- (D) inheritance composition
- (E) implementing an abstract method

08. Consider the following two classes.

```
class Person
{
    private int age;
    public Person(int a)
    {
        age = a;;
    }
    public int getAge() { return age; }
}

class Student extends Person
{
    private double gpa;
    public Student(int a, double g)
    {
        /* missing code */
    }
    public double getGPA() { return gpa; }
}
```

Assume that the following code segment appears in a client program.

```
Student kathy = new Student(17,3.85);
System.out.println("Age : " + kathy.getAge());
System.out.println("GPA : " + kathy.getGPA());
```

Furthermore, assume that the following output should be the result of executing the code segment.

```
Age: 17
GPA: 3.85
```

Which of the following can be used to replace the **/* missing code */** in constructor `Student` so that the desired output is achieved?

- (A) `age = a;`
`gpa = g;`
- (B) `gpa = g;`
`super(a);`
- (C) `super(a);`
`gpa = g;`
- (D) `Person.age = a;`
`gpa = g;`
- (E) `person.age = a;`
`Student.gpa = g;`

09. Consider the following two classes.

```
class Person
{
    private int age;
    public Person(int a) { age = a; }
    public String toString()
    {
        return "Age: " + age + "\n";
    }
}

class Student extends Person
{
    private double gpa;
    public Student(int a, double g)
    {
        super(a);
        gpa = g;
    }
    public String toString()
    {
        /* missing code */
    }
}
```

Assume that the following code segment appears in a client program.

```
Student kathy = new Student(17,3.85);
System.out.println(kathy);
```

Furthermore, assume that the following output should be the result of executing the code segment.

```
Age: 17
GPA: 3.85
```

Which of the following can be used to replace the **/* missing code */** in method `toString` so that the desired output is achieved?

- (A) `return super.toString() + "GPA: " + gpa;`
- (B) `super.toString();
 return "GPA: " + gpa;`
- (C) `return super.toString();
 return "GPA: " + gpa;`
- (D) `return "Age: " + age + "\n" + "GPA: " + gpa;`
- (E) None of the above. It is not possible to achieve that output without a `Student` object.

Questions 10-12 refer to the following two classes and the code segment below.

```
class Student
{
    private int studentID;
    private double studentGPA;
    public Student(int id, double gpa)
    {
        studentID = id;
        studentGPA = gpa;
    }
    public int getID()      { return studentID; }
    public double getGPA()   { return studentGPA; }
}

class Students
{
    private int listSize;           // Maximum number of students allowed to register.
    private Student[] studentList;  // Stores the student records.
    private int lastIndex;          // Index of the latest student registered.

    public Students(int count)
    // precondition: count is a positive integer value.
    // postcondition: studentCount, lastIndex and studentList are initialized.
    {
        /* missing Code #1 */
    }

    public void addStudent(int id, double gpa)
    // precondition: A Students object is constructed according to specifications of the
    //               Students constructor.
    //               lastIndex is the studentList index of the last Student record added.
    // postcondition: A new Student object, storing id and gpa, is added to studentList.
    {
        /* missing code #2 */
    }

    public void showStudents()
    // precondition: a Students object is constructed according to specifications of the
    //               Students constructor.
    //               lastIndex is the studentList index of the last Student record added.
    // postcondition: all the Student objects are displayed if studentList is nonempty.
    //               otherwise showStudents displays "No Student Records Stored".
    {
        if (lastIndex == -1)
            System.out.println("No Student Records Stored");
        else
        {
            /* missing code #3 */
        }
    }
}

Students students = new Students(1000);
students.addStudent(12345,2.785);
students.addStudent(12346,3.225);
students.addStudent(12347,3.565);
students.showStudents();
```

10. Which one of the following five implementations of /* **missing code #1** */ will make the `Students` constructor method work as intended.

(A) `listSize = count;`
`lastIndex = -1;`

(B) `lastIndex = -1;`
`studentList = new Students[ListSize];`

(C) `listSize++;`
`Student = new Student(id,gpa);`
`lastIndex = 0;`

(D) `listSize = count;`
`studentList = new Students[listSize];`
`lastIndex = -1;`

(E) `listSize = count;`
`studentList = new Student[listSize];`
`lastIndex = -1;`

11. Which one of the following five implementations of /* **missing code #2** */ will make the `addStudent` method work as intended. Assume that the `Students` constructor is implemented correctly.

(A) `studentList[lastIndex] = new Student(id,gpa);`
`lastIndex++;`

(B) `lastIndex++;`
`studentList[lastIndex] = new Student(id,gpa);`

(C) `listSize++;`
`studentList[lastIndex] = Student(id,gpa);`

(D) `listSize++;`
`studentList[listSize] = new Student(id,gpa);`

(E) `listSize++;`
`studentList[lastIndex] = new Student(id,gpa);`

12. Which one of the following five implementations of /* missing code #3 */ will make the showStudents method work as intended. Assume that methods Students and addStudent are implemented correctly.

(A)

```
for (int index = 0; index <= lastIndex; index++)
{
    System.out.println("Student ID: " + student[index].studentID);
    System.out.println("Student GPA: " + student[index].studentGPA());
}
```

(B)

```
for (int index = 0; index <= lastIndex; index++)
{
    System.out.println("Student ID: " + student[index].getID());
    System.out.println("Student GPA: " + student[index].getGPA());
}
```

(C)

```
for (int index = 0; index <= lastIndex; index++)
{
    System.out.println("Student ID: " + studentList[index].getID());
    System.out.println("Student GPA: " + studentList[index].getGPA());
}
```

(D)

```
for (int index = 0; index <= lastIndex; index++)
{
    System.out.println("Student ID: " + getID(index));
    System.out.println("Student GPA: " + getGPA(index));
}
```

(E)

```
for (int lastIndex = 0; lastIndex <= listSize; index++)
{
    System.out.println("Student ID: " + studentList[index].getID());
    System.out.println("Student GPA: " + studentList[index].getGPA());
}
```

Questions 13-15 refer to the following incomplete classes.

```
class Animal
{
    private int numLegs;

    public Animal(int legs)
    {
        numLegs = legs;
    }
}

class Insect extends Animal
{
    public int getNumLegs()
    {
        return numLegs;
    }
}

class Fly extends Insect
{
    public void showData()
    {

    }
}

class Ant extends Insect
{

}

class Mammal extends Animal
{

}

class Mouse extends Mammal
{
    String name;
}
```

13. Which of the following classes have a multi-level inheritance relationship?

- I. Animal -- Insect -- Fly
- II. Animal -- Insect -- Mammal
- III. Mouse -- Fly -- Ant
- IV. Animal -- Mammal -- Mouse
- V. Animal -- Mammal -- Ant

- (A) I only
- (B) II and III only
- (C) I and IV only
- (D) IV and V only
- (E) I, II, IV and V only

14. As written, the `getNumLegs` method of the `Insect` class will not compile.

What is the best way to fix this problem?

- (A) Copy the `numLegs` attribute from the `Animal` class to the `Insect` class.
- (B) In the `Animal` class, make `numLegs` have public access.
- (C) Change the implementation of `getNumLegs` to return `super.numLegs;`
- (D) Move the `getNumLegs` method to the `Animal` class.
- (E) Copy the `getNumLegs` method to the `Animal` class.

15. Which of the following would be an appropriate constructor for the `Mammal` class?

- I. `public Mammal()`
 {
 `name = "Mickey";`
 }
- II. `public Mammal(String n)`
 {
 `name = n;`
 }
- III. `public Mammal()`
 {
 `super(4);`
 `name = "Mickey";`
 }
- IV. `public Mammal(int q)`
 {
 `super(q);`
 `name = "Mickey";`
 }
- V. `public Mammal(int q, String n)`
 {
 `super(q);`
 `name = n;`
 }

- (A) I only
- (B) I and II only
- (C) III only
- (D) III and IV only
- (E) III, IV and V only

Chapter 5 Review

Static Java 1D Arrays

Java has two types of arrays. First, there is the static array, which is the traditional array with index access only. Static arrays, as the name implies cannot be resized during program execution. Second, there is the **ArrayList** class, which is a dynamic array with many methods for data access and processing.

At first glance it may seem strange to include an older and supposedly not-very-useful-anymore, array implementation. The static array cannot be resized. The static array does not offer the OOP type access with methods. Why do we even bother to keep such an array? There are several reasons why the static array is still being used. The most important reason is that static arrays allow creating multi-dimensional array objects. Objects of the newer **ArrayList** class are strictly one-dimensional. Static arrays also provide very convenient initializer lists. Details about the **ArrayList** class will be handled in a separate chapter.

APCS Examination Alert

Both static Java arrays and dynamic **ArrayList** arrays are tested on the APCS Exam.

For the APCS Exam students need to know both *one-dimensional* and *two-dimensional* implementations of the static Java array.

The statements in **Example 5.1** demonstrate the declaration of three types of one-dimensional arrays. It also shows how to use three initializer lists during the declarations. It is not necessary to provide the size of the array with initializer lists. The number of elements in the lists indicates the size of the array object. Java accepts two styles of static array declarations. The two segments in **Example 5.1** are identical and demonstrate the two styles.

Example 5.1

```
int list1[] = {11,22,33,44,55};  
char list2[] = {'A','B','C','D','E'};  
String list3[] = {"AAA","BBB","CCC","DDD","EEE"};
```

```
int[] list1 = {11,22,33,44,55};  
char[] list2 = {'A','B','C','D','E'};  
String[] list3 = {"AAA","BBB","CCC","DDD","EEE"};
```

The program in **Example 5.2** declares a one-dimensional array with 20 **int** elements. The **list** elements are displayed by a loop that uses the **length** field of static array. Make sure not to use **length()**. **length** is a final field and not a method.

Example 5.2

```
public class Example52
{
    public static void main(String args[])
    {
        int list[] = new int[20];
        for (int k = 0; k < 20; k++)
            list[k] = (int) (Math.random() * 900 + 100);

        for (int k = 0; k < list.length; k++)
            System.out.println(list[k]);
    }
}
```

The program in **Example 5.3** declares the same **list** array as **Example 5.2**. This time note that the enhanced **for** loop, introduced with Java 5.0, is used. The new **for** loop is a clean, concise control structure that can be used with many data structures.

Example 5.3

```
public class Example53
{
    public static void main(String args[])
    {
        int list[] = new int[20];
        for (int k = 0; k < 20; k++)
            list[k] = (int) (Math.random() * 900 + 100);

        for (int number: list)
            System.out.println(number);
    }
}
```

APCS Examination Alert

Expect the *for..each* loop structure, introduced with Java 5.0, to be used frequently in the APCS Exam.

Static Java Array Pitfalls

Java static arrays are quite straight forward and there are few subtle features that cause problems. The primary concern is using an index that is outside the index range. Each array has **length** array elements, but the largest index of the array is **length-1**. The program in **Example 5.4** compiles without difficulties, but generates an **ArrayIndexOutOfBoundsException** error during program execution.

Example 5.4

```
public class Example54
{
    public static void main(String args[])
    {
        int list[] = new int[10];
        for (int k = 0; k < 10; k++)
            list[k] = (int) (Math.random() * 900 + 100);

        for (int k = 0; k <= list.length; k++)
            System.out.print(list[k] + " ");
    }
}
```

Program Output

```
209 403 369 604 783 252 779 290 569 163 Exception in
thread "main" java.lang.ArrayIndexOutOfBoundsException:
10 at Example54.main(Example54.java:10)
```

Java Static Arrays Alert

An array declared in the following manner:

```
int[] list = new int[max]
```

has **max** elements, but the index range is **[0..max-1]**.

Any attempt to access an index outside the proper range will result in an **ArrayIndexOutOfBoundsException** error during program execution.

Chapter 5 Questions

Static Java 1D Arrays

01. Consider the following method.

```
public static int method0501(int n)
{
    int[] temp = new int[n];
    temp[0] = 1;
    temp[1] = 3;
    for (int k = 2; k < n; k++)
        temp[k] = temp[k-1] + temp[k-2];
    return temp[n-1];
}
```

What value is returned by the call `method0501(10)`?

- (A) 21
- (B) 34
- (C) 47
- (D) 76
- (E) 123

02. Consider the following two methods.

```
public static void method0502a(int list[])
{
    int max = list.length-1;
    for (int k = 0; k < max; k++)
        if (list[k] > list[k+1])
            method0502b(list,k);
}

public static void method0502b(int list[], int p)
{
    int temp = list[p];
    list[p] = list[p+1];
    list[p+1] = temp;
}
```

Which of the following correctly describes the result of calling method `method0502`?

- (A) The smallest number is located in `list[max]`.
- (B) The largest number is located in `list[max]`.
- (C) The `list` array is sorted in ascending order.
- (D) The `list` array is sorted in descending order.
- (E) The elements in the `list` array are reversed.

03. Consider the following incomplete `method0503` method.

```
// postcondition: method0503 returns a0 - a1 + a2 - a3 + a4 - a5 ...
public static int method0503(int list[])
{
    /* missing code */

    return temp
}
```

Which of the following implementations of `/* missing code */` will make method `method0503` work as intended?

Implementation 1

```
int temp = 0;
for (int k = 0; k < list.length; k++)
    if (k % 2 == 0)
        temp += list[k];
    else
        temp -= list[k];
```

Implementation 2

```
int temp = 0;
for (int k = 0; k < list.length; k++)
    if (k % 2 != 0)
        temp += list[k];
    else
        temp -= list[k];
```

Implementation 3

```
int temp = list[0];
for (int k = 1; k < list.length; k++)
    if (k % 2 == 0)
        temp += list[k];
    else
        temp -= list[k];
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementation 1 & 2
- (E) Implementation 1 & 3

04. Consider the following two methods.

```
public static void method0504(int list[])
{
    for (int p = 1; p < list.length; p++)
        for (int q = 0; q < list.length-p; q++)
        {
            if (list[q] > list[q+1])
                swap(list[q],list[q+1]);
        }
}

public static void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
```

Which of the following correctly describes the result of calling method `method0504`?

- (A) The last `list` element is the smallest number.
- (B) The last `list` element is the largest number.
- (C) The `list` array is sorted in ascending order.
- (D) The `list` array is sorted in descending order.
- (E) The elements in the `list` array are unchanged.

05. Consider the following code segment.

```
int list[] = {11,22,33,44,55,66,77,88,99};
int n = list.length-1;
for (int k = 0; k <= n; k++)
{
    int temp = list[k];
    list[k] = list[n-k];
    list[n-k] = temp;
}
for (int k = 0; k < list.length; k++)
    System.out.print(list[k] + " ");
```

What will be output when the program segment executes?

- (A) 99 88 77 66 55 44 33 22 11
- (B) 11 22 33 44 55 66 77 88 99
- (C) 99 22 33 44 55 66 77 88 11
- (D) 11 88 77 66 55 44 33 22 99
- (E) 11 99 22 88 33 77 44 66 55

06. Consider the following code segment.

```
int list[] = {1,5,10,15,20,25,30,35,40,45,50};  
for (int k = 1; k < list.length; k++)  
    list[k] = list[k] / list[1];  
for (int k = 0; k < list.length; k++)  
    System.out.print(list[k] + " ");
```

What will be output when the program segment executes?

- (A) 1 5 10 15 20 25 30 35 40 45 50
- (B) 1 1 2 3 4 5 6 7 8 9 10
- (C) 1 1 10 15 20 25 30 35 40 45 50
- (D) 1 1 1 1 1 1 1 1 1 1
- (E) 0 0 0 0 0 0 0 0 0 0

07. Consider the following method.

```
// precondition: list is a non-empty array of sorted integers.  
// postcondition: getMedian returns the middle number if array size is odd,  
//                 and returns the mean of the two middle numbers if array size is even.  
public static double getMedian(int list[])  
{  
    int n = list.length;  
  
    /* missing code */  
}
```

Which of the following three implementations of **/* missing code */** will make method `getMedian` work as intended?

Implementation 1

```
if (n % 2 == 0)  
    return (double) (list[n/2] + list[n/2+1]) / 2;  
else  
    return list[n/2];
```

Implementation 2

```
if (n % 2 == 0)  
    return (double) (list[n/2-1] + list[n/2]) / 2;  
else  
    return list[n/2];
```

Implementation 3

```
if (n % 2 == 0)  
    return (list[n/2-1] + list[n/2]) / 2;  
else  
    return list[n/2];
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementations 1 and 2 only
- (E) Implementations 2 and 3 only

08. Consider the following method.

```
// precondition: list is a non-empty array of integers.  
// postcondition: getMean returns the average (mean) of the integers in list.  
public static double getMean(int list[]){  
    int n = list.length;  
  
    /* missing code */  
}
```

Implementation 1

```
int sum = 0;  
for (int item: list)  
    sum += item;  
return (double) sum / n;
```

Implementation 2

```
int sum = 0;  
for (int item: list)  
    sum += list[item];  
return (double) sum / n;
```

Implementation 3

```
int sum = 0;  
for each (int item: list)  
    sum += item:list;  
return sum / n;
```

Which of the following code segments satisfies the postcondition of method getMean?

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementations 1 and 2 only
- (E) Implementations 2 and 3 only

09. Consider the following code segment.

```
boolean list[] = new boolean [10];
list[0] = true;
for (int j = 1; j < 10; j++)
    list[j] = !list[j-1];
for (boolean item: list)
    System.out.print(item + " ");
```

What will be output when the program segment executes?

- (A) true true true true true true true true true true
- (B) false true false true false true false true false true false
- (C) true false true false true false true false true false true
- (D) true false true false true false true false true false
- (E) false true false true false true false true false true

10. Consider the following code segment.

```
boolean list[] = new boolean [26];
for (int j = 0; j < 26; j++)
    list[j] = true;
for (int j = 2; j <= 26; j++)
    for (int k = j; k < 26; k += j)
        list[k] = !list[k];
for (int j = 1; j < 26; j++)
    if (list[j])
        System.out.print(j + " ");
```

What will be output when the program segment executes?

- (A) 1 4 9 16 25
- (B) 5 10 15 20 25
- (C) 1 3 5 7 9 11 13 15 17 19 21 23 25
- (D) 2 4 6 8 10 12 14 16 18 20 22 24
- (E) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

11. Consider the following code segment.

```
int list[] = {100,101,102,103,104,105,106,107,108,109,110};  
for (int j = 0; j < list.length; j++)  
    list[j] = list[list.length-(j+1)];  
for (int item: list)  
    System.out.print(item + " ");
```

What will be output when the program segment executes?

- (A) 100 101 102 103 104 105 106 107 108 109 110
- (B) 110 109 108 107 106 105 104 103 102 101 100
- (C) 100 101 102 103 104 105 104 103 102 101 100
- (D) 110 109 108 107 106 105 106 107 108 109 110
- (E) 109 108 107 106 105 106 107 108 109 110

12. Consider the following code segment.

```
int list[] = {100,101,102,103,104,105,106,107,108,109,110};  
for (int j = 2; j <= list.length; j++)  
    for(int k = 0; k < list.length - j; k++)  
        if (list[k] < list[k+1])  
        {  
            int temp = list[k];  
            list[k] = list[k+1];  
            list[k+1] = temp;  
        }  
    for (int j = 0; j < list.length; j++)  
        System.out.print(list[j] + " ");
```

What will be output when the program segment executes?

- (A) 100 101 102 103 104 105 106 107 108 109 110
- (B) 110 109 108 107 106 105 104 103 102 101 100
- (C) 100 110 109 108 107 106 105 104 103 102 101
- (D) 109 108 107 106 105 104 103 102 101 100 110
- (E) 102 102 102 102 102 102 102 102 102 102 102

13. Consider the following code segment.

```
int list[] = {100,101,102,103,104,105,106,107,108,109,110,111};  
for(int k = 0; k < list.length; k+=2)  
{  
    list[k] *= 3;  
    list[k+1]++;  
}  
for (int item: list)  
    System.out.print(item + " ");
```

What will be output when the program segment executes?

- (A) 300 303 306 309 312 315 318 321 324 327 330 333
- (B) 301 304 307 310 313 316 319 322 325 328 331 334
- (C) 300 101 306 103 312 105 318 107 324 109 330 111
- (D) 300 102 306 104 312 106 318 108 324 110 330 112
- (E) 101 303 103 309 105 315 109 321 115 327 121 333

14. Consider the following code segment and method.

```

// precondition: list is a non-empty array of integers
// postcondition: a new list is created and returned.
//                  The first half of the new list contains the elements from the original
//                  list in reverse order.
//                  The second half of the new list contains the elements from the original
//                  list again, in normal order.
//                  The first element of the original list only shows up once in the new list.
// Example:      list - 100, 101, 102, 103, 104, 105
//              newList - 105, 104, 103, 102, 101, 100, 101, 102, 103, 104, 105
public static int[] getBackwardForwards(int list[])
{
    int oLen = list.length;
    int newList[] = new int[2 * oLen - 1];

    /* missing code */

    return newList;
}

```

Which of the following implementations of /* missing code */ will make method method0503 work as intended?

<u>Implementation 1</u>	<u>Implementation 2</u>	<u>Implementation 3</u>
<pre> int k = oLen; for(int j = 0; j < oLen; j++) { newList[j] = list[k]; k--; } k = 1; for(int j = oLen; j < 2*oLen; j++) { newList[j] = list[k]; k++; } </pre>	<pre> int k = oLen - 1; for(int j = 0; j < oLen; j++) { newList[j] = list[k]; k--; } k = 1; for(int j = oLen; j < 2*oLen-1; j++) { newList[j] = list[k]; k++; } </pre>	<pre> int k = oLen - 1; for (int j = 0; j < oLen; j++) { newList[j] = list[k]; k--; } k = 0; for (int j = oLen; j < 2*oLen-1; j++) { newList[j] = list[k]; k++; } </pre>

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementation 1 & 2
- (E) Implementation 1 & 3

15. Consider the following method.

```
// precondition: listA and listB are two arbitrary arrays of integers.  
// postcondition: true is returned if the 2 lists are identical, false otherwise.  
//                 To be identical, the lists must have the same size and the same  
//                 contents at every index.  
public static boolean identical(int listA[], int listB[])  
{  
    boolean same = true;  
  
    /* missing code */  
  
    return same;  
}
```

Which of the following implementations of /* missing code */ will make method `identical` work as intended?

Implementation 1

```
int j = 0;  
while (same && j < listA.length)  
{  
    if (listA[j] != listB[j])  
        same = false;  
    j++;  
}
```

Implementation 2

```
if (listA.length != listB.length)  
    return false;  
int j = 0;  
while (same && j < listA.length)  
{  
    if (listA[j] != listB[j])  
        same = false;  
    j++;  
}
```

Implementation 3

```
for (int j = 0; j < listB.length; j++)  
{  
    if (listA[j] != listB[j])  
        same = false;  
    j++;  
}
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementation 1 & 2
- (E) Implementation 1 & 3

Chapter 6 Review

Static Java 2D Arrays

The fundamental logic of Java two-dimensional static arrays is the same as one-dimensional arrays. Two index values are used to indicate the row and column values. The program in **Example 6.1** is the typical approach used with 2D arrays. Like the earlier 1D arrays, loop control structures are used. In the case of 2D arrays two nested loops are needed. An outer loop controls the rows and the inner loop controls the columns of the 2D array.

There are now a double set of index operators. The first set of index operators specifies the number of rows in the array and the second set of index operators specify the number of columns.

Example 6.1

```
int[][] twoD = new int[2][3];
int count = 1;
for (int row = 0; row < 2; row++)
{
    for (int col = 0; col < 3; col++)
    {
        twoD[row][col] = count;
        count++;
    }
}

for (int row = 0; row < 2; row++)
{
    for (int col = 0; col < 3; col++)
    {
        System.out.print(twoD[row][col] + "   ");
    }
    System.out.println();
}
```

The **System.out.println()** statement in the nested loop structure is significant. The inner loop uses a **print** method, which will place each array value on the same line. The outer loop calls **println** and insures that the next row is displayed on the next line. The result is that the 2D array looks like a two-dimensional matrix.

The code segment in **Example 6.2** demonstrates how to declare a two-dimensional array using an initializer list

Example 6.2

```
int[][] square = {{1,2,3},{4,5,6},{7,8,9}};
```

It is not required that an initializer list is written in a two-dimensional matrix format. However, this is the common convention and it adds readability to a program. **Example 6.3** repeats the same declaration shown in **Example 6.2**, but the format is done in a matrix style.

Example 6.3

```
int[][] square = {{1,2,3},  
                  {4,5,6},  
                  {7,8,9}};
```

The declaration in Example 6.4 may not make sense, but it is used in a special program that involves a chess board. The motivation of the program logic is not important here. The key point is that the matrix display is much clearer than a rather messy long string that uses a single line approach.

Example 6.4

```
final private int ACCESS[][] = {{0,0,0,0,0,0,0,0,0,0,0},  
                               {0,0,0,0,0,0,0,0,0,0,0},  
                               {0,0,2,3,4,4,4,4,3,2,0,0},  
                               {0,0,3,4,6,6,6,6,4,3,0,0},  
                               {0,0,4,6,8,8,8,8,6,4,0,0},  
                               {0,0,4,6,8,8,8,8,6,4,0,0},  
                               {0,0,4,6,8,8,8,8,6,4,0,0},  
                               {0,0,4,6,8,8,8,8,6,4,0,0},  
                               {0,0,3,4,6,6,6,6,4,3,0,0},  
                               {0,0,2,3,4,4,4,4,3,2,0,0},  
                               {0,0,0,0,0,0,0,0,0,0,0},  
                               {0,0,0,0,0,0,0,0,0,0,0}};
```

You have to take a second look at the **length** field of the static array. In particular, there is a special issue when **length** is used with 2d static arrays. Start by looking at the program in **Example 6.5**. A small 2D array of integers is used and the values are displayed uses the **length**

field. A quick glance at the program and its output give the impression that everything is correct and there are no problems.

Example 6.5

```
public class Example65
{
    public static void main (String args[])
    {
        int[][] square = {{1,2,3},{4,5,6},{7,8,9} };
        for (int row = 0; row < square.length; row++)
        {
            for (int col = 0; col < square.length; col++)
                System.out.print(square[row][col] + " ");
            System.out.println();
        }
        System.out.println();
    }
}
```

Program Output

```
1 2 3
4 5 6
7 8 9
```

In **Example 6.5** the **length** field of the **square** array is used twice. First, it is used to determine the number of rows and then in the inner loop **length** is used to specify the number of cols. Now this works fine, but keep in mind that the array used is a square matrix with three rows and three columns. It may well be that this approach is not as easy with 2D arrays that are not square.

Program **Example 6.6** is similar to the previous program, but now the "**square**" array is no longer square. It is a matrix of two rows and three columns. With the same approach, using the **length** field, the output now is no longer correct.

Example 6.6

```
public class Example66
{
    public static void main (String args[])
    {
        int[][] square = {{1,2,3},{4,5,6}};
        for (int row = 0; row < square.length; row++)
        {
            for (int col = 0; col < square.length; col++)
                System.out.print(square[row][col] + " ");
            System.out.println();
        }
        System.out.println();
    }
}
```

Example 6.6 Continued

Program Output

```
1 2  
4 5
```

The **length** field makes more sense if a two-dimensional array is looked upon as an array of arrays. We have already seen that the **length** field correctly stores the row value for the 2D array. Now what about the column value? If a 2D array is in fact an array of arrays, then **length** will work as long as the correct array is specified.

This is precisely what is done in the next program example. Look at program **Example 6.7**. It has the same 2D array of two rows and three columns. In the outer loop **square.length** is used. Now remember that a 2D array is a 1D array of 1D arrays. How many rows or 1D arrays are there? Exactly, there are two rows and that is accurately reflected by **square.length**. Now to get the proper size of the columns we need to look at one of the "inner 1D arrays" of the **square** matrix. Consider **square[0]**, which does not use a double index, but a single one. **square[0]** is the first row or first array of **square** and that means that **square[0].length** specified the number of columns in the matrix.

Example 6.7

```
public class Example67
{
    public static void main (String args[])
    {
        int[][] square = {{1,2,3},{4,5,6}};
        for (int row = 0; row < square.length; row++)
        {
            for (int col = 0; col < square[0].length; col++)
                System.out.print(square[row][col] + " ");
            System.out.println();
        }
    }
}
```

Program Output

```
1 2 3  
4 5 6
```

The program in **Example 6.8** declares a two-dimensional array of 7 rows and 5 columns. It is another example using the **length** field with a two-dimensional array.

Example 6.8

```
public class Example68
{
    public static void main(String args[])
    {
        int k = 1;
        int matrix[][] = new int[7][5];
        for (int r = 0; r < matrix.length; r++)
            for (int c = 0; c < matrix[0].length; c++)
            {
                matrix[r][c] = k;
                k++;
            }

        for (int r = 0; r < matrix.length; r++)
        {
            for (int c = 0; c < matrix[0].length; c++)
                System.out.print(matrix[r][c]) + " ";
            System.out.println();
        }
    }
}
```

The length Field for Two-Dimensional Arrays

Consider the following statement:

```
int matrix[][] = new int[5][4];
```

The value of **matrix.length** is 5, which is the number of rows.

The value of **matrix[0].length** is 4, which is the number of columns

Since a Java static two-dimensional array is really an array of arrays. It is possible to declare only the size of the rows. In later statements the size of each column is declared, and in this case each column is a different size. This type of array is not rectangular, as the output shows, and is called a *ragged array*.

Example 6.9

```
public class Example69
{
    public static void main(String args[])
    {
        int x[][] = new int[5][];
        x[0] = new int[1];
        x[1] = new int[2];
        x[2] = new int[3];
        x[3] = new int[4];
        x[4] = new int[5];

        for (int r = 0; r < x.length; r++)
        {
            for (int c = 0; c < x[r].length; c++)
                System.out.print(x[r][c] + " ");
            System.out.println();
        }
    }
}
```

Program Output

```
0
0 0
0 0 0
0 0 0 0
0 0 0 0 0
```

APCS Examination Alert

Java static arrays can declare *ragged arrays*. This type of declaration is not part of the tested subset on the APCS Exam.

Static Java Array Pitfalls

Static Java 2D arrays are mostly straight forward, like 1D array. Like 1D arrays the primary concern is avoiding an index that is outside the index range. Each array has **length** array elements, but the largest index of the array is **length-1**. The program in **Example 6.10** compiles without difficulties, but generates an **ArrayIndexOutOfBoundsException** error during program execution.

Example 6.10

```
public class Example610
{
    public static void main (String args[])
    {
        int[][] square = {{1,2,3},{4,5,6}};
        for (int row = 1; row <= square.length; row++)
        {
            for (int col = 1; col <= square[0].length; col++)
                System.out.print(square[row][col] + " ");
            System.out.println();
        }
    }
}
```

Program Output

```
5 6 Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 3
at Example610.main(Example610.java:9)

Process completed.
```

Java Static Arrays Alert

An array declared in the following manner:

```
int[][] matrix = new int[maxRow][maxCol];
```

has **maxRow** elements in each row, but the index range is **[0..maxRow-1]**,
and has **maxCol** elements in each column, but the index range is **[0..colRow-1]**.

Any attempt to access an index outside the proper range will result in an **ArrayIndexOutOfBoundsException** error during program execution.

Chapter 6 Questions

Java Static 2D Arrays

01. Consider the following declaration.

```
double[][] mambo;
mambo = new double[4][5];
int r; // row index on mambo
int c; // column index of mambo
```

Which of the following statements stores the number of rows of `mambo`?

- (A) `mambo.length`
- (B) `mambo.rowLength`
- (C) `mambo[r].length`
- (D) `mambo[c].length`
- (E) `mambo[r][c].length`

02. Consider the following declaration.

```
double[][] mambo;
mambo = new double[4][5];
int r; // row index on mambo
int c; // column index of mambo
```

Which of the following statements stores the number of columns of `mambo`?

- (A) `mambo.length`
- (B) `mambo.rowLength`
- (C) `mambo[r].length`
- (D) `mambo[c].length`
- (E) `mambo[r][c].length`

03. Consider the following code segment and two methods.

```

int square[][] = new int[5][5];
createSquare(square);
displaySquare(square);

public static void createSquare(int[][] square)
{
    int size = square.length;
    for (int r = 0; r < size; r++)
    {
        int q = 1;
        for (int c = r; c < size; c++)
        {
            square[r][c] = q;
            q++;
        }
    }
}

public static void displaySquare(int[][] square)
{
    for (int r = 0; r < 5; r++)
    {
        for (int c = 0; c < 5; c++)
            System.out.print(square[r][c] + "   ");
        System.out.println();
    }
}

```

What is printed as a result of executing the code segment?

(A) 0 1 2 3 4 1 2 3 4 5 2 3 4 5 6 3 4 5 6 7 4 5 6 7 8	(B) 1 2 3 4 5 0 1 2 3 4 0 0 1 2 3 0 0 0 1 2 0 0 0 0 1	(C) 0 0 0 0 1 0 0 0 1 2 0 0 1 2 3 0 1 2 3 4 1 2 3 4 5
(D) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25	(E) 17 24 1 8 15 23 5 7 14 16 4 6 13 20 22 10 12 19 21 3 11 18 25 2 9	

04. Consider the following code segment and two methods.

```

int square[][] = new int[5][5];
createSquare(square);
displaySquare(square);

public static void createSquare(int[][] square)
{
    int size = square.length;
    for (int r = 0; r < size; r++)
    {
        int q = r;
        for (int c = 0; c < size; c++)
        {
            square[r][c] = q + c;
        }
    }
}

public static void displaySquare(int[][] square)
{
    for (int r = 0; r < 5; r++)
    {
        for (int c = 0; c < 5; c++)
            System.out.print(square[r][c] + "   ");
        System.out.println();
    }
}

```

What is printed as a result of executing the code segment?

(A) 0 1 2 3 4 1 2 3 4 5 2 3 4 5 6 3 4 5 6 7 4 5 6 7 8	(B) 1 2 3 4 5 0 1 2 3 4 0 0 1 2 3 0 0 0 1 2 0 0 0 0 1	(C) 0 0 0 0 1 0 0 0 1 2 0 0 1 2 3 0 1 2 3 4 1 2 3 4 5
(D) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25	(E) 17 24 1 8 15 23 5 7 14 16 4 6 13 20 22 10 12 19 21 3 11 18 25 2 9	

05. Consider the following code segment and two methods.

```
int square[][] = new int[5][5];
createSquare(square);
displaySquare(square);

public static void createSquare(int[][] square)
{
    int size = square.length;
    int r = 0;
    int c = size / 2;
    square[r][c] = 1;
    for (int k = 2; k <= size*size; k++)
    {
        if (k % size == 1)
            r++;
        else
        {
            r--;
            c++;
        }
        if (r < 0)
            r = size-1;
        if (c == size)
            c = 0;
        square[r][c] = k;
    }
}

public static void displaySquare(int[][] square)
{
    for (int r = 0; r < 5; r++)
    {
        for (int c = 0; c < 5; c++)
            System.out.print(square[r][c] + "   ");
        System.out.println();
    }
}
```

What is printed as a result of executing the code segment?

(A) 0 1 2 3 4 1 2 3 4 5 2 3 4 5 6 3 4 5 6 7 4 5 6 7 8	(B) 1 2 3 4 5 0 1 2 3 4 0 0 1 2 3 0 0 0 1 2 0 0 0 0 1	(C) 0 0 0 0 1 0 0 0 1 2 0 0 1 2 3 0 1 2 3 4 1 2 3 4 5
(D) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25	(E) 17 24 1 8 15 23 5 7 14 16 4 6 13 20 22 10 12 19 21 3 11 18 25 2 9	

06. Consider the following code segment.

```
int[][] square = {{1,2,3},{4,5,6}};
for (int row = 0; row < square.length; row++)
{
    for (int col = 0; col < square.length; col++)
        System.out.print(square[row][col] + " ");
    System.out.println();
}
```

What is printed as a result of executing the code segment?

- (A) 1 2 3 4 5 6
- (B) 1 2 3
4 5 6
- (C) `ArrayOutOfBoundsException` error
- (D) 1 2
4 5
- (E) 2 3
5 6

07. Consider the following code segment.

```
int[][] square = {{1,2,3},{4,5,6}};
for (int row = 0; row < square.length; row++)
{
    for (int col = 0; col < square[row].length; col++)
        System.out.print(square[row][col] + " ");
    System.out.println();
}
```

What is printed as a result of executing the code segment?

- (A) 1 2 3 4 5 6
- (B) 1 2 3
4 5 6
- (C) `ArrayOutOfBoundsException` error
- (D) 1 2
4 5
- (E) 2 3
5 6

08. Consider the following code segment.

```
int[][] square = {{1,2,3},{4,5,6},{7,8,9}};
int size = square.length;
for (int row = 0; row < size; row++)
{
    for (int col = 0; col < square[size].length; col++)
        System.out.print(square[row][col] + " ");
    System.out.println();
}
```

What is printed as a result of executing the code segment?

- (A) 1 2 3 4 5 6 7 8 9
- (B) 1 2 3
4 5 6
7 8 9
- (C) `ArrayOutOfBoundsException` error
- (D) 1 4 7
2 5 8
3 6 9
- (E) 9 8 7
6 5 4
3 2 1

09. Consider the following program segment.

```
int matrix[][] = {{1,2,3},{4,5,6},{7,8,9}};
int n = matrix.length;
for (int p = 0; p < n; p++)
    System.out.print(matrix[p][n-p-1] + " ");
```

What will be output when the program segment executes?

- (A) 7 5 3
- (B) 3 5 7
- (C) 1 5 9
- (D) 9 5 1
- (E) 3 6 9

10. Consider the following program segment.

```
int n = <some int > 0
int[][] matrix = new int [n][n];
int[][] matrix = {{12,34,25},{87,21,19},{56,31,99}};
for (int col = 0; col < n; col++)
    for (int row = 1; row < n; row++)
        matrix[row][col] = matrix[row-1][col];
```

Which of the following describes the values stored in matrix after the segment executes?

- (A) All the values in the top row are zero.
- (B) All the values in the entire matrix are zero.
- (C)** All the values in the entire matrix are the same.
- (D) All the values in each column are the same.
- (E) All the values in each row are the same.

11. Consider the following method.

```
public static void method0611(int matrix[][], int r, int c)
// Precondition: All element values of m are non-zero.
//                 0 <= r < matrix.length
//                 0 <= c < matrix[0].length
{
    for (int row = 0; row < matrix.length; row++)
        for (int col = 0; col < matrix[0].length; col++)
            if (row == r || col == c)
                matrix[row][col] = 0;
}
```

Which of the following describes the values stored in matrix after method0511 is called?

- (A) All the values in matrix are zero.
- (B)** All the values in row r and also in col c are zero.
- (C) All the values in row c and also in col r are zero.
- (D) Only the value of matrix[r][c] is zero.
- (E) All the values in matrix are unchanged.

12. Consider the following program segment.

```
int x = <some int value greater than zero>
int y = <some int value greater than zero>
int matrix[][] = new int[x][y];
```

Segment 1

```
for (int row = 0; row < matrix[0].length; row++)
    for (int col = 0; col < matrix.length; col++)
        matrix[row][col] = 100;
```

Segment 2

```
for (int row = 0; row < matrix.length; row++)
    for (int col = 0; col < matrix[row].length; col++)
        matrix[row][col] = 100;
```

Segment 3

```
for (int row = 0; row < matrix.length; row++)
    for (int col = 0; col < matrix.length; col++)
        matrix[row][col] = 100;
```

Which of the following program segments assigns 100 to each matrix element?

- (A) Segment 1 only
- (B) Segment 2 only
- (C) Segment 3 only
- (D) Segments 1 & 2 only
- (E) Segments 2 & 3 only

Questions 13-15 refer to the following code segment.

```
int size = <some integer greater than 2> ;  
  
int pascal[][] = new int[size][size];  
boolean assembly = true;  
for (int r = 0; r < size; r++)  
    for (int c = 0; c < size; c++)  
    {  
        if (assembly)  
            pascal[r][c] = 1;  
        else  
            pascal[r][c] = 2;  
        assembly = !assembly;  
    }  
  
for (int r = 0; r < size; r++)  
{  
    for (int c = 0; c < size; c++)  
        if (pascal[r][c] == 1)  
            System.out.print(" ");           // one blank space  
        else  
            System.out.print((char) 219); // one solid square  
    System.out.println();  
}
```

13. When will this code segment display a checkerboard/chessboard pattern?

- (A) When `size` is even
- (B)** When `size` is odd
- (C) When `size` is greater than 10
- (D) Always
- (E) Never

14. When will this code segment display vertical bars?

- (A)** When `size` is even
- (B) When `size` is odd
- (C) When `size` is greater than 10
- (D) Always
- (E) Never

15. When will this code segment display horizontal bars?

- (A) When `size` is even
- (B)** When `size` is odd
- (C) When `size` is greater than 10
- (D) Always
- (E) Never

Chapter 7 Review

The String Class

Java uses the **String** class and the **StringBuffer** class to manipulate strings of characters. Only the **String** class and its methods will be tested on the APCS Exam. String processing is important in many areas of data processing. Word processing programs are the most common applications that utilize string manipulation features.

APCS Examination Alert

There are two string manipulation classes in Java, which are the **String** class and the **StringBuffer** class.

Only the **String** class is tested on the APCS Exam.

The primary challenge with the Java **String** class is to know the available methods and understand the required parameters for correct string processing. The program in **Example 7.1** starts with a demonstration of the **length()** method. It is easy to get confused here. The Java static array has a **length** field, and the **String** class has a **length()** method. **Example 7.1** also shows that the plus (+) operator performs *concatenation* with **String** objects.

Example 7.1

```
public class Example71
{
    public static void main (String args[])
    {
        String s1 = "Argentine";
        String s2 = "Tango";
        String s3 = s1 + " " + s2;
        System.out.println("s1 length: " + s1.length());
        System.out.println("s2 length: " + s2.length());
        System.out.println("s3 length: " + s3.length());
        System.out.println();
    }
}
```

Program Output

```
s1 length: 9
s2 length: 5
s3 length: 15
```

String Class Method length

The **String** class uses the **length()** method to return the number of characters in a **String** object.

Note that the Java static array uses a **length** field and the Java **String** class uses a **length** method.

The **String** class has two **substring** methods. The more common version is shown in program **Example 7.2**, which uses two parameters. The first parameter is the index of the first character of the returned substring. The second parameter is the upper boundary of the substring. It is not the last character of the substring, but the next character. Look at the program output and check the substrings, which are returned, with each iteration through the loop.

Example 7.2

```
public class Example72
{
    public static void main (String args[])
    {
        String str = "Racecar";
        int n = str.length();
        for (int k = 1; k <= n; k++)
            System.out.println(str.substring(0,k));
        System.out.println();
        for (int k = 0; k <= n-3; k++)
            System.out.println(str.substring(k,k+3));
    }
}
```

Program Output

```
R
Ra
Rac
Race
Racec
Raceca
Racecar

Rac
ace
cec
eca
car
```

The **substring** method is overloaded. There also is a version with a single parameter. This single parameter is the starting index of the substring. There is no information required for the end index, since this method always goes from start-index to the end of the string object. The program in **Example 7.3** has a loop counter starting at **0** and ending at **n-1**. The resulting output displays several substrings starting with the entire string and ending with a single character.

Example 7.3

```
public class Example73
{
    public static void main (String args[])
    {
        String str = "ABCDEFGH";
        int n = str.length();
        for (int k = 0; k < n; k++)
            System.out.println(str.substring(k));
    }
}
```

Program Output

```
ABCDEFGH
BCDEFGH
CDEFGH
DEFGH
EFGH
FGH
GH
H
```

String Class Method substring

The **substring** method is overloaded.

String substring(int a, int b)

returns a substring starting at index **a** and ending at index **b-1**

String substring(int a)

returns a substring starting at index **a** and ending at index **length()-1**

The next **String** method to be considered is **indexOf**, which is a practical method to find the occurrence of one string as a substring in a second string. The program in **Example 7.4** demonstrates how the index value is returned if a substring is found and how **-1** is returned if the substring does not exist.

Example 7.4

```
public class Example74
{
    public static void main (String args[])
    {
        String s1 = "racecar";
        String s2 = "car";
        String s3 = "qwerty";
        int index1 = s1.indexOf(s2);
        int index2 = s1.indexOf(s3);
        System.out.println(s2 + " starts at " + index1 + " in " + s1);
        System.out.println(s3 + " starts at " + index2 + " in " + s1);

    }
}
```

Program Output

```
car starts at 4 in racecar
qwerty starts at -1 in racecar
```

String Class Method **indexOf**

int indexOf(String str)

returns the index of the first occurrence of **str**.
returns **-1** if **str** is not found.

The last **String** method to be considered is **compareTo**, which allows the comparison of different strings. This is the type of method that makes it possible to sort a list of string value in ascending or descending order.

Example 7.5

```
public class Example75
{
    public static void main(String args[])
    {
        String s1 = "racecar";
        String s2 = "CRUISESHIP";
        String s3 = "racecar";
        System.out.println(s1.compareTo(s2)); // s1 > s2
        System.out.println(s1.compareTo(s3)); // s1 == s3
        System.out.println(s2.compareTo(s1)); // s2 < s1
    }
}
```

Program Output

```
47
0
-47
```

String Class Method compareTo

int compareTo(String other)

returns a value < 0 if this is less than other.

returns 0 if this is equal to other.

returns a value > 0 if this is greater than other.

APCS Examination Alert

The **String** class has many methods to handle string processing.

Only **length**, **substring**, **indexOf** and **compareTo** are tested on the APCS Exam.

String Class Pitfalls

A common **String** pitfall was addressed earlier in this chapter. It is very easy to use the **substring** method - with two parameters - incorrectly. Students frequently think that a method call statement like...

str.substring(a,b)

...will return a substring that starts at index **a** and ends at index **b**. Remember that the second parameter is a boundary. This means that a substring is returned that starts at index **a** and ends at index **b-1**.

String objects do have a subtle problem that can cause logic errors. Consider the program in **Example 6.5**. **String** objects **s1** and **s2** are compared for equality. The program output states that the objects are not equal. This seems counter to what many students might expect with two objects, both storing **Foxtrot**, staring them in the face.

Example 7.6

```
public class Example76
{
    public static void main (String args[])
    {
        String s1 = new String("Foxtrot");
        String s2 = new String("Foxtrot");

        if (s1 == s2)
            System.out.println(s1 + " equals " + s2);
        else
            System.out.println(s1 + " does not equal " + s2);
    }
}
```

Program Output

```
Foxtrot does not equal Foxtrot
```

The problem is that **s1** and **s2** are objects of a class and not variables of a primitive type, like **int**. Only variables of primitive types can be compared using the **==** operator. What happens in **Example 7.6** is that the immediate (shallow) value of **s1** is a memory reference and the same is true of **s2**. Both objects are instantiated with a different memory reference. It is true that the values at the memory reference, **Foxtrot** and **Foxtrot**, are identical. However, the comparison made with the **==** operator does not see the deeper values.

The program in **Example 7.7** solves the comparison problem. Equality can be tested with **String** objects, but it must be done with the **equals** method, which is re-defined for the **String** class.

Example 7.7

```
public class Example76
{
    public static void main (String args[])
    {
        String s1 = new String("Foxtrot");
        String s2 = new String("Foxtrot");

        if (s1.equals(s2))
            System.out.println(s1 + " equals " + s2);
        else
            System.out.println(s1 + " does not equal " + s2);
    }
}
```

Program Output

```
Foxtrot equals Foxtrot
```

String Objects Equality

The **==** operator can only be used with primitive data types like **int**, **double**, **char** and **boolean**.

The **equals** method is required with **String** objects when testing equality using the program statement like:

if (s1.equals(s2))

Chapter 7 Questions

The String Class

01. Consider the following program segment.

```
String s1 = new String( "AARDVARK" );
String s2 = new String( "AARDVARK" );
String s3 = s1;
if (s1 == s2)
    System.out.println("equals");
else
    System.out.println("not equals");
if (s1 == s3)
    System.out.println("equals");
else
    System.out.println("not equals");
```

What will be output when the program segment executes?

- (A) equals
equals
- (B) not equals
not equals
- (C) equals
not equals
- (D) not equals
equals
- (E) Some exception message indicating improper string handling

02. Consider the following program segment.

```
String s1 = new String("AARDVARK");
String s2 = new String("AARDVARK");
String s3 = s1;
if (s1.equals(s2))
    System.out.println("equals");
else
    System.out.println("not equals");
if (s1.equals(s3))
    System.out.println("equals");
else
    System.out.println("not equals");
```

What will be output when the program segment executes?

- (A) equals
equals
- (B) not equals
not equals
- (C) equals
not equals
- (D) not equals
equals
- (E) Some exception message indicating improper String handling

03. Consider the following program segment.

```
String s1 = "sunshine";
String s2 = "shin";
System.out.println(s1.indexOf(s2));
```

What will be output when the program segment executes?

- (A) 0
- (B) 2
- (C) 3
- (D) 4
- (E) 5

04. Consider the following program segment.

```
String s1 = "racecar";
String s2 = s1.substring(1,4);
System.out.println(s2);
```

What will be output when the program segment executes?

- (A) ace
- (B) race
- (C) acec
- (D) racec
- (E) aceca

05. Consider the following program segment.

```
String s1 = "ABCD";
String s2 = "";
for (int k = 1; k < s1.length(); k++)
    s2 += s1.substring(0,k);
System.out.println(s2);
```

What will be output when the program segment executes?

- (A) AAA
- (B) ABC
- (C) AABABC
- (D) ABCCCD
- (E) AABABCABCD

06. Consider the following program segment.

```
String s1 = "ABCD";
String s2 = "";
for (int k = 1; k < s1.length(); k++)
    s2 += s1.substring(k);
System.out.println(s2);
```

What will be output when the program segment executes?

- (A) D
- (B) CD
- (C) CDD
- (D) BCDCDD
- (E) ABCDBCDCDD

07. Consider the following program segment.

```
String s1 = "FOX JUMPS";
String s2 = "The quick brown fox jumps over the lazy dog";
String s3 = "THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG";
System.out.print(s2.indexOf(s1) + " ");
System.out.println(s3.indexOf(s1));
```

What will be output when the program segment executes?

- (A) -1 17
- (B) 17 17
- (C) 16 -1
- (D) 16 16
- (E) -1 16

08. Consider the following method.

```
// precondition: s does not equal null
public static String method0708(String s)
{
    String t = "";
    for (int k = s.length()-1; k >= 0; k--)
        t += s.substring(k,k+1);
    return t;
}
```

The call `method0708("RACECAR")` returns the string RACECAR.

What string value is returned by the call `method0708("MADAM I'M ADAM")` ?

- (A) MADAM
- (B) MADAM I'M ADAM
- (C) MADAM IM ADAM
- (D) MADA M'I MADAM
- (E) MADAMIMADAM

09. Consider the following method.

```
// precondition: s is a string with one or more characters
public static String method0709(String s)
{
    String t = "";
    for (int k = s.length()-1; k >= 0; k--)
        t += s.substring(s.length()-1,s.length());
    return t;
}
```

What string value of t is returned by a call to method0709 ?

- (A) t returns the reverse string of s
- (B) t returns the same string as s
- (C) t returns a string composed of one or more first characters of s
- (D) t returns a string composed of one or more last characters of s
- (E) t returns the null string

10. Consider the following method.

```
// precondition: s is a string with one or more characters
public static boolean method0710(String s)
{
    String t = "";
    for (int k = s.length()-1; k >= 0; k--)
        t += s.substring(k,k+1);
    return s.equals(t);
}
```

For which values of s does method method0710 return true ?

- (A) Any string with only lower-case letters
- (B) Any string with only upper-case letters
- (C) For all string values of s
- (D) For no string values of s
- (E) For all string values of s that are "palindromes" (a string that reads the same backwards)

11. Consider the following two methods.

```
// precondition: s does not equal null
public static String method0711a(String s)
{
    return s.substring(1);
}

public static String method0711b(String s)
// precondition: s does not equal null
{
    return s.substring(1,s.length());
}
```

For which values of `s` is the same value returned by both methods?

- (A) Whenever `s` is a string with one character
- (B) For all string values of `s`
- (C) Whenever `s` has an odd number of characters
- (D) For no string values of `s`
- (E) For all string values of `s` that are "palindromes" (a string that reads the same backwards)

12. Consider the following two methods.

```
// precondition: s does not equal null
public static int method0712a(String s1, String s2)
{
    return s1.indexOf(s2);
}

public static int method0712b(String s1, String s2)
// precondition: s does not equal null
{
    return s2.indexOf(s1);
}
```

For which values of `s1` and `s2` is the same value returned by both methods?

- (A) Whenever the first character of `s1` is the same as the first character of `s2`
- (B) Whenever the last character of `s1` is the same as the last character of `s2`
- (C) For all string values of `s1` and `s2`
- (D) For no string values of `s1` and `s2`
- (E) Whenever `s1` is equal to `s2`

13. Consider the following code segment.

```
String s1 = "vark";
String s2 = "aardvark";
System.out.println(s1.compareTo(s2));
```

What type of value will be output when the program segment executes?

- (A) 0
- (B) an integer value > 0
- (C) an integer value < 0
- (D) aard
- (E) vark

14. Consider the following code segment.

```
String s1 = "MADAM";
String s2 = "madam";
System.out.println(s1.compareTo(s2));
```

What type of value will be output when the program segment executes?

- (A) 0
- (B) an integer value > 0
- (C) an integer value < 0
- (D) true
- (E) false

15. Consider the following code segment.

```
String s = "racecar";
String s1 = s.substring(4,7);
String s2 = s.substring(4);
System.out.println(s1.compareTo(s2));
```

What type of value will be output when the program segment executes?

- (A) 0
- (B) an integer value > 0
- (C) an integer value < 0
- (D) ecar
- (E) car

Chapter 8 Review

Number Systems

Computer science programming requires proficiency with mathematics that involves conversion between different numbers systems. Computer systems store information, and compute information in base-2. Human beings prefer to handle base-10 number systems. In the majority of computer applications, and even program language compilers, the conversion between base-2 and base-10 is handled quietly in the background by the application program or language compiler program.

APCS Examination Alert

The APCS Exam does not allow the use of calculators. It is important that you learn to do computations on paper. Number systems is one of the topics that is tested on the APCS Exam.

The table below reviews counting the first set of numbers in bases **10, 5, 8, 2** and **16**. Remember that the largest digit in any number system is one less than the base value. In base-16 it is necessary to use some new symbols like the letters [a..f] to indicate the single digits for values [11..15]. There are columns for numbers in base-5 and base-8. The three primary bases used in computer science are base-2, base-16 and base-10.

Base 10	Base 5	Base 8	Base 2	Base-16
0	0	0	0	0
1	1	1	1	1
2	2	2	10	2
3	3	3	11	3
4	4	4	100	4
5	10	5	101	5
6	11	6	110	6
7	12	7	111	7
8	13	10	1000	8
9	14	11	1001	9
10	20	12	1010	a
11	21	13	1011	b
12	22	14	1100	c
13	23	15	1101	d
14	24	16	1110	e
15	30	17	1111	f
16	31	20	10000	10

Conversion 8.1 through **conversion 8.5** demonstrate conversions from any base (up-to-16) to base-10. In each case the conversion is accomplished by using the place holding values of each digit, multiplying it times the digit number to get the value in base 10. **Conversion 8.4**, between base-2 and base-10, may look a little strange. The only possible values in base-2 are 1 and 0. It is not necessary to multiply times 1 or times 0. These steps are shown to display a consistency in the conversion steps, but base-2 conversions can be done without this step.

Conversion 8.1:

$$213 \text{ base-4} = ??? \text{ base-10}$$

$$\begin{aligned} 2 \times 4^2 + 1 \times 4^1 + 3 \times 4^0 &= \\ 2 \times 16 + 1 \times 4 + 3 \times 1 &= \\ 32 + 4 + 3 &= 39 \text{ base-10} \end{aligned}$$

Conversion 8.2:

$$2134 \text{ base-5} = ??? \text{ base-10}$$

$$\begin{aligned} 2 \times 5^3 + 1 \times 5^2 + 3 \times 5^1 + 4 \times 5^0 &= \\ 2 \times 125 + 1 \times 25 + 3 \times 5 + 4 \times 1 &= \\ 250 + 25 + 15 + 4 &= 294 \text{ base-10} \end{aligned}$$

Conversion 8.3:

$$175 \text{ base-8} = ??? \text{ base-10}$$

$$\begin{aligned} 1 \times 8^2 + 7 \times 8^1 + 5 \times 8^0 &= \\ 1 \times 64 + 7 \times 8 + 5 \times 1 &= \\ 64 + 56 + 5 &= 125 \text{ base-10} \end{aligned}$$

Conversion 8.4:

$$101101 \text{ base-2} = ??? \text{ base-10}$$

$$\begin{aligned} 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 &= \\ 1 \times 32 + 0 \times 16 + 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 &= \\ 32 + 0 + 8 + 4 + 0 + 1 &= 45 \text{ base-10} \end{aligned}$$

Conversion 8.5:

$$1fa \text{ base-16} = ??? \text{ base-10}$$

$$\begin{aligned} 1 \times 16^2 + f \times 16^1 + a \times 16^0 &= \\ 1 \times 256 + f \times 16 + a \times 1 &= \\ 1 \times 256 + 15 \times 16 + 10 \times 1 &= \\ 256 + 240 + 10 &= 506 \text{ base-10} \end{aligned}$$

It is very common for students to try and use the trusty *expansion* method to convert numbers from **base-10** to some other base. This may make sense until you look at the results and observe what you are doing. Suppose you want to convert the number **348 base-10** into a **base-5** number. Look what happens with the expansion method.

$$\begin{aligned}348 \text{ base-10} &= 3 \times 10^2 + 4 \times 10^1 + 8 \times 10^0 \\348 \text{ base-10} &= 3 \times 100 + 4 \times 10 + 8 \times 1 \\348 \text{ base-10} &= 300 + 40 + 8 = 348\end{aligned}$$

You see how you can very skillfully convert the **348** number with expansion right back into **348**. This is not a satisfactory result. A different approach is necessary. There are several different ways to convert a base-10 number to another base. One way involves dividing the base-10 number by the desired base value again and again until you get a quotient of 0. If you then take all of the remainders in reverse order, you will have your answer. This is shown with conversion 8.6 through conversion 8.8.

Conversion 8.6: Convert 201_{10} to binary (Base-2):

$$\begin{array}{rcl}201 & / & 2 = 100 \text{ remainder } 1 \\100 & / & 2 = 50 \text{ remainder } 0 \\50 & / & 2 = 25 \text{ remainder } 0 \\25 & / & 2 = 12 \text{ remainder } 1 \\12 & / & 2 = 6 \text{ remainder } 0 \\6 & / & 2 = 3 \text{ remainder } 0 \\3 & / & 2 = 1 \text{ remainder } 1 \\1 & / & 2 = 0 \text{ remainder } 1\end{array}$$

When the quotient is 0, the remainders in reverse order is the answer: $201_{10} = 11001001_2$

Conversion 8.7: Convert 1001_{10} to Base-5:

$$\begin{array}{rcl}1001 & / & 5 = 200 \text{ remainder } 1 \\200 & / & 5 = 40 \text{ remainder } 0 \\40 & / & 5 = 8 \text{ remainder } 0 \\8 & / & 5 = 1 \text{ remainder } 3 \\1 & / & 5 = 0 \text{ remainder } 1\end{array}$$

When the quotient is 0, the remainders in reverse order is the answer: $1001_{10} = 13001_5$

Conversion 8.8: Convert 506_{10} to hexadecimal (Base-16):

$$\begin{array}{r} & \overset{31}{\text{---}} \quad r \quad 10 \\ 16 \) & 506 \\ & \overset{48}{\text{---}} \\ & \overset{26}{\text{---}} \\ & \overset{16}{\text{---}} \\ & 10 \end{array}$$

$$\begin{array}{r} & \overset{1}{\text{---}} \quad r \quad 15 \\ 16 \) & 31 \\ & \overset{16}{\text{---}} \\ & 15 \end{array}$$

$$\begin{array}{r} & \overset{0}{\text{---}} \quad r \quad 1 \\ 16 \) & 1 \\ & \overset{0}{\text{---}} \\ & 1 \end{array}$$

Using the division method, when we take the remainders in reverse order we get $1(15)(10)$.

We need to do one extra step to make this right.

The **15** needs to be converted to **F**.

The **10** needs to be converted to **A**.

Now we can properly answer the question:

$$506_{10} = 1FA_{16}$$

Hex	Dec
A	10
B	11
C	12
D	13
E	14
F	15

There are some unique properties that exist between base-2 and base-16 that makes conversion possible without any arithmetic computation. Each single base-16 digit is equal to four base-2 digits. The table on the next page shows the sixteen base-16 single digits and their corresponding base-2 digits.

Base-2	Base-16	Base-2	Base-16
0000	0	1000	8
0001	1	1001	9
0010	2	1010	a
0011	3	1011	b
0100	4	1100	c
0101	5	1101	d
0110	6	1110	e
0111	7	1111	f

Conversion 8.9: 1001001010111 base-2 = ??? base-16						
1001001010111 = 0001 0010 0101 0111						
base-2			0001	0010	0101	0111
base-16			1	2	5	7
= 1257 base-16						

Conversion 8.10: 111111110010001101 base-2 = ??? base-16						
111111110010001101 = 0111 1111 1100 1000 1101						
base-2		0111	1111	1100	1000	1101
base-16		7	f	c	8	d
= 7fc8d base-16						

Conversion 8.11: 73254 base-16 = ??? base-2						
base-16		7	3	2	5	4
base-2		0111	0011	0010	0101	0100
= 0111 0011 0010 0101 0100 base-2						

Conversion 8.12: 1a2b3c base-16 = ??? base-2						
base-16	1	a	2	b	3	c
base-2	0001	1010	0010	1011	0011	1100
= 0001 1010 0010 1011 0011 1100 base-2						

Number System Pitfalls

Number Systems is a computer science topic that does not involve program examples. Common pitfalls are the problems with making simple arithmetic errors during conversion. Neatly write down your computations and keep track of each step.

Chapter 8 Questions

Number Systems

01. List the next 5 base-2 numbers after 11.

- (A) 12 13 14 15 16
- (B) 100 101 110 111 1000
- (C) 12 20 21 22 23
- (D) 100 110 111 1000 1100
- (E) 100 110 111 1110 1111

02. List the next 5 base-16 numbers after 98.

- (A) 99 100 101 102 103
- (B) 100 10a 10b 10c 10d
- (C) a9 b9 c9 d9 e9
- (D) 9a 9b 9c 9d 9e
- (E) 99 9a 9b 9c 9d

03. 2132 base-5 equals what number in base-10?

- (A) 292
- (B) 321
- (C) 242
- (D) 296
- (E) 1020

04. 10100 base-2 equals what number in base-10?

- (A) 40
- (B) 22
- (C) 20
- (D) 18
- (E) 16

05. 10101010 base-2 equals what number in base-10?

- (A) 144
- (B) 168
- (C) 196
- (D) 204
- (E) 170

06. 1cb base-16 equals what number in base-10?

- (A) 328
- (B) 428
- (C) 392
- (D) 459
- (E) 532

07. 254 base-10 equals what number in base-2?

- (A) 11111110
- (B) 10111010
- (C) 11010110
- (D) 10011110
- (E) 10100110

08. 683 base-10 equals what number in base-2?

- (A) 1011001101
- (B) 1010101010
- (C) 1101010101
- (D) 1010101011

09. 1000 base-10 equals what number in base-5?

- (A) 12444
- (B) 20024
- (C) 10030
- (D) 13000
- (E) 12298

10. 505 base-10 equals what number in base-16?

- (A) 1fb
- (B) 1f9
- (C) 26c
- (D) 2fb
- (E) 1ea

11. a1b2c3d4e5f6 base-16 equals what number in base-2?

- (A) 101011011011100011100111110110101110010111110100
- (B) 10101001111100011100111110101001110110111110100
- (C) 1010100100111000110101111101011010010111110110
- (D) 101000011011001011000011110101001110010111110110
- (E) 100100011001010111100010100010110111100010101101

12. 11111101101110010111010 base-2 equals what number in base-16?

- (A) b0ec5
- (B) 5ce0b
- (C) 7bf64
- (D) abcdef
- (E) fedcba

13. What number comes after 9999 in base-16?

- (A) 10000
- (B) AAAA
- (C) 999F
- (D) 999A
- (E) 9998

14. What is the largest 6 digit number in base-16?

- (A) 161616
- (B) 999999
- (C) 99999F
- (D) FFFFFF
- (E) FFFF9

15. 111111111111111111111111111111111111111111111111111111111111111 base-2 equals what number in base-16?
(There are 39 ones.)

- (A) 111,111,111,111,111,111,111,111,111,111,111,111,111,111,111,111
- (B) FFFFFFFFFF
- (C) 7FFFFFFFFF
- (D) FFFFFFFFFFA
- (E) 7FFFFFFF

Chapter 9 Review

The ArrayList Class

Java static arrays were shown in Chapter 05 and Chapter 06. Static arrays cannot be resized during program execution, and there are no methods available to manipulate static array objects. On the other hand, the biggest reason for using static arrays is the multi-dimension capability. The **ArrayList** class in this chapter can only handle arrays of one dimension.

The square index brackets used by static arrays for array element access will be replaced with parentheses that contain parameters for various methods. Objects of **ArrayList** object can be resized during program execution. Resizing may be to increase or decrease the number of elements in the array object.

The program in **Example 9.1** uses the **add** method to add additional members to the **list** object. It also shows the **size** method. Unfortunately, you now have a third keyword for handling data size. Static array use a **length** field. **String** object use a **length** method. **ArrayList** objects use a **size** method.

Data Structure Size Alert

Java static arrays use the **length** field.
The **String** class uses the **length** method.
The **ArrayList** class uses the **size** method.

Example 9.1

```
public class Example91
{
    public static void main(String args[])
    {
        ArrayList names = new ArrayList();
        names.add("Isolde");
        names.add("John");
        names.add("Greg");
        System.out.println(names);
        System.out.println("There are " + names.size() + " names.");
    }
}
```

Program Output

```
[Isolde, John, Greg]
There are 3 names.
```

The **add** method is overloaded. With a single parameter, the new array element is added to the end of the **ArrayList** object. It is also possible to add a new element at a specified index location. The program, in **Example 8.2** adds a fourth name at index 1. The former element at index 1, Sue in this case, is moved up one index value.

Example 8.2

```
public class Example82
{
    public static void main(String args[])
    {
        ArrayList names = new ArrayList();
        names.add("Bob");
        names.add("Sue");
        names.add("Tom");
        System.out.println(names);
        names.add(1,"Meg");
        System.out.println(names);
    }
}
```

Program Output

```
[Bob, Sue, Tom]
[Bob, Meg, Sue, Tom]
```

ArrayList Class Methods add and size

int size()

returns the number of elements in an **ArrayList** object

boolean add(E obj)

adds object to the end of the **ArrayList** object

E is the class of the Elements in the **ArrayList** object

boolean add(int index, E obj)

replaces the element at position index with obj

all elements with starting position index are shifted one index higher
index value outside the index range generates an **OutOfBoundsException**

ArrayList Class Pitfalls

The methods of the **ArrayList** class are not difficult to use, but there are a variety of problems that must be recognized. For starters the **ArrayList** shares the **IndexOutOfBoundsException** runtime error with the static array. Static or dynamic ... it does not matter ... you can only use index values of actual array elements. This error can occur with any method that accesses any array element by index number. The **ArrayList** class has several methods that access element by index values. The **get** method, which returns the object at the specified location, is one of the methods that is vulnerable to out of bounds errors.

Example 9.3 demonstrates how to use the **get** method. In this example the program will not compile, as the output cell indicates. The goal is to copy the elements of one array into another array. The **get** method returns an object when a **String** is needed. This is a problem of the Java versions prior to the release of Java 5.0. It is also a problem when you fail to specify the element class that will be stored in the **ArrayList** object.

Example 9.3

```
public class Example93
{
    public static void main(String args[])
    {
        ArrayList names1 = new ArrayList();
        ArrayList names2 = new ArrayList();
        names1.add("Bob");
        names1.add("Sue");
        names1.add("Tom");
        for (int k = 0; k < names1.size(); k++)
        {
            String temp = names1.get(k);
            names2.add(temp);
        }
    }
}
```

Program Output

```
C:\ExpoJava2007\ExpoJavaA08152007\ChapterProgsA2007\Progs20\Example93.java:21:
incompatible types
found   : java.lang.Object
required: java.lang.String
           String temp = names1.get(k);
                           ^
```

Java programs prior to the release of Java 5.0 handled the problem of **Example 9.3** by casting the class, (**String**) in this case, to prevent Java confusion. We now have a simpler approach. The **ArrayList** class is one of the *generic* classes and the type to be used can be specified when an **ArrayList** object is declared. The new approach is shown in **Example 9.4**. The next program example also shows how to use the **for..each** loop with an **ArrayList** object.

Example 9.4

```
public class Example94
{
    public static void main(String args[])
    {
        ArrayList<String> names1 = new ArrayList<String>();
        ArrayList<String> names2 = new ArrayList<String>();
        names1.add("Bob");
        names1.add("Sue");
        names1.add("Tom");
        for (int k = 0; k < names1.size(); k++)
        {
            String temp = names1.get(k);
            names2.add(temp);
        }
        for (String name: names2)
            System.out.println(name);
    }
}
```

Program Output

```
Bob
Sue
Tom
```

APCS Examination Alert

Generics is tested on APCS Exam.

You can expect multiple choice questions and free response questions that assume a knowledge of generics.

Generic classes eliminate the need for *class casting*, but *wrapper classes* are still used since *autoboxing* is not part of APCS Exam.

ArrayList Class Method get

E get(int index)

returns the element at position index

E is the declared type of the ArrayList elements

Program **Example 9.5** shows two additional methods, **set** and **remove**. Both methods are used in the same program. The **set** method replaces an existing element at the specified index with a new parameter object. There is a catch, because method **set** is a return method and it returns the element that formerly occupied the position. Look carefully at the output and you will see that **300**, the former element at index 2, is displayed. Method **remove** deletes an element at the specified index. It also is a return method which returns the very element that was just deleted. The pitfall here is that students forget that other elements are now shifted to a lower index.

Example 9.5

```
public class Example95
{
    public static void main(String args[])
    {
        ArrayList<Integer> numbers = new ArrayList<Integer>();
        for (int k = 100; k < 500; k+= 100)
            numbers.add(new Integer(k));
        System.out.println(numbers);
        System.out.println(numbers.set(2,new Integer(999)));
        System.out.println(numbers);
        numbers.remove(1) ;
        System.out.println(numbers);
    }
}
```

Program Output

```
[100, 200, 300, 400]
300
[100, 200, 999, 400]
[100, 999, 400]
```

ArrayList Class Methods **set** and **remove**

E **set** (int index, E obj)

replaces the element at position index with obj
returns the element that was formerly at the specified position

E **remove(int index)**

removes and returns the element at position index
shifts array element at position index+1 and higher to the left
returns the element that was deleted

Chapter 9 Questions

The ArrayList Class

01. Which of the following code segments correctly adds a new element to the `list` array?

Code Segment 1

```
ArrayList<String> list1 = new ArrayList<String>();
list1.add(new String("Java"));
```

Code Segment 2

```
ArrayList<Integer> list2 = new ArrayList<Integer>();
list2.add(new Integer(1000));
```

Code Segment 3

```
ArrayList<Double> list3 = new ArrayList<Double>();
list3.add(new Double(3.14159));
```

Code Segment 4

```
ArrayList<int> list4 = new ArrayList<int>();
list4.add(new Integer(500));
```

- (A) 1 and 2 only
- (B) 1 and 3 only
- (C) 2, 3 and 4 only
- (D) 1, 3 and 4 only
- (E) 1, 2 and 3 only

02. Consider the following code segment.

```
ArrayList<String> names = new ArrayList<String>();
names.add("Tom");
names.add("Ann");
names.add("Joe");
names.add("Sue");
names.add("Jan");
for (int k = 1; k < names.size(); k++)
    System.out.print(names.get(k) + " ");
```

What is printed as a result of executing the code segment?

- (A) IndexOutOfBoundsException message
- (B) Tom Ann Joe Sue Jan
- (C) Ann Joe Sue Jan
- (D) Tom Joe Sue Jan
- (E) Tom
Ann
Joe
Sue
Jan

03. Consider the following code segment.

```
ArrayList<String> names = new ArrayList<String>();
names.add("Astrid");
names.add("Leon");
names.add("Ingrid");
names.add("Remy");
for (int k = 0; k < names.size(); k++)
    System.out.print(names.set(k, "Isolde") + " ");
System.out.println();
for (int k = 0; k < names.size(); k++)
    System.out.print(names.get(k) + " ");
```

What is printed as a result of executing the code segment?

- (A) Isolde Isolde Isolde Isolde
Isolde Isolde Isolde Isolde
- (B) Astrid Leon Ingrid Remy
Isolde Isolde Isolde Isolde
- (C) Isolde Isolde Isolde Isolde
Astrid Leon Ingrid Remy
- (D) Isolde Isolde Isolde Isolde
- (E) IndexOutOfBoundsException message

04. Consider the following code segment.

```
ArrayList<String> names = new ArrayList<String>();
names.add("Tom");
names.add("Ann");
names.add("Joe");
names.add("Sue");
names.add(2, "Bob");
names.add(2, "Jan");
System.out.println();
for (String name: names)
    System.out.print(name + "   ");
```

What is printed as a result of executing the code segment?

- (A) Tom Ann Bob Jan Joe Sue
- (B) Tom Ann Jan Sue
- (C) Tom Ann Jan Bob Joe Sue
- (D) Tom Jan Bob Ann Joe Sue
- (E) IndexOutOfBoundsException message

05. Consider the following code segment.

```
ArrayList<String> animals = new ArrayList<String>();
animals.add("Giraffe");
animals.add("Tiger");
animals.add("Elephant");
animals.add("Wildebeest");
animals.add("Aardvark");
animals.remove(1);
animals.remove(3);
System.out.println();
for (String animal: animals)
    System.out.print(animal + "   ");
```

What is printed as a result of executing the code segment?

- (A) Giraffe Elephant Wildebeest
- (B) Tiger Wildebeest Aardvark
- (C) Giraffe Elephant Aardvark
- (D) Tiger Elephant Aardvark
- (E) Tiger Elephant Wildebeest

06. Consider the following code segment.

```
ArrayList<String> names = new ArrayList<String>();
names.add("Astrid");
names.add("Leon");
names.add("Ingrid");
names.add("Remy");
for (int k = 1; k < names.size(); k++)
    names.set(k,names.get(k-1));
System.out.println();
for (String name: names)
    System.out.print(name + "   ");
```

What is printed as a result of executing the code segment?

- (A) Leon Leon Leon Leon
- (B) Astrid Astrid Astrid Astrid
- (C) Leon Ingrid Remy
- (D) Remy Remy Remy Remy
- (E) IndexOutOfBoundsException message

07. Consider the following method.

```
public static double method0807(ArrayList<Integer> list)
{
    int p = 0;
    for (int k = 0; k < list.size(); k++)
    {
        Integer q = list.get(k);
        p += q.intValue();

    }
    return (double) p/list.size();
}
```

What is returned by method method0807 ?

- (A) The median value of list
- (B) The sum of list
- (C) The mean value of list
- (D) The list array with all integers converted to doubles
- (E) The list array with all Integer objects converted to int values

08. Consider the following `IntList` class with the incomplete `reverseList` method.

```
class IntList
{
    private ArrayList<Integer> intArray;
    private int max;

    public IntList(int n)
    {
        max = n;
        intArray = new ArrayList<Integer>();
        Random rnd = new Random();
        for (int k = 0; k < max; k++)
            intArray.add(new Integer(rnd.nextInt(100)));
    }

    public void reverseList()
    // postcondition: elements in intArray are reversed such that:
    //                 a0 = amax-1, amax-1 = a0, a1 = amax-2, amax-2 = a1
    {
        Integer temp;
        /* missing code */
    }
}
```

Which of the following implementations can be used to replace `/* missing code */` so that method `reverseList` will work as intended?

Implementation 1

```
for (int k = 0; k < max/2; k++)
{
    temp = intArray.set(k,intArray.get(max-1-k));
    intArray.set(max-1-k,temp);
}
```

Implementation 2

```
for (int k = 0; k < max/2; k++)
{
    temp = intArray.get(k);
    intArray.set(k,intArray.get(max-1-k));
    intArray.set(max-1-k,temp);
}
```

Implementation 3

```
for (int k = 0; k < max; k++)
{
    temp = intArray.set(k,intArray.get(max-1-k));
    intArray.set(max-1-k,temp);
}
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementations 1 and 2 only
- (E) Implementations 2 and 3 only

09. Consider the following `IntList` class with method `mystery`.

```
class IntList
{
    private ArrayList<Integer> intArray;
    private int max;

    public IntList(int n)
    {
        max = n;
        intArray = new ArrayList<Integer>();
        Random rnd = new Random();
        for (int k = 1; k <= max; k++)
            intArray.add(new Integer(rnd.nextInt(100)));
    }

    public void mystery()
    {
        for (int k = 0; k < max-1; k++)
        {
            int intVal1 = intArray.get(k);
            int intVal2 = intArray.get(k+1);
            if (intVal1 > intVal2)
            {
                Integer temp = intArray.get(k);
                intArray.set(k,intArray.get(k+1));
                intArray.set(k+1,temp);
            }
        }
    }
}
```

How are the values stored in `intArray` after a call to method `mystery`?

- (A) The values in `intArray` are reversed.
- (B) The largest value in `intArray` is the first element in the array.
- (C) The largest value in `intArray` is the last element in the array.
- (D) The values in `intArray` are sorted in ascending order.
- (E) The values in `intArray` are sorted in descending order.

10. Consider the `IntList` class with the missing `sortList` method.

```
class IntList
{
    private ArrayList<Integer> intArray;
    private int max;
    public IntList(int n)
    {
        max = n;
        intArray = new ArrayList<Integer>();
        Random rnd = new Random();
        for (int k = 1; k <= max; k++)
            intArray.add(new Integer(rnd.nextInt(100)));
    }

    // postcondition: intArray is sorted in ascending order.
    public void sortList()
    {
        for (int p = 1; p < max; p++)
            for (int q = 0; q < max-p; q++)
            {
                Integer integer1 = intArray.get(q);
                Integer integer2 = intArray.get(q+1);
                if (integer1.intValue() > integer2.intValue())
                {
                    /* missing code */
                }
            }
    }
}
```

Which of the following implementations can be used to replace `/* missing code */` so that method `sortList` will work as intended?

Implementation 1

```
intArray.set(q,intArray.get(q+1));
intArray.set(q+1,intArray.get(q));
```

Implementation 2

```
temp = intArray.get(q);
intArray.set(q,intArray.get(max-1-q));
intArray.set(max-1-q) = temp;
```

Implementation 3

```
Integer temp = intArray.get(q);
intArray.set(q,intArray.get(q+1));
intArray.set(q+1,temp);
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementations 1 and 2 only
- (E) Implementations 1 and 3 only

11. Consider the following Student class and code segment.

```
class Student
{
    private String name;
    private int age;
    public Student(String n, int a)
    {
        name = n;
        age = a;
    }
    public String getName() { return name; }
    public int getAge() { return age; }
}

ArrayList<Student> students = new ArrayList<Student>();
students.add(new Student("Sue", 25));
students.add(new Student("Tom", 23));
students.add(new Student("Ann", 20));
students.add(new Student("Joe", 18));
for (int k = 0; k < students.size(); k++)
{
    /* missing code */
}
```

Which of the following three implementations of **/* missing code */** will display the name and age field of each element in the students array?

Implementation 1

```
Student temp = students.get(k);
System.out.println(temp.getName() + " " + temp.getAge());
```

Implementation 2

```
System.out.println(students.get(k).getName() + " " + students.get(k).getAge());
```

Implementation 3

```
System.out.println(students.getName(k) + " " + students.getAge(k));
```

- (A) Implementation 1 only
- (B) Implementation 3 only
- (C) Implementations 1 and 2 only
- (D) Implementations 1 and 3 only
- (E) Implementations 1, 2 and 3

12. Consider the following Student class and code segment.

```
class Student
{
    private String name;
    private int age;
    public Student(String n, int a)
    {
        name = n;
        age = a;
    }
    public String toString()
    {
        return "[" + name + "," + age + "]";
    }
}

ArrayList<Student> students = new ArrayList<Student>();
students.add(new Student("Sue", 25));
students.add(new Student("Tom", 23));
students.add(new Student("Ann", 20));
students.add(new Student("Joe", 18));
for (Student student: students)
    System.out.println(student);
```

What will be output when the code segment executes?

- (A) Sue,25 Tom,23 Ann,20 Joe,18
- (B) [Sue,25]
[Tom,23]
[Ann,20]
[Joe,18]
- (C) [[Sue,25], [Tom,23], [Ann,20], [Joe,18]]
- (D) [[Sue,25]
[Tom,23]
[Ann,20]
[Joe,18]]
- (E) Sue,25
Tom,23
Ann,20
Joe,18

13. Which of the following code segments correctly defines list as an array data structure with the potential to store 1000 elements of any type?

Segment I

```
ArrayList list = new ArrayList();
```

Segment II

```
ArrayList[ ] list = new ArrayList[1000];
```

Segment III

```
ArrayList list = new ArrayList(1000);
```

- (A) Segment 1 only
- (B) Segment II only
- (C) Segments I and II only
- (D) Segments II and III only
- (E) Segments I, II and III

14. Compare the following two statements.

Statement 1: `ArrayList list1 = new ArrayList();`

Statement 2: `ArrayList<Integer> list2 = new ArrayList<Integer>();`

What can be concluded simply by observing the definitions of `list1` and `list2`:

- I. Both lists use generics
- II. Both lists can store Integer objects.
- III. Both lists currently are empty.
- IV. Both lists are dynamic.

- (A) I only
- (B) I and II only
- (C) III only
- (D) II and III only
- (E) II, III and IV only

15. Consider the following code segment:

```
ArrayList<Double> reals = new ArrayList<Double>();
reals.add(new Double(2.2));
reals.add(new Double(3.3));
reals.add(new Double(4.4));
```

Which of the following statements demonstrates the use of generics?

- I. double real = reals.get(0);
 - II. double real = (reals.get(0)).doubleValue();
 - III. double real = ((Double)reals.get(0)).doubleValue();
 - IV. double real = ((Double)reals.get(0));
-
- (A) I only
 - (B) I and II only
 - (C) II and III only
 - (D) III and IV only
 - (E) IV only

Chapter 10 Review

Recursion

The computer science concept of recursion is easily stated. *Recursion is the process whereby a method calls itself.* The actual understanding of recursive methods and creating recursive methods is another story. Please be aware that recursion is an important topic and will be tested.

APCS Examination Alert

Recursion is an important topic on the AP Computer Science Exam. The APCS Exam only requires that students can use and evaluate provided recursive methods. Students are not required to create original recursive solutions in Java source code.

The program in **Example 10.1** shows fundamental recursion. The **main** method calls method **count** and **count** calls itself as long as **k < 99**.

Example 10.1

```
public class Example101
{
    static int k = 10;

    public static void main(String args[])
    {
        count();
    }

    public static void count()
    {
        k++;
        System.out.print(k + " ");
        if (k % 10 == 0)
            System.out.println();
        if (k < 99)
            count();
    }
}
```

Example 10.1 Continued

Program Output

11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	

Example 10.1 is a comfortable example. Method **count** has no parameters and there are no other complexities to consider. This first example does demonstrate the first rule of recursion.

First Rule Of Recursion

All recursive methods require an *exit* or *base case* that stops the recursive process.

It does not take long to fall into a variety of recursive pitfalls. Some text books state that recursion is elegant and easier than iteration. There are certain complex problems where recursion is simpler than iteration, but that is not true for most situations. Once you accept that recursion is how a computer thinks and you devote time studying recursive examples, the concept becomes more manageable. It is natural that this computer science topic takes more exposure than most topics. With that idea in mind you need to look at the common pitfalls of recursion.

Recursion Pitfalls

The first common pitfall occurs easily with a method shown in the previous program example. **Example 10.2** shows the method only to observe the potential problem. The base case, which occurs when **k** is no longer less than 99 is easily misunderstood. Students look at the statement and conclude that the largest number displayed is **98**. Recursion stops when **k equals 99**. When **k equals 98** method **count** is called for the last time and **99** is printed before the method stops.

Example 10.2

```
public static void count()
{
    k++;
    System.out.print(k + "   ");
    if (k < 99)
        count();
}
```

In a typical method it is common to see a variable with a starting value. This type of statement in a recursive method will result in a computer crash. If you look at **Example 10.3** you will see a method that has a base case, but this exit is never reached, because the value of **k** is restarted each time that method **count** is called. It is true that the **k++;** statement increases **k** each time that the method is called, but it is equally true that the statement **int k = 0;** forces **k** to start over each time.

Example 10.3

```
public static void count()
{
    int k = 0;
    System.out.print(k + "   ");
    if (k < 99)
    {
        k++;
        count();
    }
}
```

Methods **count1** and **count2**, in **Example 10.4**, both display numbers in the $[a .. b]$ integer value range. Both methods will display the exact same numbers. Method **count1** uses *post-recursion*. In this case the recursive calls comes after the method action of printing values. Method **count2** uses *pre-recursion*. Recursive calls are made before any method action occurs. With pre-recursion there are many unfinished method calls. Printing is not possible, because every arrival at the method results in a departure on a new recursive call. All the print statements are stored for later execution. Since unfinished method business is stored on a stack, which behaves like a LIFO (Last In, First Out), the numbers will be displayed in reverse order. Method **count1** prints in forward order.

Example 10.4

```
public static void count1(int a, int b)
{
    if (a <= b)
    {
        System.out.print(a + "   ");
        count1(a+1,b);           // Post-Recursion
    }
}

public static void count2(int a, int b)
{
    if (a <= b)
    {
        count2(a+1,b);         // Pre-Recursion
        System.out.print(a + "   ");
    }
}
```

Post-recursion methods do not cause much trouble. The trouble making occurs with pre-recursion when it is easy to forget that unfinished recursive calls must eventually be accounted for. Consider the **booHiss** method in **Example 10.5**.

Example 10.5

```
public int booHiss(int n)
{
    if (n == 1)
        return 25;
    else
        return n + booHiss(n-1);
}
```

Suppose that method **booHiss** is called with the statement `System.out.println(booHiss(5));` It is possible that you consider the result at the time when the base case is reached. At that time **n** is 2 and added to 25 equals 27. Returning 27 is the wrong answer and ignores all the values that have patiently been waiting to be used. Each time when the method is called, the value of **n** is stored and must be added when the exit is finally reached as shown in the **Example 10.5 Table**.

Example 10.5 Table

What value is returned by the call booHiss(5) ?		
CALL #	n	Method booHiss returns
1	5	5 + booHiss(4)
2	4	4 + booHiss(3)
3	3	3 + booHiss(2)
4	2	2 + booHiss(1)
5	1	25
m3(5) == 25 + 2 + 3 + 4 + 5 == 39		

Perhaps the easiest mistake to make is the case where multiple recursive calls occur on the same line. The **hissBoo** method in **Example 10.6** is an example of a method with two recursive calls in one program statement. The table calculates the value as it was done in the previous example. It may seem correct, but the answer is wrong.

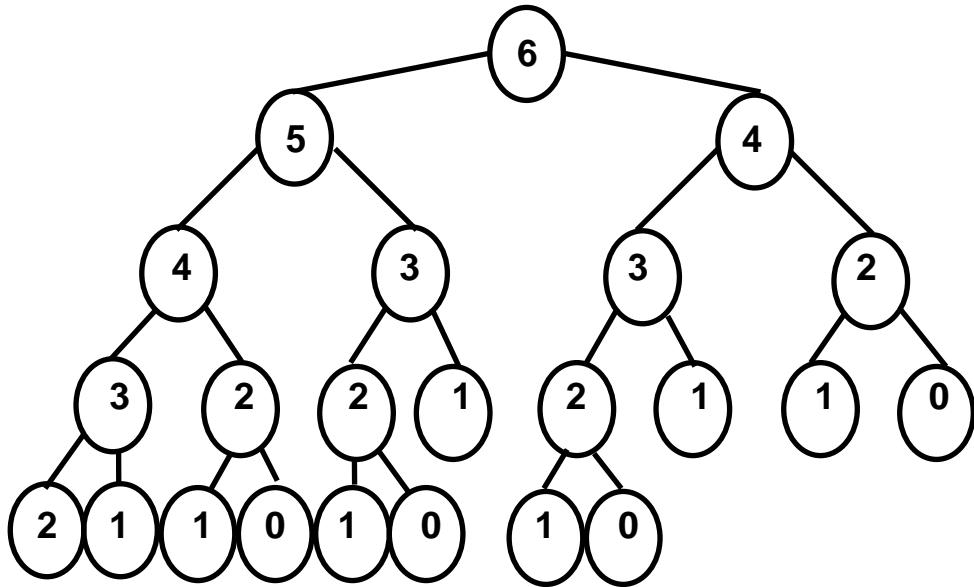
Example 10.6

```
public int hissBoo(int n)
{
    if (n == 1 || n == 0)
        return 0;
    else
        return n + hissBoo(n-1) + hissBoo(n-2);
}
```

What value is returned by the call **hissBoo(6)** ?

CALL #	n	Method hissBoo returns
1	6	6 + hissBoo(5) + hissBoo(4)
2	5	5 + hissBoo(4) + hissBoo(3)
3	4	4 + hissBoo(3) + hissBoo(2)
4	3	3 + hissBoo(2) + hissBoo(1)
5	2	2 + hissBoo(1) + hissBoo(0)
6	1	0
hissBoo(6) == 0 + 2 + 3 + 4 + 5 + 6 == 20		

Keep in mind that each recursive call executes a statement that makes two recursive calls. The binary tree below demonstrates how this happens. It also shows that the correct answer is the sum of the values stored by each recursive call.



This shows that the correct answer is **$6 + 5 + 4 + 4 + 3 + 3 + 2 + 3 + 2 + 2 + 2 = 38$** .

If you are wondering why the 1s are not added, look at the *base case* in the program. When n is equal to 1 (or when n is equal to 0), the value of 0 is returned. Non-zero values are returned only when n is 2 or more.

Chapter 10 Questions

Recursion

01. Consider the following method.

```
// precondition: a > 0 and b > 0
public static int method1001(int a, int b)
{
    if (b == 1)
        return a;
    else
        return a + method1001(a,b-1);
}
```

Which of the following expressions represents the result of calling `method1001` ?

- (A) $a + b$
- (B) $a * b$
- (C) a^b
- (D) b^a
- (E) A!

02. Consider the following method.

```
// precondition: a > 0 and b > 0
public static int method1002(int a, int b)
{
    if (b == 1)
        return a;
    else
        return a * method1002(a,b-1);
}
```

Which of the following expressions represents the result of calling `method1002` ?

- (A) $a + b$
- (B) $a * b$
- (C) a^b
- (D) b^a
- (E) A!

03. Consider the following method.

```
// precondition: a > 0
public static int method1003(int a)
{
    if (a == 0)
        return 1;
    else
        return a * method1003(a-1);
}
```

Which of the following expressions represents the result of calling `method1003` ?

- (A) $a + b$
- (B) $a * b$
- (C) $a!$
- (D) a^b
- (E) b^a

04. Consider the following method.

```
public static int method1004(int a)
{
    if (a <= 1)
    {
        return 1;
    }
    else
    {
        if (a % 2 == 0)
            return a - method1004(a-1);
        else
            return a + method1004(a-1);
    }
}
```

What value is returned as a result of the call `method1004(7)` ?

- (A) 0
- (B) 1
- (C) 4
- (D) 8
- (E) 10

05. Consider the following method.

```
public static int method1005(int x, int y)
{
    if (x <= y)
        return x;
    else
        return method1005(x-1,y+1);
}
```

What value is returned as a result of the call `method1005(7, 2)` ?

- (A) 0
- (B) 1
- (C) 4
- (D) 6
- (E) 8

06. Consider the following method.

```
public static int method1006(int n1, int n2)
{
    int rem = n1 % n2;
    if (rem == 0)
        return n2;
    else
    {
        n1 = n2;
        n2 = rem;
        return method1006(n1,n2);
    }
}
```

What value is returned as a result of the call `quest1006(150, 135)` ?

- (A) 1
- (B) 5
- (C) 15
- (D) 85
- (E) 285

07. Consider the following two methods.

```
public static int method1007a(int x, int y)
{
    return x / method1007b(x,y) * y;
}

public static int method1007b(int x, int y)
{
    if (x % y == 0)
        return y;
    else
        return method1007b(y,x%y);
}
```

What value is returned as a result of the call `method1007a(120,108)` ?

- (A) 1
- (B) 12
- (C) 228
- (D) 1080
- (E) 2160

08. Consider the following method.

```
public static int method1008(int q)
{
    if (q == 0)
        return 1;
    else
        return method1008(q-1) + method1008(q-1) ;
}
```

What value is returned as a result of the call `method1008(n)` ?

- (A) 2^n
- (B) n^2
- (C) $(n-1)^2$
- (D) $n * (n-1)$
- (E) $2^{(n-1)}$

09. Consider the following method.

```
public static void method1009(String s)
{
    int n = s.length();
    if (n > 0)
    {
        System.out.println(s);
        String temp = s.substring(0,n-1);
        method0909(temp);
    }
}
```

Which of the following is printed as a result of the call `method1009("TANGO")` ?

- (A) O
- (B) TANGO
- (C) TANGO
ANGO
NGO
GO
O
- (D) TANGO
TANG
TAN
TA
T
- (E) TANGO
TANG
TAN
TA

10. Consider the following method.

```
public static void method1010(String s)
{
    int n = s.length();
    if (n > 0)
    {
        String temp = s.substring(0,n-1);
        method1010(temp);
        System.out.println(s);
    }
}
```

Which of the following is printed as a result of the call `method1010("TANGO")` ?

- (A) O
- (B) TA
TAN
TANG
TANGO
ANGOT
- (C) TANGO
ANGO
NGO
GO
O
- (D) TA
TAN
TANG
TANGO
- (E) T
TA
TAN
TANG
TANGO

11. Consider the following four methods.

// Iterative Method <pre>public static int linear1(int list[], int key) { boolean found = false; int k = 0; while (k < list.length && !found) { if (list[k] == key) found = true; else k++; } if (found) return k; else return -1; }</pre>	// Recursive Method <pre>public static int linear2(int list[], int key) { int k = 0; if (k == list.length) return -1; else if (list[k] == key) return k; else { k++; return linear2(list,key); } }</pre>
// Recursive Method <pre>public static int linear3(int list[], int key, int k) { if (k == list.length) return -1; else if (list[k] == key) return k; else return linear3(list,key,k+1); }</pre>	// Recursive Method <pre>public static int linear4(int list[], int key, int k) { if (k == list.length) return -1; else if (list[k] == key) return k; else { k++; return linear4(list,key,k); } }</pre>

Method `linear1` is a correct, iterative *linear search* method. Which of the three recursive methods performs the same action as the `linear1` method?

- (A) `linear2` only
- (B) `linear3` only
- (C) `linear4` only
- (D) `linear2` and `linear3` only
- (E) `linear3` and `linear4` only

12. Consider the following four methods.

// Iterative Method <pre> public static int binary1(int list[], int key) { int lo = 0; int hi = list.length-1; int mid = 0; boolean found = false; while (lo <= hi && !found) { mid = (lo + hi) /2; if (list[mid] == key) found = true; else if (key > list[mid]) lo = mid + 1; else hi = mid - 1; } if (found) return mid; else return -1; } </pre>	// Recursive Method <pre> public static int binary2(int list[], int key, int lo, int hi) { int mid = 0; if (lo > hi) return -1; else { mid = (lo + hi) /2; if (list[mid] == key) return mid; else if (key > list[mid]) return binary2(list,key,mid+1,hi); else return binary2(list,key,lo,mid-1); } } </pre>
// Recursive Method <pre> public static int binary3(int list[], int key) { int lo = 0; int hi = list.length-1; int mid = 0; if (lo > hi) return -1; else { mid = (lo + hi) /2; if (list[mid] == key) return mid; else if (key > list[mid]) { lo = mid + 1; return binary3(list,key); } else { hi = mid - 1; return binary3(list,key); } } } </pre>	// Recursive Method <pre> public static int binary4(int list[], int key, int lo, int hi) { int mid = 0; if (lo > hi) return -1; else { mid = (lo + hi) /2; if (list[mid] == key) return mid; else if (key > list[mid]) { lo = mid + 1; return binary4(list,key,lo,hi); } else { hi = mid - 1; return binary4(list,key,lo,hi); } } } </pre>

Method `binary1` is a correct, iterative *binary search* method. Which of the three recursive methods performs the same action as the `binary1` method?

- (A) `binary2` only
- (B) `binary3` only
- (C) `binary4` only
- (D) `binary2` and `binary4` only
- (E) `binary3` and `binary4` only

Questions 13-14 refer to the following Ada class.

```
public class Ada
{
    public static void main(String args[])
    {
        String s1 = "HANNAH";
        String s2 = "Bob";
        String s3 = "racecar";
        String s4 = "qwerty";
        String s5 = "madam I'm Adam";
        String s6 = "Z";
        String s7 = "";

        System.out.print(boohiss(s1) + " ");
        System.out.print(boohiss(s2) + " ");
        System.out.print(boohiss(s3) + " ");
        System.out.print(boohiss(s4) + " ");
        System.out.print(boohiss(s5) + " ");
        System.out.print(boohiss(s6) + " ");
        System.out.println(boohiss(s7));
    }

    public static boolean boohiss(String s)
    {
        int n = s.length();
        if (n < 2)
            return true;
        else
        {
            char a = s.charAt(0);
            char b = s.charAt(n-1);
            if (a != b)
                return false;
            else
                return boohiss(s.substring(1,n-1));
        }
    }
}
```

13. What will be output when the program executes?

- (A) true false true false true false true
- (B) true false true false false true true
- (C) false true false true false true true
- (D) true true true true true true
- (E) false false false false false false false

14. When does method boohiss return true?
- (A) When s contains only capital letters.
 - (B) When s contains only lowercase letters.
 - (C) When s contains a mixture of capital and lowercase letters.
 - (D) When s is a palindrome, meaning the string is spelled exactly the same forwards and backwards.
 - (E) For all String values of s.

15. Consider the following code segment and method.

```
String s1 = "JESSICA";
String s2 = "brenda";
System.out.println(mystery(s1,s2));

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));
}
```

What will be output when the program segment executes?

- (A) JESSIONICda
- (B) JEbrSSenICdA
- (C) JEbrSSenICda
- (D) JEbrSSenICdaA
- (E) Stack Overflow due to infinite recursion.

Chapter 11 Review

Implementing Methods and Redefining Methods

There is more to method writing than starting with a method heading and filling in program statements between the opening brace and closing brace. There are three general approaches that can be used when you first start to write a method. First, is this a brand-new method that must be designed and written from scratch? Second, is this a redefinition of a method in a super class? Third, is this an implementation of an abstract method in an interface? This chapter is concerned with redefining methods and implementing abstract methods.

Inheritance allows access to the public methods of a super class. Inheritance also allows the redefinition of super class methods. You might wonder why there is a need to use inheritance if super class methods are redefined. The point of using inheritance is to increase program reliability by using existing classes that have been thoroughly tested and improved. Sooner or later the situation occurs where you wish to create a class that is similar to an existing class. It is possible to start from scratch, but the existing class contains a number of methods that are perfect for your needs. The existing class also contains several methods that must be rewritten. In such a situation your new class benefits by becoming a sub class of the existing class. You happily use the existing super class methods that are perfect for your needs and you redefine other methods. This approach gives greater reliability and less work.

This may seem a repeat of the inheritance chapter, but there is a point. There are two important Java methods that are used very frequently and they are constantly redefined in different classes. They are methods **toString** and **equals**.

The highest level class, which is the super class for all other classes, is the **Object** class. The **Object** class contains the original definition of methods **toString** and **equals**. This means that these two methods are always available in any class you create. It also means that you will always get the original definition of these methods, unless you redefine one or both of the methods. Before redefinition becomes an issue, you must realize the details of the original definition in the **Object** class.

APCS Examination Alert

The **Object** class is the super class of all other classes.

Methods **toString** and **equals** are defined in the **Object** class.

Methods **toString** and **equals** can be used by any class you create.

Methods **toString** and **equals** are tested on the APCS Exam.

Method **toString** works quietly in the background. Methods **print** and **println** display a string, which is returned by method **toString**. Even though it is not visible in the program code, each time **print** or **println** is used in any program statement, **toString** is also used to assist in the output display of the parameter information used by **print** and **println**. The original **toString** returns the string representation of the *immediate* or *shallow* value of a variable. That means you will get practical values for primitive data types, but memory references for objects.

The Original **toString** Method

print and **println** request display instructions from the **toString** method.

The **toString** method, as defined by the **Object** class, returns the actual string representation values of all the primitive types like **int** and **double**.

toString returns the class name followed by the memory reference of any variable object.

Example 11.1 shows a small **Student** class. Method **println** is used to display the value of the **int** variable **number** and the **student** object. The output shows the **100** value for **number**, but the output of **student** is the name of the class followed by a memory reference.

Method **toString** follows the instructions precisely specified in the **Object** class.

Example 11.1

```
public class Example111
{
    public static void main (String args[])
    {
        int number = 100;
        Student student = new Student ("Tom", 21);
        System.out.println(number + " " + student);
    }
}

class Student
{
    private String name;
    private int age;

    public Student(String n, int a)
    {
        name = n;
        age = a;
    }
}
```

Program Output

```
100  Student@19821f
```

Make sure that you do not get confused. The program in **Example 11.2** shows an **ArrayList** object **names** and this object displays without difficulty. The four names stored in the object are shown without any memory reference. This program actually reminds you of the whole *redefinition* point. The **toString** method is redefined for the **ArrayList** class or at least some super class of the **ArrayList** class.

Example 11.2

```
public class Example112
{
    public static void main (String args[])
    {
        ArrayList<String> names = new ArrayList<String>();
        names.add("Sue");
        names.add("Tom");
        names.add("Joe");
        names.add("Ann");
        System.out.println(names);
    }
}
```

Program Output

```
[Sue, Tom, Joe, Ann]
```

Method **toString** Notes

print and **println** request display instructions from the **toString** method.

The **toString** method, as defined by the **Object** class, returns the actual string representation values of all the primitive types like **int** and **double**.

toString returns the class name followed by the memory reference of any variable object.

Many classes, such as the **ArrayList** class, use a redefined **toString** method.

If **toString** is not redefined in any class you declare, the output of your class object will be the class name followed by a memory reference.

Class **Student** is rewritten in **Example 11.3**. This time method **toString** is redefined. It is a short method without many details. The only requirement is that a string is returned. In this case a format is used to mimic the style of output that is used by the **ArrayList** class.

Example 11.3

```
class Student
{
    private String name;
    private int age;

    public Student(String n, int a)
    {
        name = n;
        age = a;
    }

    public String toString()
    {
        return "[" + name + ", " + age + "]";
    }
}
```

The new, and improved, **Student** class can now be used and tested. **Example11.3** continued shows that the output does in fact have the same output format as an **ArrayList** object, including square brackets and commas.

Example 11.3 Continued

```
public class Example113
{
    public static void main (String args[])
    {
        Student student = new Student("Tom",21);
        System.out.println(student);
    }
}
```

Program Output

```
[Tom, 21]
```

Method **equals** is not very practical with its original definition. Like **toString**, **equals** is redefined for many Java standard classes, like the **String** class. When you create your own classes make sure that you consider the equality issue. The program in **Example 11.4** compares students and checks equality according to GPA.

Example 11.4

```
public class Example114
{
    public static void main (String args[])
    {
        Student student1 = new Student("Tom",21,3.575);
        Student student2 = new Student("Sue",23,2.925);
        Student student3 = new Student("Ann",19,3.575);
        if (student1.equals(student2))
            System.out.println("student1 equals student2");
        else
            System.out.println("student1 does not equal student2");
        if (student1.equals(student3))
            System.out.println("student1 equals student3");
        else
            System.out.println("student1 does not equal student3");
    }
}

class Student
{
    private String name;
    private int age;
    private double gpa;

    public Student(String n, int a, double g)
    {
        name = n;
        age = a;
        gpa = g;
    }

    public String getName() { return name; }
    public int getAge() { return age; }
    public double getGPA() { return gpa; }

    public boolean equals(Object obj)
    {
        Student temp = (Student) obj;
        return temp.getGPA() == gpa;
    }
}
```

Program Output

```
student1 does not equal student2
student1 equals student3
```

Method equals Notes

The **equals** method, as defined by the **Object** class, compares the immediate, *shallow*, values of two objects.

Many classes, such as the **String** class and the **ArrayList** class, use a redefined **equals** method.

If method **equals** is not redefined in any class you create, equality will be tested according the shallow or memory references of your objects.

Implementing methods appears the same as redefining methods at first glance. It is not the same. You redefine a method of a super class. You implement an abstract method of an interface or abstract class. Implementing of a method is the first time the method is defined.

An interface looks like a class with methods that only have a method heading. The methods are abstract, because there is a method name and there is a method signature, but there is no method body with program statements. **Example 11.5** shows **Widget**, which is an example of an interface. Note how every abstract method declares a heading ending with a semi-colon. The **Bumba** class, in **Example 11.6** implements the **Widget** interface.

Example 11.5

```
public interface Widget
{
    public int getWidgets();
    public void setWidgets(int w);
}
```

Example 11.6

```
class Bumba implements Widget
{
    private int numWidgets;

    public Bumba(int w)           { numWidgets = w; }

    public int getWidgets()        { return numWidgets; }

    public void setWidgets(int w)  { numWidgets = w; }
}
```

The classes in **Example 11.7** and **Example 11.8** are very similar to the previous two examples. In this case an abstract super class is used with a sub class. It is necessary to state that the class is abstract. It is also necessary to declare every method as abstract. You may declare methods in an interface abstract, but it is not required. All interface methods must be abstract. The primary difference between an abstract class and an interface is that not every member of an abstract class must be abstract. Note that the **Widget** class has an attribute and a constructor. Methods **getWidgets** and **setWidgets** are abstract and they are implemented in the **Bumba** class.

Example 11.7

```
public abstract class Widget
{
    protected int numWidgets;

    public Widget(int w)
    {
        numWidgets = w;
    }

    public abstract int getWidgets();

    public abstract void setWidgets(int w);
}
```

Example 11.8

```
class Bumba extends Widget
{
    private int numWidgets;

    public Bumba(int w)
    {
        super(w);
    }

    public int getWidgets()
    {
        return numWidgets;
    }

    public void setWidgets(int w)
    {
        numWidgets = w;
    }
}
```

There is a small, but important standard Java interface, called **Comparable**. The interface has a single method, called **compareTo**. Method **equals** only compares for equality. Method **compareTo** also compares for inequality. The program in **Example 11.9** declares a **Gumba** class which implements the **compareTo** method.

Example 11.9

```
public class Example119
{
    public static void main(String args[])
    {
        Gumba gum1 = new Gumba(200);
        Gumba gum2 = new Gumba(300);
        Gumba gum3 = new Gumba(200);
        Gumba gum4 = new Gumba(100);
        System.out.print(gum1.compareTo(gum2) + " ");
        System.out.print(gum1.compareTo(gum3) + " ");
        System.out.print(gum1.compareTo(gum4) + " ");

    }
}

class Gumba implements Comparable
{
    private int numGumbas;

    public Gumba(int g)
    {
        numGumbas = g;
    }

    public int getGumbas()
    {
        return numGumbas;
    }

    public void setGumbas(int g)
    {
        numGumbas = g;
    }

    public int compareTo(Object obj)
    {
        Gumba temp = (Gumba) obj;
        if (this.numGumbas < temp.numGumbas)
            return -1;
        else if (this.numGumbas == temp.numGumbas)
            return 0;
        else
            return 1;
    }
}
```

Program Output

```
-1 0 1
```

Method **compareTo** Notes

Method **compareTo** is the only method of the **Comparable** interface.

Multiple objects of a class can be compared for equality and inequality by implementing the **compareTo** method.

int compareTo (Object other)

returns -1, if **this** is less than **other** (or any value < 0)

returns 0, if **this** is equal to **other**

returns 1, if **this** is greater than **other** (or any value > 0)

APCS Examination Alert

The **Object** class and its methods **toString** and **equals**, as defined in the **Object** class, are tested on the APCS Exam.

Redefinition of **toString** and **equals** is tested on the APCS Exam.

The **Comparable** interface, its method **compareTo** and implementing method **compareTo** is tested on the APCS Exam.

Redefinition and Implementation Pitfalls

It is not necessary to have a special pitfall section for this chapter. There are not many subtle problems. You need to know the exact method headings and make sure that they are properly redefined or implemented. One potential problem happens with a parameter, such as . . .

int compareTo (Object other)

When the **other** variable is used it must be casted to the proper class. Without the casting it will not be possible to make a proper comparison. The same issue applies to the **equals** method.

Chapter 11 Questions

Implementing Methods and Redefining Methods

01. Consider the following interface.

```
interface MyList
{
    public void addList(Object item);
    public void removeList(int index);
    public void showList();
}
```

Which of the following is true about the MyNames class implementing the MyList interface?

- (A) The MyNames class must implement any MyList method that is necessary for the MyNames class.
- (B) The MyNames class must implement each one of the MyList methods and no other methods.
- (C) The MyNames class must implement each one of the MyList methods and the MyNames constructor.
- (D) The MyNames class may only implement each one of the MyList methods,
and declare necessary attributes.
- (E) The MyNames class must implement each one of the MyList methods,
and may optionally declare other methods and attributes.

02. Consider the following `MyList` interface and incomplete `MyNames` class.

```
interface MyList
{
    public void addList(Object item);
    public void removeList(int index);
    public void showList();
}

class MyNames implements MyList
{
    private ArrayList<String> names;

    public MyNames()
    {
        names = new ArrayList<String>();
    }

    public void addList(Object item)
    // Precondition: item is a String object
    // Postcondition: item is appended to the names array
    {
        /* missing code */
    }

    public void removeList(int index)
    {
        /* missing code */
    }

    public void showList()
    {
        /* missing code */
    }
}
```

Which of the following three `addList` implementations of `/* missing code */` will make the method `addList` satisfy the post condition?

Implementation 1

```
String name = (String) item;
names.add(name);
```

Implementation 2

```
names.add((String)item);
```

Implementation 3

```
names.add(item);
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementations 1 and 2 only
- (E) Implementation 1, 2 and 3

03. Consider the following `MyList` interface and incomplete `MyNames` class.

```
interface MyList
{
    public void addList(Object item);
    public void removeList(int index);
    public void showList();
}

class MyNames implements MyList
{
    private ArrayList<String> names;

    public MyNames()
    {
        names = new ArrayList<String>();
    }

    public void addList(Object item)
    {
        /* missing code */
    }

    public void removeList(int k)
    // precondition: 0 <= index < names.length
    // post condition: names element with index k is removed
    {
        /* missing code */
    }

    public void showList()
    {
        /* missing code */
    }
}
```

Which of the following three `removeList` implementations of `/* missing code */` will make the method `removeList` satisfy the post condition?

Implementation 1

```
for (String name: names)
    names.remove(k);
```

Implementation 2

```
(String) names.remove(item);
```

Implementation 3

```
names.remove(k);
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementations 1 and 2 only
- (E) Implementation 1, 2 and 3

04. Consider the following code segment.

```
ArrayList<String> names = new ArrayList<String>();
names.add("Astrid");
names.add("Leon");
names.add("Ingrid");
names.add("Remy");
System.out.println(names);
```

What is printed as a result of executing the code segment?

- (A) ArrayList@<some hexadecimal number> like ArrayList@18d107f
- (B) [Astrid, Leon, Ingrid, Remy]
- (C) Astrid
- (D) Astrid Leon Ingrid Remy
- (E) Remy Ingrid Leon Astrid

05. Consider the following Quirk class declaration.

```
class Quirk
{
    public Quirk(int nq)      { numQuirks = nq; }

    public String toString() { return numQuirks; }

    public int getQuirks()   { return numQuirks; }

    private int numQuirks;
}
```

Assume that a Quirk object is constructed with the following code segment.

```
Quirk q = new Quirk(50);
```

Which of the following three statements will display the number of quirks stored by the Quirk object?

- I. System.out.println(q)
- II. System.out.println(q.toString());
- III. System.out.println(q.getQuirks());

- (A) I only
- (B) II only
- (C) III only
- (D) I and II only
- (E) II and III only

06. Consider the following Quirk class declaration.

```
class Quirk
{
    public Quirk(int nq)      { numQuirks = nq; }

    public String toString() { return String.valueOf(numQuirks); }

    public int getQuirks()   { return numQuirks; }

    private int numQuirks;
}
```

Assume that a Quirk object is constructed with the following code segment.

```
Quirk q = new Quirk(50);
```

Which of the following three statements will display the number of quirks stored by the Quirk object?

- I. System.out.println(q);
- II. System.out.println(q.toString());
- III. System.out.println(q.getQuirks());

- (A) I only
- (B) II only
- (C) III only
- (D) I and II only
- (E) I, II and III

07. Consider the following `IntList` class with the incomplete `toString` method.

```
class IntList
{
    public IntList(int n)
    {
        Random rnd = new Random();
        intArray = new int[n];
        for (int k = 0; k < n; k++)
            intArray[k] = rnd.nextInt(100);
    }

    public String toString()
    // post condition: elements of intArray are returned in the format:
    //                  [e0, e1, e2, . . . , en-2, en-1]
    {
        /* missing code */
    }

    private int intArray[];
}
```

Which of the following three implementations of `/* missing code */` will make method `toString` work as intended?

Implementation 1

```
String temp = "[ ";
for (int k = 0; k < intArray.length-1; k++)
    temp += intArray[k] + ", ";
temp += intArray[intArray.length-1] + "]";
return temp;
```

Implementation 2

```
System.out.print("[ ");
for (int k = 0; k < intArray.length-1; k++)
    System.out.print(intArray[k] + ", ");
System.out.println("]");
return "";
```

Implementation 3

```
return "[" + intArray + "]";
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementations 1 and 2 only
- (E) Implementations 1 and 3 only

08. Consider the following Car class with the missing implementation of the compareTo method.

```
class Car implements Comparable
{
    public Car(int hp)
    {
        horsePower = hp;
    }

    public int getHP()
    {
        return horsePower;
    }

    /* missing compareTo method */

    private int horsePower;
}
```

The Car class needs to implement the compareTo method in such a manner that a comparison is returned about the horse power of two Car objects. For instance, assume that object car1 has 150 hp and object car2 has 200 hp. The call car1.compareTo(car2) needs to return -50 and the call car2.compareTo(car1) needs to return 50.

Which of the following three implementations will make method compareTo work as intended?

Implementation 1

```
public int compareTo(Object source)
{
    Car temp = (Car) source;
    return horsePower - temp.horsePower;
}
```

Implementation 2

```
public int compareTo(Object source)
{
    return horsePower - source.horsePower;
}
```

Implementation 3

```
public void compareTo(Object source)
{
    System.out.println(horsePower - source.horsePower);
}
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementations 1 and 2 only
- (E) Implementations 2 and 3 only

09. Consider the following Banana class and the `compareTo` method.

```
class Banana
{
    public Banana(int b)
    {
        bananas = b;
    }

    public int getBananas()
    {
        return bananas;
    }

    public int compareTo(Object source)
    {
        Banana temp = (Banana) source;
        return bananas - temp.bananas;
    }

    private int bananas;
}
```

The Banana class needs to implement the Comparable interface and the `compareTo` method in such a manner that a comparison is returned about the number of bananas of two Banana objects. For instance, assume that object `banana1` has 20 bananas and object `banana2` has 30 bananas. The call `banana1.compareTo(banana2)` needs to return -10 and the call `banana2.compareTo(banana1)` needs to return 10.

Now consider the code segment below, which is designed to check if the Banana class work correctly.

```
Banana banana1 = new Banana(20);
Banana banana2 = new Banana(30);
System.out.println(banana1.compareTo(banana2));
System.out.println(banana2.compareTo(banana1));
```

Executing the code segment in an appropriate program will demonstrate that the Banana class satisfies the requirements stated above. Which of the following is a true statement about the implementation of the Comparable interface and the `compareTo` method in the Banana class?

- (A) The Banana class implements the Comparable interface and method `compareTo` correctly.
- (B) All implementations of the Comparable interface are correct, because the implementer can decide on the functionality of the `compareTo` method and any other methods.
- (C) The Banana class does implement a `compareTo` method according to specifications, but it does not implement the Comparable interface correctly.
- (D) The Banana class implements the Comparable interface correctly and the `compareTo` method incorrectly.
- (E) The Banana class fails to extend the Comparable interface, which prevents access to the `compareTo` features that are defined in Comparable.

10. Consider the following class and code segment.

```
class Student
{
    public Student(int a, double g)
    {
        age = a;
        gpa = g;
    }

    public int getAge()
    {
        return age;
    }

    public double getGPA()
    {
        return gpa;
    }

    public boolean equals(Object obj)
    {
        return ((Student)obj).getAge() == age;
    }

    private int age;
    private double gpa;
}

Student tom = new Student(25,3.125);
Student sue = new Student(23,3.125);
Student ann = new Student(25,2.515);
System.out.println(tom.equals(sue));
System.out.println(tom.equals(ann));
```

What is printed as a result of executing the code segment?

- (A) true
false
- (B) true
true
- (C) false
false
- (D) false
true
- (E) 25 23
25 25

11. Consider the following class and code segment.

```
class Student
{
    public Student(int a, double g)
    {
        age = a;
        gpa = g;
    }

    public int getAge()
    {
        return age;
    }

    public double getGPA()
    {
        return gpa;
    }

    public boolean equals(Object obj)
    {
        /* missing code */
    }

    private int age;
    private double gpa;
}

Student tom = new Student(25,3.125);
Student sue = new Student(23,3.125);
Student ann = new Student(25,2.515);
System.out.println(tom.equals(sue));
System.out.println(tom.equals(ann));
```

Which of the following implementation can be used to replace **/* missing code */** so that objects of the `Student` class test equality according to GPA?

Implementation 1

```
return ((Student)obj).getGPA() == gpa;
```

Implementation 2

```
Student temp = (Student) obj;
return temp.getGPA() == gpa;
```

Implementation 3

```
return (Student)obj.getGPA() == gpa;
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementation 1 and 2
- (E) Implementation 2 and 3

12. Consider the following class and code segment.

```
class Student
{
    public Student(int a, double g)
    {
        age = a;
        gpa = g;
    }

    public int getAge()
    {
        return age;
    }

    public double getGPA()
    {
        return gpa;
    }

    public boolean equals(Object obj)
    {
        /* missing code */
    }

    private int age;
    private double gpa;
}

Student tom = new Student(25,3.125);
Student sue = new Student(23,3.125);
Student ann = new Student(25,2.515);
System.out.println(tom.equals(sue));
System.out.println(tom.equals(ann));
```

Which of the following implementation can be used to replace **/* missing code */** so that all objects of the Student class test equal?

Implementation 1

```
return ((Student)obj).getAge() == age && ((Student)obj).getGPA() == gpa;
```

Implementation 2

```
Student temp = (Student) obj;
return temp.getAge() == age && temp.getGPA() == gpa;
```

Implementation 3

```
return true;
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementation 1 and 2
- (E) Implementation 2 and 3

13. Consider the following `Borb` class declaration.

```
class Borb
{
    private int numBorbs;

    public Borb(int n)
    {
        numBorbs = n;
    }

    public String toString()
    {
        return "Borb count: " + numBorbs;
    }
}
```

Assume that a `Borb` object is constructed with the following code segment.

```
Borb b = new Borb(100);
```

Which of the following three statements will display the number of `borbs` stored by the `Borb` object?

- I. `System.out.println(b);`
 - II. `System.out.println(b.toString());`
 - III. `b.toString();`
-
- (A) I only
 - (B) II only
 - (C) III only
 - (D) I and II only
 - (E) II and III only

14. Consider the following class and code segment.

```
Student student = new Student("Bert Pauw", 58, 3.75);
System.out.println(student);

class Student
{
    private String name;
    private int age;
    private double gpa;

    public Student(String n, int a, double g)
    {
        name = n;
        age = a;
        gpa = g;
    }

    public String toString()
    {
        return name;
    }
}
```

What is printed as a result of executing the code segment?

- (A) Bert Pauw
- (B) Bert Pauw, 58, 3.75
- (C) Bert Pauw
58
3.75
- (D) [Bert Pauw, 58, 3.75]
- (E) Student@ <some memory location>

15. Consider the following class and code segment.

```
Student student = new Student("Bert Pauw", 58, 3.75);
System.out.println(student);

class Student
{
    private String name;
    private int age;
    private double gpa;

    public Student(String n, int a, double g)
    {
        name = n;
        age = a;
        gpa = g;
    }

    public String toString()
    {
        String temp = name;
        temp += "\n";
        temp += String.valueOf(age);
        temp += "\n";
        temp += String.valueOf(gpa);
        temp += "\n";
        return temp;
    }
}
```

What is printed as a result of executing the code segment?

- (A) Bert Pauw
- (B) Bert Pauw, 58, 3.75
- (C) Bert Pauw
58
3.75
- (D) [Bert Pauw, 58, 3.75]
- (E) Student@ <some memory location>

Chapter 12 Review

Java Standard Library Methods

The grand total of Java libraries for classes and methods is huge. The College Board has selected a subset of these Java standard library methods to be tested for the examination. The greater majority of these methods have already been reviewed in other chapters. The intention of this chapter is to consolidate all the standard library methods in one location.

Since the methods in this section are explained in other chapters, they will only be summarized here. The information you see is similar to the review sheets that you will receive at the actual APCS Exam. These summaries are important, because most students have learned many concepts and classes that will not be tested. The whole intention of this study guide is to focus only on computer science topics and Java features that will be tested.

class java.lang.Object

boolean equals (Object other)	// returns true if this equals other and false otherwise
String toString ()	// returns a String representation of the object

class java.lang.Integer

Integer (int value)	// constructs a newly allocated Integer object that represents the int value
int intValue ()	// returns the value of this Integer as an int
Integer.MIN_VALUE	// minimum value represented by an int
Integer.MAX_VALUE	// maximum value represented by an int

class java.lang.Double

Double (double value)	// constructs a newly allocated Double object that represents the double value
double doubleValue ()	// returns the value of this Double as a double

class java.lang.String

int length ()	// returns the length - or number of characters - of this string
String substring (int from, int to)	// returns the substring beginning at <code>from</code> and ending at <code>to-1</code>
String substring (int from)	// returns <code>substring(from, length())</code>
int indexOf (String str)	// returns the index of the first occurrence of <code>str</code> ; returns <code>-1</code> if not found
int compareTo(String other)	// returns a value < 0 if <code>this</code> is less than <code>other</code> // returns a value = 0 if <code>this</code> is equal to <code>other</code> // returns a value > 0 if <code>this</code> is greater than <code>other</code>

class java.lang.Math

static int abs (int x)	// returns the absolute value of an <code>int</code> value
static double abs (double x)	// returns the absolute value of a <code>double</code> value
static double pow (double base, double exp)	// returns the value of <code>base</code> raised to the power of <code>exp</code>
static double sqrt (double x)	// returns the positive square root of a <code>double</code> value
static double random ()	// returns a <code>double</code> in the range [0.0 . . . 1.0]

interface java.util.List<E>

int size ()	// returns the number of elements in the list
boolean add (E obj)	// appends <code>obj</code> to end of list and returns <code>true</code>
void add (int index, E obj)	// inserts <code>obj</code> at position <code>index</code> (<code>0 <= index <= size</code>) // moving elements at position <code>index</code> and higher to the right and adjusts size
E get (int index)	// returns the element at position <code>index</code>
E set (int index, E obj)	// replaces the element at position <code>index</code> with <code>obj</code> // returns the element formerly at position <code>index</code>
E remove (int index)	// removes element from position <code>index</code> , moving elements at position <code>index+1</code> and higher to the left and adjusts size // returns the element formerly at position <code>index</code>

class java.util.ArrayList<E> implements java.util.List<E>

int size ()	// returns the number of elements in the list
boolean add (E obj)	// appends obj to end of list and returns true
void add (int index, E obj)	// inserts obj at position index (0 <= index <= size) // moving elements at position index and higher to the right and adjusts size
E get (int index)	// returns the element at position index
E set (int index, E obj)	// replaces the element at position index with obj // returns the element formerly at position index
E remove (int index)	// removes element from position index, moving elements at position index+1 // and higher to the left and adjusts size // returns the element formerly at position index

It may seem odd to have a **List** interface with a set of methods followed by an **ArrayList** interface with an identical set of methods. The intention here is to show the relationship between the interface and the implementing class. The interface dictates the method signature and the method behavior. The class is free to implement the requirements specified by the interface.

Sometimes, if you do not see the whole picture, certain requirements will seem odd. For instance, many people who first encounter the **add** method are surprised that it is a return method, which returns **true**. Looking at the manner in which an **ArrayList** object operates gives no justification for the use of a return method here.

However, there is a bigger picture here. Method **add** does not start in the **List** interface. It starts higher up in the **Collection** interface. Method **add** is expected to return true if the object is changed as a result of calling the **add** method. In the case of an **ArrayList** object that will certainly be the case. Now there are also **Set** classes, which implement the same **add** method that started in the **Collection** interface. Sets do not have duplicate elements. A new element is only added to a **Set** object, if the element does not already exist. If an attempt is made to add an element that exists, the **add** method will not add the element and returns **false** to indicate that the object was not changed.

In other words, the **Collection** interface provided a very general blue print and local implementations are used to satisfy the needs for the current class. This means that the **add** method is implemented differently for **Set** class compared to the implementation for the **List** classes.

Chapter 12 Questions

Java Standard Library Methods

01. Consider the following program segment.

```
for (int k = 1; k <= 10; k++)
{
    double rndDouble = Math.random();
    int rndInt = (int) (rndDouble * 50 + 10);
    System.out.print(rndInt + " ");
}
```

After the segment executes, which one of the following is a possible output?

- (A) 55 48 12 12 28 30 26 39 54 30
- (B) 34 14 26 60 47 11 29 51 35 19
- (C) 56 30 23 32 5 43 8 49 41 27
- (D) 10 20 30 40 50 60 50 40 30 20 10
- (E) 0.35 0.15 0.59 0.52 0.13 0.48 0.41 0.20 0.36 0.19

02. Consider the following program segment.

```
for (int k = 1; k <= 10; k++)
{
    double rndDouble = Math.random();
    int rndInt = (int) /* missing code */
    System.out.print(rndInt + " ");
}
```

Which of the following implementations of **/* missing code */** will generate a sequence of integers in the [400..1600] range?

- (A) (rndDouble * 1600);
- (B) (rndDouble * 1600 + 400);
- (C) (rndDouble * 1201 + 400);
- (D) (rndDouble * 401 + 1600);
- (E) (rndDouble * 1200 + 400);

03. Consider the following two code segments.

Segment 1

```
for (int k = 1; k <= 10; k++)
{
    double rnd1 = Math.random() * 100;
    System.out.println(rnd1 + " ");
}
```

Segment 2

```
for (int k = 1; k <= 10; k++)
{
    double rnd2 = Math.random() * 100;
    System.out.println((int) rnd2 + " ");
}
```

Which of the following statements is true about the execution comparison of **Segment 1** and **Segment 2**?

- I. Segment 1 and Segment 2 will display different numbers.
 - II. Segment 1 and Segment 2 will display numbers in the same range.
 - III. Segment 1 and Segment 2 will display numbers in a different range.
-
- (A) I only
 - (B) II only
 - (C) III only
 - (D) I and II
 - (E) I and III

04. Consider the following program segment.

```
int temp = 100;
for (int k = 1; k <= 10; k++)
{
    temp *= -1;
    if (k % 2 == 1)
        System.out.print(Math.sqrt(-temp) + " ");
}
```

What is printed as a result of executing the code segment?

- (A) NaN NaN NaN NaN NaN
- (B) 10.0 10.0 10.0 10.0 10.0
- (C) NaN 10.0 NaN 10.0 NaN 10.0 NaN 10.0
- (D) 10.0 NaN 10.0 NaN 10.0 NaN 10.0 NaN 10.0
- (E) ArithmeticException

05. Consider the following program segment.

```
int temp = 100;
for (int k = 1; k <= 10; k++)
{
    temp *= -1;
    if (k % 2 == 1)
        System.out.print(Math.sqrt(Math.abs(temp/0)) + " ");
}
```

What is printed as a result of executing the code segment?

- (A) NaN NaN NaN NaN NaN
- (B) 10.0 10.0 10.0 10.0 10.0
- (C) NaN 10.0 NaN 10.0 NaN 10.0 NaN 10.0 10.0
- (D) 10.0 NaN 10.0 NaN 10.0 NaN 10.0 NaN 10.0 NaN
- (E) ArithmeticException

06. Consider the following program segment.

```
double a = 2.5;
double b = -2.0;
System.out.print(Math.pow(a,b));
```

What is printed as a result of executing the code segment?

- (A) -6.25
- (B) 6.25
- (C) NaN
- (D) 0.16
- (E) ArithmeticException

07. Consider the following program segment.

```
ArrayList<Integer> list = new ArrayList<Integer>();
for (int k = 10; k < 50; k+=10)
    list.add(new Integer(k));
int sum = 0;
for (int index = 0; index < list.size(); index++)
{
    Integer temp = list.get(index);
    sum += temp.intValue();
}
System.out.println(sum);
```

What is printed as a result of executing the code segment?

- (A) 150
- (B) 120
- (C) 100
- (D) 50
- (E) 10

08. Consider the following program segment.

```
String s1 = new String("GridWorld");
String s2 = s1.substring(4,5);
System.out.println(s2);
```

What will be output when the program segment executes?

- (A) World
- (B) dWorl
- (C) Worl
- (D) dW
- (E) W

09. Consider the following program segment.

```
String s1 = new String("GridWorld");
int start = s1.indexOf("W");
String s2 = s1.substring(start);
System.out.println(s2);
```

What will be output when the program segment executes?

- (A) World
- (B) dWorl
- (C) Worl
- (D) dW
- (E) W

10. Consider the following program segment.

```
ArrayList<String> names = new ArrayList<String>();
names.add("Tom");
names.add("Sue");
names.add("Joe");
names.add("Ann");
names.add(2, "Bob");
System.out.println(names);
```

What will be output when the program segment executes?

- (A) [Tom, Sue, Bob, Ann]
- (B) [Tom, Sue, Bob, Joe, Ann]
- (C) [Tom, Bob, Sue, Joe, Ann]
- (D) [Tom, Bob, Joe, Ann]
- (E) [Tom, Sue, Joe, Ann, Bob]

11. Consider the following program segment.

```
ArrayList<String> names = new ArrayList<String>();
names.add("Tom");
names.add("Sue");
names.add("Joe");
names.add("Ann");
System.out.println(names.set(2, "Bob"));
System.out.println(names.get(2));
```

What will be output when the program segment executes?

- (A) Bob
Bob
- (B) Joe
Sue
- (C) Bob
Joe
- (D) Joe
Bob
- (E) Sue
Joe

12. Consider the following program segment.

```
ArrayList<String> names = new ArrayList<String>();
names.add("Tom");
names.add("Sue");
names.add("Joe");
names.add("Ann");
System.out.println(names.remove(1));
System.out.println(names.remove(3));
```

What will be output when the program segment executes?

- (A) Sue
IndexOutOfBoundsException
- (B) Sue
Ann
- (C) Tom
Joe
- (D) IndexOutOfBoundsException
- (E) Compile error

13. Consider the following program segment.

```
String s1 = "car";
String s2 = "racecar";
String s3 = "qwerty";
String s4 = "carport";

System.out.println(contains(s1,s2));
System.out.println(contains(s1,s3));
System.out.println(contains(s1,s4));

// precondition:  a and b are 2 nonempty Strings.
// postcondition: The method returns true if a is a substring of b,
//                  false otherwise.
public static boolean contains(String a, String b)
{
    /* missing code */
}
```

Which of the following statements implements the `contains` method as specified?

- (A) `return a.indexOf(a) == -1;`
- (B) `return a.indexOf(a) != -1;`
- (C) `return b.indexOf(a) == -1;`
- (D) `return b.indexOf(a) != -1;`
- (E) `return a.equals(b);`

14. Which of the following is a valid definition for the `qwerty` object?

- I. `List<String> qwerty = new List<String>();`
- II. `List<String> qwerty = new ArrayList<String>();`
- III. `ArrayList<String> qwerty = new List<String>();`
- IV. `ArrayList<String> qwerty = new ArrayList<String>();`

- (A) I and II only
- (B) I and III only
- (C) I and IV only
- (D) II and III only
- (E) II and IV only

15. Consider the following incomplete method.

```
// postcondition: mean returns the "mean" average of the
//                  double values in list
public static double mean(ArrayList<Double> list)
{
    /* missing code */
}
```

Which of the following implementations of **/* missing code */** will make method `mean` execute as intended?

Implementation I

```
double total = 0.0;
for (int j = 0; j < list.size(); j++)
    total += list.get(j);
return total / list.size();
```

Implementation II

```
double total = 0.0;
for (int j = 0; j < list.size(); j++)
    total += list.get(j).doubleValue();
return total / list.size();
```

Implementation III

```
double total = 0.0;
for (Double n: list)
    total += n.doubleValue();
return total / list.size();
```

- (A) Implementation I only
- (B) Implementation II only
- (C) Implementation III only
- (D) Implementations II and III only
- (E) Implementations I, II and III

Chapter 13 Review

Algorithms and Informal Algorithmic Analysis

The point of algorithmic analysis is not to reinvent the wheel, but to know which wheel to use. Computer science has many established algorithms. The biggest job of computer programs is data processing and data needs to be continuously searched and sorted. If there exist many algorithms, and these algorithms are conveniently available in some library, there is still the question about which algorithm to pick. This is where algorithmic analysis assists and helps you to decide the best algorithm for the job.

Algorithmic analysis for the APCS Examination is informal only. It is informal compared to the very mathematical analysis that is done at more advanced computer science courses. In this introductory level the analysis may be informal, limited, but it can still be very practical.

Analysis starts by understanding execution frequency of various control structures. The next five program examples, **Example 13.1 . . . Example 13.5** show various loop situations. In each program a **count** variable is initialized to **0** and the program displays the number of times that the loop or loops repeat, as indicated by the **count** value. With each program example there is also a *generalized* statement of how many times **count** increased based on variable **n**.

Example 13.1

```
public class Example131
{
    public static void main(String args[])
    {
        int count = 0;
        for (int k = 1; k <= 10; k++)
            count++;
        System.out.println("Count: " + count);
    }
}
```

Program Output

Count: 10

Example 13.2

```
public class Example132
{
    public static void main(String args[])
    {
        int n = 100;
        int count = 0;
        for (int k = 1; k <= n; k++)
            count++;
        System.out.println("Count: " + count);
    }
}
```

Program Output

Count: 100

General count: **n**

Example 13.3

```
public class Example133
{
    public static void main(String args[])
    {
        int n = 100;
        int count = 0;
        for (int k = 1; k <= n/2; k++)
            count++;
        System.out.println("Count: " + count);
    }
}
```

Program Output

Count: 50

General count: $n/2$

Example 13.4

```
public class Example134
{
    public static void main(String args[])
    {
        int n = 10;
        int count = 0;
        for (int p = 1; p <= n; p++)
            for (int q = 1; q <= n; q++)
                count++;
        System.out.println("Count: " + count);
    }
}
```

Program Output

Count: 100

General count: $n \times n$ or n^2

Example 13.5

```
public class Example135
{
    public static void main(String args[])
    {
        int n = 10;
        int count = 0;
        for (int p = 1; p < n; p++)
            for (int q = 1; q < n; q++)
                count++;
        System.out.println("Count: " + count);
    }
}
```

Program Output

Count: 81

General count: $(n-1) \times (n-1)$ or $n^2 - 2n + 1$

You can expect questions that present program segments with loop structures that ask how many times a method is called or what value is returned. Such program segments, like the first five examples shown here do not perform a practical function, but it helps to analyze algorithms.

Informal analysis also involves recognizing why one algorithm implementation is more efficient than another implementation. Several implementations will be shown of search algorithms followed by a variety of sort algorithms. At each level, advantages and disadvantages will be pointed out.

All the methods that will be shown are meant to be part of an **IntList** class. **Example 13.6** shows the initial class with its attributes and constructor. The constructor generates a list of random integers in a specified number range. The size of the list and range is provided by the constructor parameters.

Example 13.6

```
class IntList
{
    private int intArray[]; // stores array elements
    private int size;      // number of elements in the array
    private int minInt;    // smallest random integer
    private int maxInt;    // largest random integer

    public IntList(int s, int min, int max)
    {
        Random rndInt = new Random(12345);
        minInt = min;
        maxInt = max;
        size = s;
        intArray = new int[size];
        int range = maxInt - minInt + 1;
        for (int k = 0; k < size; k++)
            intArray[k] = rndInt.nextInt(range) + minInt;
    }
}
```

All the methods could be presented as if they stand alone, but then the storing data structure would become one of the parameters in very poor Object Oriented Design. The **IntList** class is meant to store and process an array of integers. The integers are stored in a static **int** array, called **intArray**. As you look at the methods that follow, you must realize that they belong inside the **IntList** class. Some methods will have several iterations. They are presented intentionally with some poor features. As you observe different implementations, you can analyze features that are bad and appreciate improvements.

The first **linearSearch** method, shown in **Example 13.7**, is inefficient. Each array element is compared to the search number, **sn**. Variable **found** becomes **true** when a match is found. This search works, but it is inefficient since the search continues to make comparisons after the search number is already found.

Example 13.7

```
public boolean linearSearch(int sn)
{
    boolean found = false;
    for (int k = 0; k < size; k++)
        if (intArray[k] == sn)
            found = true;
    return found;
}
```

The next example is an improvement. The **linearSearch** method in **Example 13.8** checks a compound condition, which exits the loop when the search number is found.

Example 13.8

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

The third **linearSearch** method, in **Example 13.9**, is practical. The method does not return true or false. It may be great to know that some number exists in a list, but the real question is where in the list is the number located.

Example 13.9

```
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;
}
```

The methods in **Example 13.10** present the **bubbleSort** method. This implementation also uses a **swap** method. In the bubble sort, adjacent elements in the array are compared and exchanged if they are out of order. This implementation sorts in ascending order.

Example 13.10

```
private void swap(int x, int y)
{
    int temp = intArray[x];
    intArray[x] = intArray[y];
    intArray[y] = temp;
}

public void bubbleSort()
{
    for (int p = 1; p < size; p++)
        for (int q = 0; q < size-p; q++)
            if (intArray[q] > intArray[q+1])
                swap(q,q+1);
}
```

The first bubble sort implementation can be improved by stopping the comparison process when the list is sorted. The **bubbleSort** in **Example 13.11** checks to see if any **swap** method calls are made. Anytime that an entire iteration pass occurs, without any **swap** calls, the list must be sorted. This implementation is often called the *smart* bubble sort. The **swap** method is not included, but it is identical to the **swap** method of the previous example.

Example 13.11

```
public void bubbleSort()
{
    boolean sorted;
    int p = 1;
    do
    {
        sorted = true;
        for (int q = 0; q < size-p; q++)
            if (intArray[q] > intArray[q+1])
            {
                swap(q,q+1);
                sorted = false;
            }
        p++;
    }
    while (!sorted);
}
```

You will probably not see any questions about the *bubble sort* on the APCS Exam. The bubble sort is not that popular in the computer science community. It is considered an inefficient sort, which can be replaced by better sorts. It is included here, because it helps to demonstrate the analysis process and the goal to seek better implementations.

The bubble sort does much processing. Comparisons are made every step along the way and a **swap** routine of three program statements must be executed whenever data is in the wrong order. The **selectionSort** method, shown in **Example 13.12**, eliminates the majority of the swapping. Each array element must still be compared, but now there is a comparison with the largest - or smallest - element found so far. At the end of each pass, the largest element is swapped and placed at the end of the list. There is now only one swap made for one comparison pass.

Example 13.12

```
public void selectionSort()
{
    int p,q, smallest;
    for (p = 0; p < size-1; p++)
    {
        smallest = p;
        for (q = p+1; q < size; q++)
            if (intArray[q] < intArray[smallest])
                smallest = q;
        if (intArray[p] != intArray[smallest])
            swap(p,smallest);
    }
}
```

The **binarySearch** method, in **Example 13.13**, goes to the mid point of a list and makes a comparison. If the search item is not found, half the list is discarded and a midpoint is found in the remaining segment. This process continues very efficiently until the item is found.

Example 13.13

```
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;
}
```

The binary search is not just a little faster than the linear search. Consider a list with one million elements. A linear search could take as many as 1,000,000 comparisons to find the search item. On average, the linear search still takes 500,000 comparisons. The efficient binary search takes no more than 20 comparisons in a worst case scenario. Do keep in mind that the binary search requires that the list sorted.

Another sort to consider is the insertion sort. At first glance the sort, which keeps adding new elements in their correct location, is no faster than the selection sort. That is true, but the selection sort must sort the entire list. An insertion sort can take a few new elements and insert them in an existing, sorted list.

There is a set of sophisticated sorting algorithms that are amazingly fast. These sorts are especially fast when the data list becomes very large and the bubble sort, selection sort and insertion sort start to take an agonizingly long time to sort.

For the AP CS Exam you are not required to know many details about advanced sorts, which are called *quick sort*, *merge sort*, *binary tree sort* and *heap sort*. You are expected to know something about one of these sorts, called the **Merge Sort**. Imagine that you have two separate lists, which are already sorted. Now bring these two lists together and merge them together, somewhat like a zipper. If you have two sets of index cards, each already alphabetized, you have an easier time creating one large alphabetized set than with two sets of random index cards.

However, you have one list, and this list is not sorted. Split your list in two and two again and continue to split the list until you have a whole bunch of little lists. Each tiny list has precisely one element. Now work yourself back up.

Merge two 1-element lists to get a sorted 2-element list.

Merge two 2-element lists to get a sorted 4-element list.

Merge two 4-element lists to get a sorted 8-element list.

Continue this process until the entire list is sorted.

APCS Examination Alert

The AP CS Exam is not overly concerned with memorizing the precise source implementation of an algorithm. The primary concern is to understand the logic and the pseudo code of the algorithm in a language-independent manner.

Students need to have knowledge of the linear search, binary search, selection sort, insertion sort and a realization that more advanced sorts exist. In particular, students need a fundamental understanding of the merge sort.

Chapter 13 Questions

Algorithms and Informal Algorithmic Analysis

01. Consider the following program segment, which is intended to perform a *Linear Search*.

```
boolean found = false;
for (int k = 0; k < list.length; k++)
    if (list[k] == searchNumber)
        found = true;
```

Why is this implementation of the *Linear Search* algorithm considered undesirable?

- (A) Searching will continue after a match for `searchNumber` is found.
- (B) It will only return the correct result if the last array element is the `searchNumber`.
- (C) It will only return the correct result if the first array element is the `searchNumber`.
- (D) It will return the wrong result if the `searchNumber` is not found.
- (E) It will cause an `ArrayIndexOutOfBoundsException`.

02. Which of the following implementations for /* **missing code** */ will improve the undesirable implementation of the *Linear Search*, shown in question 01?

```
boolean found = false;
```

```
/* missing code */
```

Implementation 1

```
for (int k = 0; k < list.length; k++)
    if (list[k] == searchNumber)
        found = true;
    else
        found = false;
```

Implementation 2

```
int k = 0;
while (k < list.length)
{
    if (list[k] == searchNumber)
        found = true;
    k++;
}
```

Implementation 3

```
int k = 0;
while (k < list.length && !found)
{
    if (list[k] == searchNumber)
        found = true;
    k++;
}
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementation 1 and 2
- (E) Implementation 2 and 3

03. What is true about searching sorted data with a *Linear Search*?

- (A) Sorted data can make a *Linear Search* more efficient.
- (B) The arrangement of the data does not alter the *Linear Search* efficiency.
- (C) A *Linear Search* will always be more efficient with sorted data.
- (D) A *Linear Search* will always be less efficient with sorted data.
- (E) A *Linear Search* requires a sorted list of data.

04. Consider method whatSort below.

```
public static void whatSort()
{
    for (int q = 0; q < list.length-1; q++)
        if ( list[q] > list[q+1])
        {
            int temp = list[q];
            list[q] = list[q+1];
            list[q+1] = temp;
        }
}
```

Assume that `list` stores a random list of integers prior to calling method `whatSort`. How will the integers be arranged in the list array after a call to method `whatSort`?

- (A) Data will be arranged in ascending order.
- (B) Data will be arranged in descending order.
- (C) The largest integer will be stored in the last array element.
- (D) The smallest integer will be stored in the first array element.
- (E) The data order will remain unchanged.

05. Consider method `selectionSort` below. Assume that method `swap` exists, which swaps the array elements of `intArray` according to the provided index parameters.

```
public static void selectionSort()
{
    for (int p = 0; p < intArray.length-1; p++)
    {
        int temp = p;
        for (int q = p+1; q < intArray.length; q++)
            if (intArray[q] < intArray[temp])
                temp = q;
        if (intArray[p] != intArray[temp])
            swap(p,temp);
    }
}
```

Assume that `intArray` stores a random list of integers prior to calling method `selectionSort`. How will the numbers be arranged in the `intArray` list after a call to method `selectionSort`?

- (A) The numbers will be arranged in ascending order.
- (B) The numbers will be arranged in descending order.
- (C) The largest integer will be stored in the last array element.
- (D) The smallest integer will be stored in the first array element.
- (E) The data order will remain unchanged.

06. When using a binary search, what is the MAXIMUM number of comparisons necessary to locate a specific item in a list of 1025 elements?

- (A) 1
- (B) 9
- (C) 10
- (D) 11
- (E) 1024

07. Assume that an integer array contains the following set of sorted integers.

{100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200}

How many comparisons are made by a *Binary Search* method to find number 150?

- (A) 1
- (B) 2
- (C) 3
- (D) 4
- (E) 5

08. How can a random list be sorted with a *Merge Sort*, when the merging process only works with sorted lists?
- (A) It cannot and that is why the *Merge Sort* only works with two or more lists that are sorted already.
(B) A secondary sort, like the *Selection Sort*, is used to sort small segments prior to merging them.
(C) A random list can be repeatedly divided until only single member lists are left, which can then be merged, because a single-element list is sorted.
(D) Merging is not an issue, because the sort name incorrectly implies a merging process.
(E) The *Merge Sort* merge only one element at a time to a growing sorted list.
09. What is true about the execution times of the *Linear Search* and the *Binary Search* as the searching list grows larger?
- (A) The execution times for both searches become the same.
(B) The execution times stay the same relative to each other.
(C) The *Binary Search* becomes faster and faster relative to the *Linear Search*.
(D) The *Linear Search* becomes faster and faster relative to the *Binary Search*.
(E) It is not possible to make predictions unless the data list is known.
10. What value is stored by count when the segment below finishes executing?
- ```
int n = <some integer greater than 1>
int count = 0;
for (int p = 1; p < n; p++)
 for (int q = 1; q < n; q++)
 count++;
```
- (A)  $n$   
(B)  $n^2$   
(C)  $n^2 - n$   
(D)  $n^2 - 2n + 1$   
(E)  $n^2 + 2n + 1$

11. What value is stored by count when the segment below finishes executing?

```
int n = <some integer greater than 1>
int count = 0;
for (int p = 1; p <= n; p++)
 for (int q = 1; q <= n-p; q++)
 count++;
```

- (A)  $n^2$  times.
- (B)  $(n^2-n)/2$  times.
- (C)  $n^2/p$  times.
- (D)  $n^2 - 4n + 4$  times.
- (E)  $n/p^2$  times.

12. Consider the behavior of the *Selection Sort* when the data list to be sorted grows.

What is the expected execution time of sorting a list of data, each time that the data list doubles in size.

- (A) As the list size doubles the execution time stays the same.
- (B) As the list size doubles the execution time doubles.
- (C) As the list size doubles the execution time quadruples.
- (D) As the list size doubles the execution time initially quadruples and then steadily takes less time each time the list doubles, until the execution time stays constant.
- (E) As the list size doubles the execution time initially quadruples and then steadily takes less time each time the list doubles, until the execution time doubles.

13. Consider the following search method.

```
public int search(int key)
{
 boolean found = false;
 int index = 0;
 while (index < size && !found)
 {
 if (intArray[index] == key)
 found = true;
 else
 index++;
 }
 return index;
}
```

This search method can be used for

- (A) deletion only.
- (B) insertion only.
- (C) both insertion and deletion with sorted data.
- (D) both insertion and deletion with random data.

14. Consider the following method.

```
public void whichOne(int index, int number)
{
 int size = intArray.length;
 for (int k = size-1; k > index; k--)
 intArray[k] = intArray[k-1];
 intArray[index] = number;
}
```

Assume that `index` represents the array location where a new array elements needs to be inserted or an existing array element needs to be deleted.

Method `whichOne`

- (A) deletes `number` from `intArray`.
- (B) either deletes `number` or inserts `number`; it depends on how method `whichOne` is called.
- (C) either deletes `number` or inserts `number`; it depends on the value of `index`.
- (D) inserts `number` at the end of the array.
- (E) inserts `number` in `intArray` at location `index`.

15. Consider the following method.

```
public void whichOne(int index, int number)
{
 for (int k = index; k < size-1; k++)
 intArray[k] = intArray[k+1];
 size--;
}
```

Assume that `index` represents the array location where a new array elements needs to be inserted or an existing array element needs to be deleted.

Method `whichOne`

- (A) deletes number from `intArray`.
- (B) either deletes number or inserts number; it depends on how method `whichOne` is called.
- (C) either deletes number or inserts number; it depends on the value of `index`.
- (D) inserts number at the end of the array.
- (E) inserts number in `intArray` at location `index`.

# Chapter 14 Review

## Polymorphism

Polymorphism is the third, and final, piece of the Object Oriented Definition presented earlier. It is also the feature of OOP that is the most elusive. The exotic name does not help, but with today's special effects, students know about morphing from one shape into another shape. Morph means shape or form and *polymorphism* means *many forms*. For starters, do be concerned with the technical vocabulary. You must know the vocabulary and be assured that it will show up on the APCS Exam. The key point is not to be intimidated by strange words. The strangeness is due to lack of familiarity.

We now have a feature in computer science in general and Java specifically that has many forms. You have seen some early examples in the form of the plus (+) operator. This operator performs arithmetic addition with numerical values. This same plus operator can also handle string concatenation. In this case the plus operator has many forms. When you first learned about this operator, you probably learned that the operator is overloaded.

Later in the course you learned about methods with identical identifiers and different heading signatures. This is common with constructors and also with some methods. The **substring** method is one example of an *overloaded* method. This too is an example polymorphism.

Now the examples you learned earlier in the course that were called *overloaded* is not at issue here. There are different varieties of polymorphism and this chapter assumes that the *overloaded* operators and methods are in good shape and shall remain with the term *overloaded*.

This chapter is concerned with the ability of a single method to behave in many different ways. This is not a case of different method signatures. A single method is presented and somehow this single method has the amazing ability to behave differently, as appropriate.

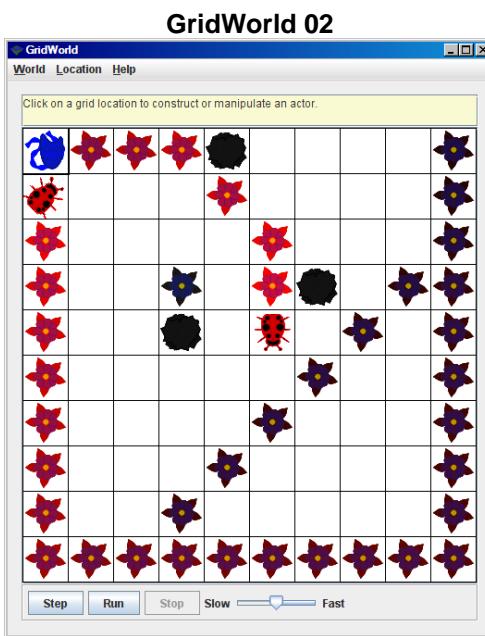
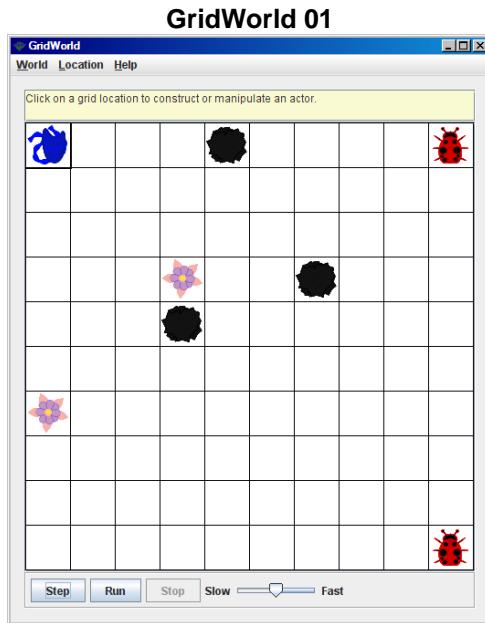
Since you are studying this guide with the assumption that you are preparing for the APCS Exam, another assumption can be made that you are familiar with the current case study. This case study, called GridWorld, was developed and copyrighted by the College Board. It is one of the best examples of the value and use of polymorphism.

In the case study a two-dimensional grid of cells is presented. The cells in the grid contain objects of the **Actor** class or any sub class of the **Actor** class. Every object on the grid *acts* a certain way. The **Actor** class includes an **act** method. It is not very sophisticated. The object makes 180 degree turns continuously.

Other classes are more original. The **Bug** class moves in the direction that it is facing and deposits flowers in its last location. If movement is not possible, the **Bug** object makes clockwise turns of 45 degrees until movement is possible.

The **Rock** class is pretty dull. The **Rock** object sits in place and does nothing, except stand in the way of other objects. This is practical if certain boundaries are necessary. You can add color to **Rock** objects, but it *acts* by doing nothing.

The starting screen of a GridWorld execution is shown by **GridWorld 01**. There is one blue **Actor** object. There are two red **Bug** objects. There are two **Flower** objects and there are three black **Rock** objects. In the picture of **GridWorld 02** you will see the same grid after the **run** button is clicked and each object has *acted* many times.



This brief GridWorld introduction is not sufficient, if you have never heard about the case study. For those students who have familiarity, it is meant as a reminder. Now it is time to get to the point. The GridWorld program has some hidden loop somewhere that *steps* through the program executions. For each step in the loop, every cell is checked. If the cell contains an object, the **act** method of the object is called. The result is many different types of behavior.

Now here is the interesting part. Each object has an **act** method. This **act** method is either a redefinition of the super class **act** method or it has no **act** method defined. If no **act** method is redefined, the **act** method of the super class is used. The result is that every object on the grid knows how to act or behave. This is all the result of calling a modest little program segment, which is part of the **step** method, shown in **Example 14.1**.

**Example 14.1**

```
for (Actor a : actors)
{
 if (a.getGrid() == gr)
 a.act();
}
```

**Code copied from the GridWorld Case Study developed by the College Board.  
AP® Computer Science GridWorld Case Study:  
Copyright© 2005-2006 Cay S. Horstmann (<http://horstmann.com>)**

The **actors** object contains all the objects found on the grid. A check is made if these objects are still present on the current grid, if such is the case, the **act** method of the object is called. Well *polymorphism* is certainly shown here. There is one modest **act** method, and yet this method takes on *many forms*.

This brings up the question, how does Java know which one of these *many forms* need to be executed? That is surprisingly simple. Each **act** call is really an **a.act()** call. In this case, variable **a** is the object found on the grid. Now **a** may be an **Actor** object or any sub class of the **Actor** class, like a **Bug**, **Rock**, **Flower** or **BoxBug**.

It was mentioned earlier that the goal of OOP is reliability, and each of the three OOP features brings reliability to the table. How does *polymorphism* add reliability? Suppose that some outside code segment calls many different objects and tells these objects to behave. Each object is provided with behavior instructions appropriate to the object. That sounds good, but it is quite possible that the wrong instructions are passed on.

On the other hand, if each class is responsible for its own behavior, then such behavior is implemented with the creation of the class. In good program design, the behavior is also tested. Now it is only necessary to tell an actor to go ahead and act. This is somewhat like a play. During rehearsals each actor is told what to say and what to do. The night of the performance, actors are told *to act*. Each actor now performs the job he/she is meant to do. This adds reliability.

The program in **Example 14.2** shows a complete program that demonstrates *polymorphism*. There are three classes, which implement the **Animal** interface. The abstract method of the **Animal** interface is implemented three different ways. When the same **getMove** method is called three times, three different outputs result with three objects of three different classes. The three classes are different, but all three classes implement the same **Animal** interface, which makes this process possible.

### Example 14.2

```
public class Example14.2
{
 public static void main (String args[])
 {
 Animal animal1 = new Mammal();
 Animal animal2 = new Bird();
 Animal animal3 = new Fish();
 ArrayList<Animal> animals = new ArrayList<Animal>();
 animals.add(animal1);
 animals.add(animal2);
 animals.add(animal3);

 for (Animal animal: animals)
 System.out.println(animal.getMove());

 System.out.println();
 }
}

abstract interface Animal
{
 abstract String getMove();
}

class Mammal implements Animal
{
 String animalType;
 public Mammal() { animalType = "Mammal"; }
 public String getMove() { return animalType + " walks."; }
}

class Bird implements Animal
{
 String animalType;
 public Bird() { animalType = "Bird"; }
 public String getMove() { return animalType + " flies."; }
}

class Fish implements Animal
{
 String animalType;
 public Fish() { animalType = "Fish"; }
 public String getMove() { return animalType + " swims."; }
}
```

#### Program Output

```
A Mammal walks.
A Bird flies.
A Fish swims.
```

## Redefinition and Implementation Pitfalls

Polymorphism requires some "umbrella" interface or abstract class. In the previous example the umbrella is the **Animal** class, which is implemented by three classes. The program in **Example 14.3** has no interface. There are three separate classes without any interface to implement or a super classes to extend. Forget this detail and the program does not compile. It recognizes the **Mammal** class and then creates errors.

### Example 14.3

```
public class Example143
{
 public static void main (String args[])
 {
 Mammal animal1 = new Mammal();
 Bird animal2 = new Bird();
 Fish animal3 = new Fish();
 ArrayList<Mammal> animals = new ArrayList<Mammal>();
 animals.add(animal1);
 animals.add(animal2);
 animals.add(animal3);
 for (Mammal animal: animals)
 System.out.println(animal.getMove());
 }
}

class Mammal
{
 String animalType;
 public Mammal()
 {
 animalType = "Mammal";
 }
 public String getMove()
 {
 return animalType + " walks.";
 }
}

class Bird
{
 String animalType;
 public Bird()
 {
 animalType = "Bird";
 }
 public String getMove()
 {
 return animalType + " flies.";
 }
}

class Fish
{
 String animalType;
 public Fish()
 {
 animalType = "Fish";
 }
 public String getMove()
 {
 return animalType + " swims.";
 }
}
```

#### Program Output

```
C:\ExpoJava2007\ExpoJavaA08152007\ChapterProgsA2007\Progs21\Example144.java:17:
cannot find symbol
symbol : method add(Bird)
location: class java.util.ArrayList<Mammal>
 animals.add(animal2);
 ^
C:\ExpoJava2007\ExpoJavaA08152007\ChapterProgsA2007\Progs21\Example144.java:18:
cannot find symbol
symbol : method add(Fish)
location: class java.util.ArrayList<Mammal>
 animals.add(animal3);
 ^
2 errors
```

# Chapter 14 Questions

## Polymorphism

---

01. Why will the program below not compile?

```
public class DS1401
{
 public static void main (String args[])
 {
 Bank tom = new Bank();
 }
}

interface Bank
{
 public double getCheckingBalance();
 public void makeCheckingDeposit(double amount);
 public void makeCheckingWithdrawal(double amount);
}
```

- I. Bank is abstract and cannot be used to instantiate an object.
  - II. The Bank interface is not implemented by a class.
  - III. There are no print or println statements, so there will be no output.
  - IV. There are semicolons (;) at the end of the method headings in Bank.
- 
- (A) I only
  - (B) II only
  - (C) I and II only
  - (D) III and IV only
  - (E) I, II, III and IV

02. Consider the following three interface declarations.  
Which of the three interface declarations is correct?

```
interface Dance1
{
 private String danceMusic;
 public void playMusic();
 public void danceBasic();
}

interface Dance2
{
 public void playMusic();
 public void danceBasic();
}

interface Dance3
{
 public Dance3()
 public void playMusic()
 public void danceBasic()
}
```

- (A) Dance1 only
- (B) Dance2 only
- (C) Dance3 only
- (D) Dance1 and Dance2
- (E) Dance2 and Dance3

03. Consider the following Dance interface.

```
interface Dance
{
 public void playMusic();
 public void danceBasic();
}
```

Which of the following classes correctly declares an implementation of Dance?

```
class Waltz1 implements Dance
{
 private String music;
 public Waltz1(String m) { music = m; }
 public void playMusic() { System.out.println("Play " + music); }
 public void danceBasic()
 {
 System.out.println("Dance the basic box step");
 }
}

class Waltz2 implements Dance
{
 public void playMusic() { System.out.println("Play music"); }
 public void danceBasic()
 {
 System.out.println("Dance the basic box step");
 }
}

class Waltz3 implements Dance
{
 public void playMusic() { System.out.println("Play music "); }
 public void danceBasic()
 {
 System.out.println("Dance the basic box step");
 }
 public void makeRoutine()
 {
 System.out.println("Make dance routine");
 }
}
```

- (A) Waltz1 only
- (B) Waltz2 only
- (C) Waltz1 and Waltz2 only
- (D) Waltz2 and Waltz3 only
- (E) Waltz1, Waltz2 and Waltz3

04. Consider the following three class declarations.  
Which of the three classes is a correct declaration?

```
abstract class Dance1
{
 private String danceMusic;
 public void playMusic();
 public void danceBasic();
}

abstract class Dance2
{
 public void playMusic();
 public void danceBasic();
}

abstract class Dance3
{
 public Dance3();
 public void playMusic();
 public void danceBasic();
}
```

- (A) Dance1 only
- (B) Dance2 only
- (C) Dance3 only
- (D) Dance1, Dance2 and Dance3
- (E) None of the classes is a correct declaration.

05. Consider the following three class declarations.  
Which of the three classes is a correct declaration?

```
abstract class Dance1
{
 private String danceMusic;
 public abstract void playMusic();
 public abstract void danceBasic();
}

abstract class Dance2
{
 public abstract void playMusic();
 public abstract void danceBasic();
}

abstract class Dance3
{
 public abstract Dance3();
 public abstract void playMusic();
 public abstract void danceBasic();
}
```

- (A) Dance1 only
- (B) Dance2 only
- (C) Dance 3 only
- (D) Dance1 and Dance2
- (E) Dance1 and Dance3

06. Consider the following Dance class.

```
abstract class Dance
{
 private String music;
 public Dance(String m) { music = m; }
 public String getMusic() { return music; }
 public abstract void playMusic();
 public abstract void danceBasic();
}
```

Which of the following classes correctly declares a subclass of Dance?

```
class Waltz1 extends Dance
{
 public Waltz1(String m) { super(m); }
 public void playMusic() { System.out.println("Play " + getMusic()); }
 public void danceBasic()
 {
 System.out.println("Dance the basic box step");
 }
}

class Waltz2 extends Dance
{
 public void playMusic() { System.out.println("Play music"); }
 public void danceBasic()
 {
 System.out.println("Dance the basic box step");
 }
}

class Waltz3 extends Dance
{
 public void playMusic() { System.out.println("Play music "); }
 public void danceBasic()
 {
 System.out.println("Dance the basic box step");
 }
 public void makeRoutine()
 {
 System.out.println("Make dance routine");
 }
}
```

- (A) Waltz1 only
- (B) Waltz2 only
- (C) Waltz1 and Waltz2 only
- (D) Waltz2 and Waltz3 only
- (E) Waltz1, Waltz2 and Waltz3

07. Consider the following Dance interface.

```
interface Dance
{
 public void playMusic();
 public void danceBasic();
}
```

Which of the following classes correctly declares an implementation of Dance?

```
class Waltz implements Dance
{
 private String music;
 public Waltz(String m) { music = m; }
 public void playMusic() { System.out.println("Play " + music); }
 public void danceBasic()
 {
 System.out.println("Dance the basic Waltz step");
 }
}

class ChaCha implements Dance
{
 private String music;
 public ChaCha(String m) { music = m; }
 public void playMusic() { System.out.println("Play " + music); }
 public void danceBasic()
 {
 System.out.println("Dance the basic box step");
 }
}

class Tango implements Dance
{
 private String music;
 public Tango(String m) { music = m; }
 public void playMusic() { System.out.println("Play " + music); }
 public void danceBasic()
 {
 System.out.println("Dance the basic box step");
 }
}
```

- (A) Waltz only
- (B) ChaCha only
- (C) Waltz and ChaCha only
- (D) ChaCha and Tango only
- (E) Waltz, ChaCha and Tango

Questions 08-10 refer to the following interface and classes.

```
abstract interface Animal
{
 abstract String getType();
 abstract String getMove();
}

class Mammal implements Animal
{
 String animalType;
 public Mammal() { animalType = "Mammal"; }
 public String getType() { return animalType; }
 public String getMove() { return " walks."; }
}

class Bird implements Animal
{
 String animalType;
 public Bird() { animalType = "Bird"; }
 public String getType() { return animalType; }
 public String getMove() { return " flies."; }
}

public class DS1408
{
 public static void main (String args[])
 {
 Mammal animal1 = new Mammal();
 Bird animal2 = new Bird();
 getData(animal1);
 getData(animal2);
 }

 public static void getData(Mammal obj)
 {
 System.out.println("A " + obj.getType() + obj.getMove());
 }

 public static void getData(Bird obj)
 {
 System.out.println("A " + obj.getType() + obj.getMove());
 }
}
```

08. Class DS1408 uses two `getData` methods, which need to display the animal type and movement. Will these two methods, with the same identifier, compile and correctly display animal type and movement?
- (A) No, the program will not compile due to a duplicate identifier error.  
(B) No, because the Java interpreter cannot distinguish between the `getData` methods.  
(C) No, because type casting to the appropriate class is omitted.  
(D) Yes, because each method has a unique signature heading.  
(E) Yes, because the `Bird` object and `Mammal` object extends the `Animal` interface.

09. It is possible to replace the two `getData` methods, shown in question 08, with a single `getData` method and execute the program with the exact same results. Which of the following `getData` methods can achieve that requirement?

**Implementation 1**

```
public static void getData(Animal obj)
{
 System.out.println("A " + obj.getType() + obj.getMove());
}
```

**Implementation 2**

```
public static void getData(Object obj)
{
 System.out.println("A " + obj.getType() + obj.getMove());
}
```

**Implementation 3**

```
public static void getData(Animal obj)
{
 if (obj == Mammal)
 System.out.println("A " + Mammal.getType() + Mammal.getMove());
 else
 System.out.println("A " + Bird.getType() + Bird.getMove());
}
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementations 1 and 2
- (E) Implementations 1 and 3

10. Questions 08 and 09 used an abstract Animal interface and two implementing classes.  
The same result can be created with an abstract class.  
Which of the following class declarations duplicates the result of the Animal interface?

**Declaration 1**

```
abstract class Animal
{
 public abstract String getType();
 public abstract String getMove();
}
// Classes Mammal and Bird extends Animal and use the same bodies as questions 08-09.
```

**Declaration 2**

```
class Animal
{
 public String getType();
 public String getMove();
}
// Classes Mammal and Bird extends Animal and use the same bodies as questions 08-09.
```

**Declaration 3**

```
abstract class Animal
{
 public String getType();
 public String getMove();
}
// Classes Mammal and Bird implements Animal and use the same bodies as questions 08-09.
```

- (A) Declaration 1 only
- (B) Declaration 2 only
- (C) Declaration 3 only
- (D) Declarations 1 and 2
- (E) Declarations 1 and 3

11. Consider the following interface and classes.

```
interface Shape
{
 public void drawShape();
}

class Circle implements Shape
{
 public void drawShape() { /* code to draw a circle */ }
}

class Square implements Shape
{
 public void drawShape() { /* code to draw a square */ }
}

public class DS1311
{
 public static void main (String args[])
 {
 Circle shape1 = new Circle();
 Square shape2 = new Square();
 }
}
```

The main method uses the `Circle` class and `Square` class to instantiate the `shape1` and `shape2` objects. Which one of the following segments can also instantiate objects `shape1` and `shape2`?

**Segment 1**

```
Shape shape1 = new Circle();
Shape shape2 = new Square();
```

**Segment 2**

```
Object shape1 = new Circle();
Object shape2 = new Square();
```

**Segment 3**

```
Class shape1 = new Circle();
Class shape2 = new Square();
```

- (A) Segment 1 only
- (B) Segment 2 only
- (C) Segment 3 only
- (D) Segment 1 and Segment 2 only
- (E) Segment 1, Segment 2 and Segment 3

12. Consider the following classes.

```
abstract class Shape
{
 public abstract void drawShape();
}

class Circle extends Shape
{
 public void drawShape() { /* code to draw a circle */ }
}

class Square extends Shape
{
 public void drawShape() { /* code to draw a square */ }
}

public class DS1412
{
 public static void main (String args[])
 {
 Circle shape1 = new Circle();
 Square shape2 = new Square();
 }
}
```

The main method uses the `Circle` class and `Square` class to instantiate the `shape1` and `shape2` objects. Which one of the following segments can also instantiate `shape1` and `shape2`?

**Segment 1**

```
Shape shape1 = new Circle();
Shape shape2 = new Square();
```

**Segment 2**

```
Object shape1 = new Circle();
Object shape2 = new Square();
```

**Segment 3**

```
Class shape1 = new Circle();
Class shape2 = new Square();
```

- (A) Segment 1 only
- (B) Segment 2 only
- (C) Segment 3 only
- (D) Segment 1 and Segment 2 only
- (E) Segment 1, Segment 2 and Segment 3

13. Consider the following code segment and classes.

```
MyBank tom = new MyBank(5000.0);
System.out.println("Tom's checking balance: " + tom.getBalance());
tom.makeCheckingDeposit(2500.0);
System.out.println("Tom's checking balance: " + tom.getBalance());
tom.makeCheckingWithdrawal(1500.0);
System.out.println("Tom's checking balance: " + tom.getBalance());

abstract class Bank
{
 public abstract double getBalance();
 public abstract void makeDeposit(double amount);
 public abstract void makeWithdrawal(double amount);
}

class MyBank extends Bank
{
 private double checking;
 public MyBank(double c) {checking = c; }
 public double getBalance() {return checking; }
 public void makeDeposit(double amount) {checking += amount; }
 public void makeWithdrawal(double amount) {checking -= amount; }
}
```

What is the output of this program?

- (A) Tom's checking balance: 5000  
Tom's checking balance: 7500  
Tom's checking balance: 6000
- (B) Tom's checking balance: 5000  
Tom's checking balance: 2500  
Tom's checking balance: 4000
- (C) Tom's checking balance: 4000  
Tom's checking balance: 2500  
Tom's checking balance: 5000
- (D) Tom's checking balance: 6000  
Tom's checking balance: 7500  
Tom's checking balance: 5000
- (E) Tom's checking balance: 5000  
Tom's checking balance: 7500  
Tom's checking balance: 4000

14. Consider the following interface and class declarations.

```
abstract interface Bank
{
 public abstract double getBalance();
 public abstract void makeDeposit(double amount);
 public abstract void makeWithdrawal(double amount);
 public abstract void closeAccount();
}

class MyBank implements Bank
{
 private double checking;
 public MyBank(double c) { checking = c; }
 public double getBalance() { return checking; }
 public void makeDeposit(double amount) { checking += amount; }
 public void makeWithdrawal(double amount) { checking -= amount; }
 public void makeLoan(double amount) { checking += amount; }
}
```

What is incorrect with the MyBank class implementation of the Bank interface?

- (A) The methods in the MyBank class must be declared abstract.
- (B) The checking attribute is not part of the Bank interface.
- (C) The closeAccount abstract method of the Bank interface is not implemented in the MyBank class.
- (D) The makeLoan method of the MyBank class cannot be implemented.
- (E) The checking attribute must be declared as final.

15. Consider the following interface and abstract class declarations.

```
abstract interface Bank1
{
 public final double rate = 0.05;
 public abstract double getBalance();
 public abstract void makeDeposit(double amount);
 public abstract void makeWithdrawal(double amount);
 public abstract void computeInterest();
}

abstract class Bank2
{
 public final double rate = 0.05;
 public abstract double getBalance();
 public abstract void makeDeposit(double amount);
 public abstract void makeWithdrawal(double amount);
 public abstract void computeInterest();
}
```

The declarations of the Bank1 interface and Bank2 abstract class appear identical.

Both declarations involve abstract methods, as they are shown above.

What is true about the requirements of abstract interfaces and abstract classes?

- (A) The keyword `abstract` is optional for Bank1 and required for Bank2.
- (B) The keyword `abstract` is required for Bank1 and optional for Bank2.
- (C) The keyword `abstract` is optional for Bank1 and also optional for Bank2.
- (D) The keyword `abstract` is required for Bank1 and also required for Bank2.
- (E) The use of `abstract` in an interface or an abstract class depends upon the implementing class.

# Chapter 15 Review

## Program Design

In a few short pages it is not possible to "review" program design properly. Program design does not have the concrete descriptions of computer science features that can be demonstrated with a program, like a binary search algorithm. A few fundamental comments will be made here, but please find a good text book that includes program design or do research on this topic.

Especially, check on the College Board web site and read the information about program design that is included with the course description for AP Computer Science. Be especially concerned with *Object Oriented Design*. Today programs are more complicated. Object Oriented Programming was born to handle these huge complicated programs in a reliable manner.

**Table 15.1** shows the four basic steps involved with program design based on Object Oriented Programming.

**Table 15.1**

| Step # | Step Mission                              |
|--------|-------------------------------------------|
| 1      | <b>Understand the Problem Description</b> |
| 2      | <b>Class Design</b>                       |
| 3      | <b>Method Design</b>                      |
| 4      | <b>Class Interaction</b>                  |

Object Oriented Design is great, but it cannot ignore some fundamental rules of program design that have existed for several decades before Object Oriented Programming came on the scene. The rules in **Table 15.2** serve any style of programming well.

**Table 15.2**

| Fundamental Program Design Rules                                                                                                                                                                                                                                                                                              |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Write your program in a clear, consistent style.</li><li>• Use meaningful, self-documenting identifiers.</li><li>• Do not place all your code in one module.</li><li>• Use the <b>one task - one module</b> philosophy.</li><li>• Place common purpose modules in one file.</li></ul> |

How is program design going to be tested on the APCS Examination? Well for starters, you are expected to understand clearly how a program works. In other words, before you can design your own programs, you must be able to understand how somebody else's design works. You are also expected to design a single class. Design is great, but at some point comes implementation and students in a first course must be able to write the method body of the method signature described in a class design. Additionally, you should also be able to write a class that implements an interface or extends another class.

### AP Examination Alert

#### Program design topics for the APCS Examination:

- Comprehend the design of a provided program
- Understand a problem description, purpose and goals
- Apply data abstraction and encapsulation
- Understand class specifications
- Understand "is-a" class relationships
- Understand "has-a" class relationships
- Understand and implement a given class hierarchy
- Identify and use existing class libraries
- Identify reusable code from existing code
- Design a single class
- Design an interface
- Implement a provided interface
- Extend a class

You can expect AP Computer Science Examination questions on the multiple choice segment and also on the free response segment about program design.

### Program Design Pitfalls

Program design is the least objective topic on the APCS Examination. Many questions involve using program code and the free response section requires writing program code. Program code is correct or not correct. Program design is more elusive. It is a mistake to answer design questions based on your opinion of how programs should be created. You need to get an idea how the people creating the APCS Exam view program design. Their approach is to follow acceptable policies established at leading universities by leaders in computer science. It is a good idea to go to the College Board web site and look at the course description for AP Computer Science. This includes valuable information about program design. The information presented in the course description will give you an accurate indication of the feelings about program design that will be reflected on the APCS Exam.

# Chapter 15 Questions

## Program Design

---

01. The *precondition* of a method describes

- (A) what parameters need to be passed to a method.
- (B) what local variables need to be declared in a method.
- (C) the memory requirements of the data used by a method.
- (D) everything that must be true before method execution starts.
- (E) everything that must be true during the execution of a method.

02. The *postcondition* of a method describes

- (A) the memory that is available after executing a method.
- (B) everything that must be true after the precondition is satisfied.
- (C) everything that must be true at the end of executing a method, provided the preconditions are true.
- (D) everything that must be true during the execution of a method.
- (E) the values of the parameters after the execution of a method.

03. You are hired to write a gradebook program for some school district.  
What is the first step in developing this program?

- (A) Get the exact program specifications.
- (B) Make a list of necessary classes and determine class interaction.
- (C) Create the algorithms necessary for the gradebook program.
- (D) Decide on the input/output features of the program.
- (E) Determine which program language is most suitable for the program.

04. In Object Oriented Design, the fundamental component of a program is the

- (A) driver class, which contains the `main` method.
- (B) `main` method of the driver class.
- (C) class interaction.
- (D) object, which has state and operations that access its state.
- (E) use of existing Java class libraries.

05. Consider the following statement.

\_\_\_\_\_ is the concept of thinking about programming features and using programming features without concern, or knowledge, about the implementation of such features.

- (A) information hiding.
- (B) polymorphism.
- (C) encapsulation.
- (D) inheritance.
- (E) composition.

06. A team of programmers works together on a large project. Each team member is assigned to create, write and test one class that will be used in the project. Which of the following is the minimal required information for each programmer to start his/her assigned task?

- I. The data that will need to be stored by the class.
- II. The method signatures that will access the class data.
- III. The preconditions and postconditions of each class method

- (A) I only
- (B) II only
- (C) I and II only
- (D) II and III only
- (E) I, II and III

07. What is the most important client requirement that must be satisfied by any program design?

- (A) The program must execute efficiently.
- (B) The program must be portable.
- (C) The program must be very readable and easy to improve at a later date.
- (D) The program must be intuitive and provide clear on-line help information.
- (E) The program must execute reliably under all circumstances.

08. A data structure needs to store information in about 20,000 college student records. A variety of information needs to be stored in each student record, such as *name*, *age*, *gpa*, *date of birth*, *address* and *social security number*.

The data structure is to be used in a program with the following two data processing requirements:

- [1] Frequent quick access to any student records to update student information.
- [2] End-of-semester access to print a student GPA list from highest to lowest.

Consider the following two methods to implement the two processing requirements.

**Method 1** stores all the student records in a data structure that is sorted according to *GPA* so as to allow efficient processing of a GPA list, but a linear search is required to access any individual student information according to *social security number*.

**Method 2** stores all the student records in a data structure that is sorted according to student *social security number*. The data structure must be sorted according to *GPA* before a sorted *GPA* list can be printed.

Which of the following statements is true about the manner in which **method 1** and **method 2** satisfy the two data processing requirements?

- (A) Method 1 is better, because it keeps the data sorted, such that all 20,000 records can be easily printed according to *GPA* order.
- (B) Method 1 is better, because it takes longer to sort records than it takes to search for one specific record.
- (C) Method 2 is better, because frequent searches take longer than occasional sorting.
- (D) Method 2 is better, because it takes longer to search for one record than to sort all the data.
- (E) Method 1 and method 2 are essentially identical and each works as efficiently as the other does.

09. A soft-drink company needs a program to analyze shopping mall surveys about its current products. The program needs to satisfy the following requirements.

- Input survey data.
- Keep track of the total number of surveys completed.
- Keep track of survey statistics for each individual soft-drink product.
- Compute survey statistics for each individual product.
- Sort soft-drink products according to customer satisfaction.

Your first step in the design of this program is to select a class or classes with appropriate data storage and methods to satisfy the processing requirements. Which of the following is the best design?

- (A) Use one Survey class with data fields numSurveys and productStats, along with methods enterSurvey, computeStats and sortProducts.
- (B) Use one Survey superclass with data fields numSurveys and productStats and three subclasses EnterSurvey, ComputeStats and SortData.
- (C) Use one Survey superclass with data fields numSurveys and productStats and one subclass ProcessData, which includes methods enterSurvey, computeStats and sortData.
- (D) Use four unrelated classes, which are Storage, EnterSurvey, ComputeStats and SortData.
- (E) Create an interface with method signatures for enterSurvey, computeStats and sortData and three classes to implement each abstract method.

10. Consider a CardDeck class. This class needs data fields to store the number of decks and the total number of cards left. Additionally, the class needs the ability to access data and deal hands. Which of the following classes shows the best and most complete Object-Oriented Design?

|                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>(A)</b><br><pre>public class CardDeck {     public int numDecks;     public int numLeft;      public CardDeck(int nd, int nl)     {         numDecks = nd;         numLeft = nl;     }      public void shuffleCards()     { // code to shuffle cards }      public void dealHand()     { // code to deal hands } }</pre>                                                                                                                    | <b>(B)</b><br><pre>public class CardDeck {     private int numDecks;     private int numLeft;      public CardDeck(int nd, int nl)     {         numDecks = nd;         numLeft = nl;     }      public void shuffleCards()     { // code to shuffle cards }      public void dealHand()     { // code to deal hands } }</pre>                                                                                          |
| <b>(C)</b><br><pre>public class CardDeck {     public int numDecks;     public int numLeft;      public CardDeck(int nd, int nl)     {         numDecks = nd;         numLeft = nl;     }      public void shuffleCards()     { // code to shuffle cards }      public void dealHand()     { // code to deal hands }      public int getDecks { return numDecks; }     public int getLeft { return numLeft; } }</pre>                           | <b>(D)</b><br><pre>public class CardDeck {     private int numDecks;     private int numLeft;      public CardDeck(int nd, int nl)     {         numDecks = nd;         numLeft = nl;     }      public void shuffleCards()     { // code to shuffle cards }      public void dealHand()     { // code to deal hands }      public int getDecks { return numDecks; }     public int getLeft { return numLeft; } }</pre> |
| <b>(E)</b><br><pre>public class CardDeck {     private int numDecks;     private int numLeft;      public CardDeck(int nd, int nl)     {         numDecks = nd;         numLeft = nl;         shuffleCards();     }      public void shuffleCards()     { // code to shuffle cards }      public void dealHand()     { // code to deal hands }      public int getDecks { return numDecks; }     public int getLeft { return numLeft; } }</pre> |                                                                                                                                                                                                                                                                                                                                                                                                                         |

11. Consider the following code segment.

```
ArrayList<Integer> list1 = new ArrayList<Integer>();
< program code that enters appropriate values to the list1 object >
ArrayList<Integer> list2 = list1;
< code statements to perform various processing functions >
```

The visible code has some potential problems that can negatively impact the later invisible code.  
Which of the following are potential problems with the visible code segment?

- I. An `ArrayOutOfBoundsException` will occur.
  - II. `list2` makes a *shallow* copy of `list1`, which can result in unwanted changes to the state of `list1`.
  - III. `list2` is never constructed with the `new` operator.
- 
- (A) I only
  - (B) II only
  - (C) III only
  - (D) I and II only
  - (E) I and III only

12. Consider the following situation.

A donut shop is interested in using a program that keeps track of how many and what type of donuts are sold each day of the year. At the conclusion of the year, the program needs to quickly access data for any given day and also provide information about the total quantity of donuts sold in various categories. For the purpose of this question assume that only glazed, chocolate and jelly donuts are sold. Which of the following code segments would be most suitable for such a program? The classes below do not have any constructors or accessing methods. Your consideration is limited to the manner in which the required data is stored.

|                                                                                                                                                                                       |                                                                                                                                                                                                                                                                                |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| (A)<br><pre>class DonutDay {     private int glazed;     private int chocolate;     private int jelly; }  class DonutStats {     private ArrayList&lt;DonutDay&gt; donutYear; }</pre> | (B)<br><pre>class DonutDay {     private int glazed;     private int chocolate;     private int jelly; }  class DonutStats {     private ArrayList&lt;DonutDay&gt; donutYear;     private int totalGlazed;     private int totalChocolate;     private int totalJelly; }</pre> |
| (C)<br><pre>class DonutStats {     private int donutDay[];     private donutDay donutYear[]; }</pre>                                                                                  | (D)<br><pre>class DonutStats {     private int donutYear[][];</pre>                                                                                                                                                                                                            |
| (E)<br><pre>class DonutStats {     private int donutYear[][];     private int totalGlazed;     private int totalChocolate;     private int totalJelly; }</pre>                        |                                                                                                                                                                                                                                                                                |

13. An electrician's toolbox can be compared to

- (A) a class.
- (B) a method.
- (C) a program.
- (D) an IDE.
- (E) an interface.

14. An individual tool in an electrician's toolbox can be compared to

- (A) a class.
- (B) a method.
- (C) a program.
- (D) an IDE.
- (E) an interface.

15. Which Java methods must always be declared public?

- (A) Static methods
- (B) Helper methods
- (C) Constructor methods
- (D) Object methods
- (E) Abstract methods

# Chapter 16 Questions

## Sample AP Examination I

---

### Section I

**Time** - 1 hour and 15 minutes

**Number of questions** - 40

**Percent of total grade** - 50

01. Consider the following method.

```
// precondition: m > 0
public static int mystery(int m)
{
 int counter = 0;
 for (int p = 1; p < m; p++)
 for (int q = 0; q < m-p; q++)
 counter++;
 return counter;
}
```

What value is returned as a result of the call `mystery(10)`?

- (A) 45
- (B) 50
- (C) 64
- (D) 81
- (E) 100

02. The boolean expression

`A && ((B || C) && (B || D))`

evaluates to

- (A) true, whenever A equals true.
- (B) false, whenever A equals false.
- (C) true, whenever both C equals true and also D equals true.
- (D) true, whenever either C equals true or D equals true.
- (E) false, whenever either C equals false or D equals false.

03. Consider the following method.

```
public static void mystery(int[] list)
{
 int n = list.length - 1;
 for (int k = 0; k <= n; k++)
 if (k > list.length/2)
 list[k] = list[n - k];
}
```

Assume that an integer array contains the following values: {11, 22, 33, 44, 55, 66, 77, 88, 99}. Which of the following represents the contents of `list` as a result of calling method `mystery`?

- (A) {11, 22, 33, 44, 55, 99, 88, 77, 66}
- (B) {55, 44, 33, 22, 11, 22, 33, 44, 55}
- (C) {99, 88, 77, 66, 55, 66, 77, 88, 99}
- (D) {11, 22, 33, 44, 55, 44, 33, 22, 11}
- (E) {11, 22, 33, 44, 55, 66, 77, 88, 99}

Questions 04-05 refer to the following Student class.

```
class Student implements Comparable
{
 private String name;
 private int age;

 public Student (String n, int a)
 {
 name = n;
 age = a;
 }

 /* missing compareTo method */

 /* missing toString method */
}
```

04. The Student class implements the Comparable interface. Student objects need to be compared according to their age standing. The compareTo method needs to be implemented as follows:

```
student1.compareTo(student2) returns >0 if student1 age is greater than student2 age.
student1.compareTo(student2) returns 0 if student1 age equals student2 age.
student1.compareTo(student2) returns <0 if student1 age is less than student2 age.
```

Which of the following three implementations will make method compareTo work as intended?

**Implementation 1**

```
public int compareTo(Object source)
{
 if (age > source.age)
 return 1;
 else if (age == source.age)
 return 0;
 else
 return -1;
}
```

**Implementation 2**

```
public int compareTo(Object source)
{
 Student temp = (Student) source;
 if (age > temp.age)
 return 1;
 else if (age == temp.age)
 return 0;
 else
 return -1;
}
```

**Implementation 3**

```
public int compareTo(Object source)
{
 Student temp = (Student) source;
 return age - temp.age;
}
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementations 1 and 2 only
- (E) Implementations 2 and 3 only

05. The `Student` class redefines the `toString` method. `Student` objects need to be displayed between square brackets, including both the `name` field and the `gpa` field separated by a hyphen.

Consider the following two `Student` object instantiations:

```
Student student1 = new Student("Tom", 23);
Student student2 = new Student("Sue", 21);
```

The `toString` method needs to be implemented to print the following output.

|                                            |                 |              |
|--------------------------------------------|-----------------|--------------|
| <code>System.out.println(student1);</code> | needs to print: | [ Tom - 23 ] |
| <code>System.out.println(student2);</code> | needs to print: | [ Sue - 21 ] |

Which of the following three implementations will make method `toString` work as intended?

**Implementation 1**

```
public String toString()
{
 return "[" + name + " - " + age + "]";
}
```

**Implementation 2**

```
public String toString()
{
 String temp = "";
 temp += "[";
 temp += name;
 temp += " - ";
 temp += age;
 temp += "]";
 return temp;
}
```

**Implementation 3**

```
public String toString()
{
 System.out.println("[" + name + " - " + age + "]");
}
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementations 1 and 2 only
- (E) Implementations 1, 2 and 3

06. Consider the following method.

```
public static String weirdString (String str)
{
 String temp = "";
 for (int k = 0; k < str.length(); k++)
 {
 if (k % 2 != 0)
 temp += str.substring(k,k+1);
 else
 temp += "Q";
 }
 return temp;
}
```

What value is returned as a result of the call `weirdString( "COMPUTER" )` ?

- (A) QOQPQTQR
- (B) CQMQUQEQQ
- (C) QQMPQQER
- (D) COQQUTQQ
- (E) QQQQQQQQ

07. Consider the following code segment.

```
int n = <some int > 0>
int rndNum = (int) (Math.random() * n) + n;
```

What is the range of random numbers that can be assigned to `rndNum`?

- (A) [ 0 .. n ]
- (B) [ 0 .. 2n-1 ]
- (C) [ n .. 2n-1 ]
- (D) [ n .. 2n ]
- (E) [ n .. 2n + 1 ]

08. Imagine that you are creating a graphics program that involves moving 3d objects. The program requires complex mathematical operations. All these mathematical operations have been placed in a special Math3d class.

Which of the following Math3d class information is required to use the class effectively?

- I. The preconditions and post conditions of each public method.
- II. The method's heading signature.
- III. The mathematical logic and implementation source code of each method.

- (A) I only
- (B) II only
- (C) I and II only
- (D) II and III only
- (E) I, II and III

09. Consider the following two methods.

```
public static int holland(int x, int y)
{
 int temp = belgium(x,y);
 return x / temp * y;
}

public static int belgium(int a, int b)
{
 int temp = a % b;
 if (temp == 0)
 return b;
 else
 return belgium(b,temp);
}
```

What value is returned as a result of the call `holland(108,120)` ?

- (A) 12
- (B) 24
- (C) 240
- (D) 540
- (E) 1080

10. Consider the following code segment.

```
ArrayList<Integer> list1 = new ArrayList<Integer>();
list1.add(new Integer(111));
list1.add(new Integer(222));
list1.add(new Integer(333));
list1.add(new Integer(444));
ArrayList<Integer> list2 = list1;
list1.set(1,new Integer(555));
list2.set(2,new Integer(666));
System.out.println(list1);
System.out.println(list2);
```

What is printed as a result of executing the segment?

- (A) [111, 555, 666, 444]  
[111, 555, 666, 444]
- (B) [111, 555, 333, 444]  
[111, 222, 666, 444]
- (C) [555, 666, 333, 444]  
[555, 666, 333, 444]
- (D) [111, 222, 666, 444]  
[111, 555, 333, 444]
- (E) [111, 555, 666, 222, 333, 444]  
[111, 555, 666, 222, 333, 444]

11. Consider the following three classes.

```
class Engine
{
 private double literSize;
 private int horsePower;
 public Engine(double ls, int hp)
 {
 literSize = ls;
 horsePower = hp;
 System.out.println("Engine object is constructed");
 }
}

class Van
{
 private String companyName;
 private String carType;
 private Engine engine;
 public Van(String cn, String ct, double ls, int hp)
 {
 companyName = cn;
 carType = ct;
 engine = new Engine(ls,hp);
 System.out.println("Van object is constructed");
 }
}

class CamperVan extends Van
{
 private boolean kitchen;
 private int sleepingSpaces;
 private int captainSeats;
 public CamperVan(String cn, String ct, double ls, int hp, boolean k, int ss, int cs)
 {
 super(cn,ct,ls,hp);
 kitchen = k;
 sleepingSpaces = ss;
 captainSeats = cs;
 System.out.println("CamperVan object is constructed");
 }
}
```

Assume that the following code segment appears in a client program. What is printed by the code segment?

```
CamperVan model123 = new CamperVan("Chrysler", "SuperCaravan", 4.5, 285, true, 4, 4);
```

- (A) CamperVan is constructed
- (B) CamperVan object is constructed  
Van object is constructed
- (C) CamperVan object is constructed  
Van object is constructed  
Engine object is constructed
- (D) Engine object is constructed  
Van object is constructed  
CamperVan object is constructed
- (E) Engine object is constructed  
CamperVan object is constructed  
Van object is constructed

12. Consider the following method

```
public static void taiwan(int[][] m)
{
 int rSize = m.length-1;
 int cSize = m[0].length-1;
 for (int r = 0; r <= rSize; r++)
 {
 for (int c = 0; c <= cSize/2; c++)
 {
 int t = m[r][cSize-c];
 m[r][cSize-c] = m[r][c];
 m[r][c] = t;
 }
 }
}
```

Which of the following correctly describes the integers stored by `m` after a call to method `taiwan` ?

- (A) The top row is moved to the bottom.
- (B) All the rows are in reverse order.
- (C) All the columns are in reverse order.
- (D) The matrix is reversed both in rows and columns.
- (E) The matrix is unchanged.

13. Consider the following code segment.

```
for (int k = 1; k <= 10; k++)
{
 int rndInt = (int) (Math.random() * 900) + 100;
 System.out.print(rndInt + " ");
}
```

What is printed as a result of executing the code segment?

- (A) A different set of random integers in the [100..900] range for each separate execution.
- (B) A different set of random integers in the [100..1000] range for each separate execution.
- (C) A different set of random integers in the [100..999] range for each separate execution.
- (D) An identical set of random integers in the [100..999] range for each separate execution.
- (E) An identical set of random integers in the [100..1000] range for each separate execution.

14. Consider the following three classes.

```
class Vehicle
{}

class Car extends Vehicle
{}

class Ambulance extends Vehicle
{}

class Truck extends Vehicle
{}
```

Which of the following implementations of method `addObjects` correctly adds objects of the `Car`, `Ambulance` and `Truck` classes to the `list` array?

**Implementation 1**

```
public static void addObjects(Vehicle x)
{
 ArrayList<Vehicle> list = new ArrayList<Vehicle>();
 list.add(x);
}
```

**Implementation 2**

```
public static void addObjects(Car,Ambulance,Truck x)
{
 ArrayList<Car,Ambulance,Truck> list = new ArrayList<Car,Ambulance,Truck>();
 list.add(x);
}
```

**Implementation 3**

```
public static void addObjects1(Car c)
{
 ArrayList<Car> list = new ArrayList<Car>();
 list.add(c);
}
public static void addObjects2(Ambulance a)
{
 ArrayList<Ambulance> list = new ArrayList<Ambulance>();
 list.add(a);
}
public static void addObjects3(Truck t)
{
 ArrayList<Truck> list = new ArrayList<Truck>();
 list.add(t);
}
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementation 1 and 3 only
- (E) Implementation 2 and 3 only

15. Consider the following class.

```
class Stuff implements Comparable
{
 private int stuff;

 public Stuff (int s)
 {
 stuff = s;
 }

 public int getStuff()
 {
 return stuff;
 }

 public void setStuff(int s)
 {
 stuff = s;
 }

 public int compareTo(Object source)
 {
 Stuff temp = (Stuff) source;
 if (temp.getStuff() < this.stuff)
 return -1;
 else if (temp.getStuff() == this.stuff)
 return 0;
 else
 return 1;
 }
}
```

Assume that the following code segment appears in a client program.  
What is printed by the code segment?

```
Stuff s1 = new Stuff(50);
Stuff s2 = new Stuff(100);
System.out.println(s1.compareTo(s2));
```

- (A) 50
- (B) -50
- (C) -1
- (D) 0
- (E) 1

16. The primary benefit of Object Oriented Programming is greater program reliability.

Which of the following programming features are used by OOP in achieving this reliability benefit?

- I. Encapsulation, which keeps data and accessing methods in the same module
  - II. Inheritance, which allows the use of established, and tested methods, in sub classes.
  - III. Polymorphism, which manages methods with identical method signatures to behave according to their own class implementation.
  - IV. Separation, which keeps data and accessing methods in separate modules
- (A) IV only  
(B) I and II only  
(C) I and III only  
(D) II, III and IV only  
(E) I, II and III only

17. Objects process data like airports process luggage and travelers.

Does that mean that an airport is an analogy of Object Oriented Design in the manner that it processes travelers and luggage to its final destination?

Which feature of Object Oriented Programming is demonstrated by an airport?

- (A) encapsulation  
(B) inheritance  
(C) composition  
(D) polymorphism  
(E) An airport does not process travelers and luggage in an Object Oriented manner.

18. Consider the following method.

```
public static void ifElse(int n)
{
 if (n == 1)
 {
 System.out.print(n + " ");
 ifElse(n - 1);
 }
 else if (n == 2)
 {
 System.out.print(n + " ");
 ifElse(n - 2);
 }
 else if (n == 3)
 {
 System.out.print(n + " ");
 ifElse(n - 3);
 }
 else if (n == 4)
 {
 System.out.print(n + " ");
 ifElse(n - 4);
 }
 System.out.print(n + " ");
}
```

Which of the following is printed as a result of the call `ifElse(4)` ?

- (A) 0
- (B) 4
- (C) 4 0
- (D) 4 0 4
- (E) Displays exception error, because there is no proper base case to exit the method.

19. Consider the following classes.

```
class Fruit
{
 private int quantity;

 public Fruit(int q)
 {
 quantity = q;
 }

 public int getQuantity()
 {
 return quantity;
 }
}

class Apple extends Fruit
{
 private String name;

 public Apple(String n, int q)
 {
 super(q);
 name = n;
 }

 public String toString()
 {
 return getQuantity() + " " + name + " Apples";
 }
}
```

Assume that the following code segment appears in a client program.  
What is printed by the code segment?

```
Apple rd = new Apple("Red Delicious",1000);
System.out.println(rd);
```

- (A) Red Delicious 1000 Apples
- (B) 1000 null Apples
- (C) 1000 Red Delicious Apples
- (D) 0 null Apples
- (E) 0 0 Apples

20. Consider the following method.

```
// precondition: number > 0
public static String monkey(int nr)
{
 String temp = "]";
 while (nr > 0)
 {
 temp = nr % 2 + temp;
 nr /= 2;
 }
 return "[" + temp;
}
```

What string is returned as a result of the call `monkey(n)` ?

- (A) A string displaying the binary numbers of `nr`, like [ 1 2 4 8 16 ... ]
- (B) A string containing the sum of the binary numbers of `nr`, like [1+2+4+8+16+...]
- (C) A string containing the base-2 representation of `nr`
- (D) A string containing the reverse of the base-2 representation of `nr`
- (E) A string containing all the factors of `nr`

21. Consider the following method.

```
// precondition: p < q
public static int frog(int p, int q)
{
 int n = 1;
 while (n < p)
 {
 while (n < q)
 {
 n++;
 }
 }
 return n;
}
```

What value is returned as a result of the call `frog(a,b)` where `a > 1` and `b > 1` ?

- (A) `a + b`
- (B) `a * b`
- (C) `(a-1) * (b-1)`
- (D) `a`
- (E) `b`

22. Consider the following method.

```
// precondition: p and q are positive integers
public static boolean mambo(int p, int q)
{
 return (p % q) == (p/q);
}
```

Assume that the precondition of `mambo` is satisfied.

For which values of `p` and `q` is `true` returned by `mambo`?

- (A) For all odd values of `p` and `q`
- (B) For all even values of `p` and `q`
- (C) For all binary (1, 2, 4, 8, 16, etc.) values of `p`
- (D) For all binary (1, 2, 4, 8, 16, etc.) values of `q`
- (E) For all values of `p`, such that  $p = n * q + r$ , where  $n > 0$  and  $r < q$

23. Consider the following method.

```
public static void removeNr(ArrayList<Integer> list)
{
 for (int k = list.size()-1; k >= 0; k--)
 {
 Integer temp = k;
 if (temp.intValue() % 2 == 0)
 list.remove(k);
 }
}
```

Which of the following describes the result of the call `removeNr(list)`?

- (A) All odd integer values are replaced with `null`.
- (B) All even integer values are replaced with `null`.
- (C) All odd integer values are deleted from the `list` array.
- (D) All even integer values are deleted from the `list` array.
- (E) `list` is an empty array.

24. Consider the following method.

```
public static String sillyString(String s1, String s2)
{
 String temp = s1 + s2;
 return temp.substring(s1.length(),temp.length());
}
```

What value is returned as a result of the call `sillyString(string1,string2)` ?

- (A) `string1 + string2`
- (B) `string2`
- (C) `string1`
- (D) `string2` minus the last character
- (E) `string1` minus the last character

25. Consider the following code segment.

```
ArrayList<Double> randomList = new ArrayList<Double>();
int n = <some int > 0
for (int k = 1; k < n; k++)
{
 Double rndDbl = new Double(Math.random());
 randomList.add(rndDbl);
}
double temp = 0;
for (Double nr : randomList)
 temp += nr.doubleValue();
System.out.println(temp/n);
```

What is printed as a result of executing the code segment?

- (A) A double value  $x$ , such that  $0 \leq x < 1$
- (B) A double value  $x$ , such that  $0 \leq x \leq n$
- (C) A double value  $x$ , such that  $0 \leq x < 0$
- (D) A double value  $x$ , such that  $0 \leq x < 10/n$
- (E) A double value  $x$ , such that  $0 \leq x < n$

26. Consider the following method.

```
// Precondition: p > 0; q > 0; p != q
public static int indonesia(int p, int q)
{
 if (p%q > 0)
 return q;
 else
 return indonesia(q,p%q);
}
```

What value is returned as a result of the call `indonesia(a,b)` ?

- (A) a
- (B) b ArithmeticException error
- (C) a % b
- (D) a + b
- (E) a \* b

27. Consider the following interface and classes declarations.

```
abstract interface AAA
{
 public abstract int getData();
 public abstract void showData();
}

abstract class BBB implements AAA
{
 private int number;
 public BBB(int n) { number = n; }
 public int getData() { return number; }
 public abstract void setData();
}

class CCC extends BBB
{
 public CCC(int n) { super(n); }
 public void showData() { System.out.println(getData()); }
}
```

Will these three declarations compile?

- (A) Yes, all three declarations are correct.
- (B) No, abstract class BBB cannot implement an interface.
- (C) No, abstract class BBB cannot declare a constructor.
- (D) No, class CCC cannot pass information to an abstract class.
- (E) No, class CCC fails to implement method `setData`.

28. Consider the following code segment and method `convertArrays` specifications.

```
ArrayList<Integer> list1 = new ArrayList<Integer>();
list1.add(new Integer(1000));
list1.add(new Integer(2000));
list1.add(new Integer(3000));
int[] list2 = convertArrays(list1);
```

Method `convertArrays` takes the int values stored in a dynamic array parameter, adds them to a static array object capable of storing as many integers as required, and returns the static array object.

Which of the following methods correctly implements `convertArrays`?

**Implementation 1**

```
public static int[] convertArrays(ArrayList list)
{
 int[] temp = new int[list.size()];
 int index = 0;
 for (Integer num : list)
 {
 temp[index] = num.intValue();
 index++;
 }
 return temp;
}
```

**Implementation 2**

```
public static int[] convertArrays(ArrayList<Integer> list)
{
 int[] temp = new int[list.size()];
 int index = 0;
 for (Integer num : list)
 {
 temp[index] = num.intValue();
 index++;
 }
 return temp;
}
```

**Implementation 3**

```
public static int[] convertArrays(ArrayList<Integer> list)
{
 int[] temp = new int[100];
 for (int index = 0; index < list.size(); index++)
 temp[index] = list.get(index).intValue();
 return temp;
}
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementation 1 and 3 only
- (E) Implementation 2 and 3 only

29. Consider the following two classes.

```
class Eye
{
 private Person person;

 /* constructor */
}

class Person
{
 private Color clr;

 /* constructor */
}
```

What type of relationship is established by these two classes?

- (A) Inheritance, such that Person is-an Eye
- (B) Inheritance, such that Eye is-a Person
- (C) Composition, such that Eye has-a Person
- (D) Composition, such that Person has-an Eye
- (E) Relationships between classes cannot be established without constructor definitions

30. Consider the following code segment.

```
System.out.println(3 + 5 * (4 + 6) * 2 + 7);
```

What is printed as a result of executing the code segment?

- (A) 83
- (B) 167
- (C) 110
- (D) 1120
- (E) 144

31. Consider the following two methods.

```
public static boolean method1(int n)
{
 System.out.println("Calling method 1");
 return n < 10;
}

public static boolean method2(int n)
{
 System.out.println("Calling method 2");
 return n < 10;
}
```

Assume that the following code segment appears in a client program.

```
for (int n = 9; n < 11; n++)
{
 if (method1(n) && method2(n))
 System.out.println();
}
```

What is printed as a result of executing the code segment?

(A) Calling method1  
Calling method2

Calling method1  
Calling method2

(B) Calling method1  
Calling method2

(C) Calling method1  
Calling method1

(D) Calling method2  
Calling method2

(E) Calling method1  
Calling method2

Calling method1

32. Consider the following code segment.

```
boolean bool = <true or false>
bool = (bool == true || bool == false);
System.out.println(bool);
```

What is printed as a result of executing the code segment?

- (A) The code segment always prints `true`.
- (B) The code segment always prints `false`.
- (C) The code segment prints `true`, if the initial value of `bool` is `true` and `false` otherwise.
- (D) The code segment prints `true`, if the initial value of `bool` is `false` and `true` otherwise.
- (E) The code segment will print `true` or `false`, but the actual value cannot be determined.

33. Consider the following method.

```
public static void salsa(int[] list)
{
 for (int p = 0; p < list.length-1; p++)
 {
 int temp1 = p;
 for (int q = p+1; q < list.length; q++)
 if (list[q] > list[temp1])
 temp1 = q;
 int temp2 = list[p];
 list[p] = list[temp1];
 list[temp1] = temp2;
 }
}
```

Assume that the following code segment appears in a client program.

```
int[] intArray = {66,22,77,11,44,55,33,99,88};
salsa(intArray);
for (int k = 0; k < intArray.length; k++)
 System.out.print(intArray[k] + " ");
```

What is printed as a result of executing the code segment?

- (A) 88 99 33 55 44 11 77 22 66
- (B) 99 88 77 66 55 44 33 22 11
- (C) 11 22 33 44 55 66 77 88 99
- (D) 66 22 44 88
- (E) 77 11 33 99

34. The linear search algorithm does not require that a list of data is sorted in any order.  
Finding data in an array of  $n$  randomly ordered elements requires, on average,  $n/2$  comparisons.

Can the efficiency of the linear search improve if the algorithm is used on a sorted list?

- (A) No, it does not improve.
- (B) Yes, it improves. Regardless of the sorting order, finding the requested data is faster.
- (C) Yes, it improves, but only if the list is sorted in ascending order.
- (D) Yes, it improves, but only if the list is sorted in descending order.
- (E) No, it does not improve if the search item exists.  
Yes, it improves, if the search item does not exist.

35. Consider the following interface.

```
interface Orps
{
 public int getOrps();
 public void setOrps(int w);
}
```

Which of the following three implementations of interface Orps is correct?

```
class OrpsA implements Orps
{
 private int numOrps;
 public OrpsA(int nd) { numOrps = nd; }
 public int getOrps() { return numOrps; }
 public void setOrps(int nd) { numOrps = nd; }
}

class Orps implements OrpsB
{
 private int numOrps;
 public OrpsB(int nd) { numOrps = nd; }
 public int getOrps() { return numOrps; }
 public void setOrps(int nd) { numOrps = nd; }
}

class OrpsC implements Orps
{
 private int numOrps;
 public OrpsC(int nd) { numOrps = nd; }
 public int getOrpsC() { return numOrps; }
 public void setOrpsC(int nd) { numOrps = nd; }
}
```

- (A) Class OrpsA only
- (B) Class OrpsB only
- (C) Class OrpsC only
- (D) Classes OrpsA and OrpsB only
- (E) Classes OrpsA, OrpsB and OrpsC

36. A search item exists in an array of  $n$  randomly arranged elements.  
How many comparisons, on average, will be made by a linear search method?

- (A)  $n$  comparisons
- (B)  $n/2$  comparisons
- (C)  $n^2$  comparisons
- (D)  $n^2/2$  comparisons
- (E)  $n - n/2$  comparisons

37. Consider the following generic `MyList` declaration.

```
class MyList<E>
{
 int size;
 int index;
 Object[] elements;

 public MyList(int s)
 {
 size = s;
 index = 0;
 elements = new Object[size];
 }

 public void add(E obj)
 {
 elements[index] = obj;
 index++;
 }

 public E remove()
 {
 index--;
 return (E) elements[index];
 }
}
```

Which one of the following program statements uses the `MyList` class properly?

**Segment 1**

```
MyList<String> students = new MyList<String>(100);
```

**Segment 2**

```
MyList<Integer> numbers = new MyList<Integer>(50);
```

**Segment 3**

```
MyList<E> fractions = new MyList<Double>(50);
```

- (A) Segment 1 only
- (B) Segment 2 only
- (C) Segment 3 only
- (D) Segments 1 and 2 only
- (E) Segments 1, 2 and 3

38. Consider the following two Boolean expressions.

`a && (a || b)`

`a || (a && b)`

For which values of `a` and `b` are the two expressions equivalent?

- (A) For all values of `a` and `b`
- (B) For no values of `a` and `b`
- (C) Only whenever `a` is true
- (D) Only whenever `b` is true
- (E) Only whenever both `a` is true and `b` is true.

39. Consider the following code segment.

```
String[] list = {"John", "Greg", "Maria", "Heidi"};
for (int k = 0; k < list.length-1; k++)
{
 list[k] = list[k+1];
 list[k+1] = list[k];
}
for (int k = 0; k < list.length; k++)
 System.out.print(list[k] + " ");
```

What is output as a result of executing the code segment?

- (A) John John John John
- (B) Greg Greg Greg Greg
- (C) Greg Maria Heidi John
- (D) Greg Maria Heidi Heidi
- (E) Heidi Maria Greg John

40. Consider the following ordered list of numbers.

11 23 27 31 32 41 45 45 50 59 62 67 74 83 87 91 93 94 98

How many comparisons will be made by a binary search method to find element 83?

- (A) 2
- (B) 3
- (C) 4
- (D) 5
- (E) 6

# Chapter 16 Free Response Questions

## Sample AP Examination I

---

### Section II

**Time** - 1 hour and 45 minutes

**Number of questions** - 4

**Percent of total grade** - 50

#### Question 1.

There are three measures of central tendency in statistics. One of the measures of central tendency is the arithmetic mean, which is calculated by adding all of the numbers and dividing by the quantity of the numbers. A second measure of central tendency is the median, which is the middle value of a list of  $n$  ordered numbers. If  $n$  is even, the median is the mean of the two middle numbers.

#### Part (a).

Write method `createList`, which constructs an `ArrayList` object, generates a set of random integers in the **[10..99]** range, stores these integers in the `ArrayList` object, and returns the array object.

Complete method `createList` below.

```
// precondition: n > 0
// postcondition: Returns an ArrayList object with n Integer values.
// Each Integer object is created with a random int
// in the [10..99] range.
public static ArrayList<Integer> createList (int n)
```

**Part (b).**

Write method `getMean`, which is described as follows. Method `getMean` returns the average, called the **mean**, of a set of numbers. The **mean** is computed by adding the numbers and dividing the sum by the quantity of the numbers.

For example:

The **mean** of {10, 18, 17} is **15**

The **mean** of {10, 15} is **12.5**

Complete method `getMean` below.

```
// precondition: list is a non-empty array with Integer objects.
// postcondition: Returns the mean of the int values stored
// by the list array.
public static double getMean (ArrayList<Integer> list)
```

Part (c).

Write method `getMedian`, which is described as follows. Method `getMedian` returns a measure, called the median, of a set of `n` numbers. The median is the middle number in a sorted list of numbers. If `n` is even, the median is the mean of the two middle numbers.

For example:

The **median** of {10, 13, 17, 18, 21} is **17**.

The **median** of {54, 60, 64, 68, 74, 81, 94} is **68**

The **median** of {10, 11, 13, 17, 19, 21} is **15**.

The **median** of {50, 60, 70, 80} is **65**

In writing method `getMedian` you may call method `sortList` specified below.

```
// precondition: list is a non-empty ArrayList object of int elements.
// postcondition: The int values in list are sorted in ascending order.
public static void sortList (ArrayList list)
```

Complete method `getMedian` below.

```
// precondition: list is a non-empty ArrayList object of int elements.
// The int elements in list are randomly ordered.
// postcondition: Returns the median value of list.
public static double getMedian (ArrayList<Integer> list)
```

## Question 2.

This question involves identifying palindromes. A *palindrome* is a string of characters that reads the same from front-to-back and back-to-front. Three examples of palindromes are **racecar**, **madam** and **123454321**.

These first three examples are perfect palindromes. It is possible to read the string of characters from either end with the exact same results, without any special conditions. There are other strings of characters, which can be considered palindromes, but they are not perfect. Three examples of imperfect palindromes are **Racecar**, **Madam I'm Adam** and **A man, a plan, a canal, Panama**. **Racecar** can be considered a palindrome if case-sensitivity is ignored. The other two examples are also palindromes if case-sensitivity, spaces and punctuation are ignored.

For this question a palindrome shall be defined as a *string of alpha-numeric characters that reads the same from front-to-back and back-to-front without case-sensitivity*. This definition means that strings like **Racecar** and **Madam** will be considered palindromes, but **Madam I'm Adam** is not a palindrome.

Consider the following incomplete declaration of a **Palindrome** class. The constructor determines if the parameter is a palindrome by using the **isPalindrome** method. The private method **isPalindrome** needs to return true if the string is a palindrome and false otherwise. The method **toString** needs to be redefined to produce the output shown in the execution example.

```
class Palindrome
{
 private boolean palindrome;
 private String str;

 Palindrome(String s)
 {
 str = s;
 palindrome = isPalindrome();
 }

 public String toString()
 {
 /* to be redefined in part (a) */
 }

 private boolean isPalindrome()
 {
 /* to be implemented in part (b) */
 }
}
```

The code segment below is part of a client program that constructs and displays two **Palindrome** objects.

| Code segment in client program using the <b>Palindrome</b> class                                                                             | Execution Output                                                           |
|----------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------|
| Palindrome p1 = new Palindrome("Racecar"); System.out.println(p1); Palindrome p2 = new Palindrome("Madam I'm Adam"); System.out.println(p2); | String: Racecar Palindrome: true  String: Madam I'm Adam Palindrome: false |

**Part (a).**

Redefine the `toString` method as discussed earlier. Method `toString` must return a string that will display two lines of output. The following table shows the result of several different calls made by `System.out.println(/* Palindrome Object */)` based on the argument of the constructor.

| Constructor argument | <code>toString</code> returns               |
|----------------------|---------------------------------------------|
| Racecar              | String: Racecar<br>Palindrome: true         |
| 12345BOB54321        | String: 12345BOB54321<br>Palindrome: true   |
| Madam I'm Adam       | String: Madam I'm Adam<br>Palindrome: false |

Complete method `toString` below.

```
// precondition: str is not null; palindrome is true or false;
// postcondition: Returns the value str and palindrome.
public String toString()
```

**Part (b).**

Write method `isPalindrome` as described earlier. In writing method `isPalindrome` you may call method `toUpperCase` specified below.

```
// precondition: some String object s is constructed
// postcondition: returns s with all lower-case characters converted to
// upper-case characters, if they exist
public String toUpperCase()
```

Complete method `isPalindrome` below.

```
// precondition: str is not null
// postcondition: returns true if str is a palindrome, and false otherwise
private boolean isPalindrome()
```

### Question 3.

Consider the following declarations of the `Product` class and the `Inventory` class. Objects of the `Product` class store information about the product in the fields: `item`, `price` and `quantity`. The `Inventory` class stores an array of `Product` objects.

The `Product` class declaration is complete. There is a constructor to create objects with provided information, there are three "get" methods to retrieve any of the stored information and there is a redefined `toString` method to display the object's contents.

```
class Product
{
 private String item;
 private double price;
 private int quantity;

 public Product(String i, double p, int q)
 { /* code to initialize private data */ }

 public String getItem() { return item; }
 public double getPrice() { return price; }
 public int getQuantity() { return quantity; }

 public String toString()
 { /* returns string to display private data */ }
}
```

The `Inventory` class declaration is incomplete. The constructor creates an `ArrayList` object, called `products`, which is initially empty. Method `addNewProduct` adds elements to the `products` array. Method `mustOrder` identifies those product items that are low in inventory and must be ordered.

```
class Inventory
{
 private ArrayList<Product> products;
 private int max;

 public Inventory()
 { /* code to initialize Inventory object */ }

 public void addNewProduct(String name, double cost, int amount)
 {
 /* to be implemented in part (a) */
 }

 public ArrayList<String> mustOrder()
 {
 /* to be implemented in part (b) */
 }

 public void displayProducts()
 { /* code to display all products */ }
}
```

Part (a).

Write method `addNewProduct`, which is described as follows. Method `addNewProduct` constructs a new `Product` object using the `item`, `price` and `quantity` parameters, which is then inserted into the `products` array, maintaining alphabetical order by `item` field. The code segment below is part of a client program that used the `Inventory` class and demonstrates the result of calling the `addNewProduct` method multiple times.

| Code segment in client program using the <code>Inventory</code> class | Execution Output                  |
|-----------------------------------------------------------------------|-----------------------------------|
| <code>Inventory hardware = new Inventory();</code>                    | <code>drill - 139.75 - 25</code>  |
| <code>hardware.addNewProduct("hammer",15.95,100);</code>              | <code>hammer - 15.95 - 100</code> |
| <code>hardware.addNewProduct("saw",24.50,15);</code>                  | <code>saw - 24.5 - 15</code>      |
| <code>hardware.addNewProduct("drill",139.75,25);</code>               |                                   |
| <code>hardware.addNewProduct("vise",87.65,10);</code>                 | <code>vise - 87.65 - 10</code>    |
| <code>hardware.displayProducts();</code>                              |                                   |

Complete method `addNewProduct` below.

```
// precondition: name is a correct product item.
// cost is the retail cost of the product item.
// amount is the quantity of new items added to inventory.
// postcondition: A new Product object is constructed with name, cost
// and amount.
// The new Product object is added to the products array
// such that products is ordered alphabetically by item.
public void addNewProduct (String name, double cost, int amount)
```

Part (b).

Write method `mustOrder`, which is described as follows. Method `mustOrder` returns an `ArrayList` object of product items that need to be ordered due to low inventory. Any inventory product that is below 20 in quantity needs to be ordered. If no product inventory is below 20 return an empty `ArrayList` object.

The code segment below is part of a client program that uses the `Inventory` class and demonstrates the result of calling the `mustOrder` method.

| Code segment in client program using the <code>Inventory</code> class                                                                                                                                                                                                                                    | Execution Output                                                                                   |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| <pre>Inventory hardware = new Inventory(); hardware.addNewProduct("hammer", 15.95, 100); hardware.addNewProduct("saw", 24.50, 15); hardware.addNewProduct("drill", 139.75, 25); hardware.addNewProduct("vise", 87.65, 10);  hardware.displayProducts();  System.out.println(hardware.mustOrder());</pre> | <pre>drill - 139.75 - 25 hammer - 15.95 - 100 saw - 24.5 - 15 vise - 87.65 - 10  [saw, vise]</pre> |

Complete method `mustOrder` below.

```
// precondition: products is a non-empty array of product objects
// postcondition: mustOrder returns an ArrayList of String objects whose
// quantity is less than 20 or if no items are below
// 20 level returns an empty ArrayList object.
public ArrayList<String> mustOrder()
```

#### Question 4.

In mathematics, there exists the concept of a set. Some programming languages, including Java, implement this concept. Although Java has several standard classes that implement the set concept, this question concerns developing your own `MySet` class using only the `ArrayList` class. It will mimic one or more of the standard classes.

A set is similar to the `ArrayList` class and static arrays. However, members of a set cannot be accessed with an index, order is not important, and only unique values are stored in a set, which means a set will not have duplicate items.

Consider the following incomplete declaration of the `MySet` class. An `ArrayList` object is used to store the `MySet` elements. Method `contains` determines if the argument object is a member of the `MySet` object. Method `add` adds new elements to the `MySet` object, which are not already stored in the object. Method `remove` removes an element from the `MySet` object, if it exists.

```
class MySet
{
 private ArrayList set;
 private int numItems;

 public MySet()
 {
 set = new ArrayList();
 numItems = 0;
 }

 public int size()
 {
 return numItems;
 }

 private int contains (Object item)
 {
 /* to be implemented in part (a) */
 }

 public void add(Object item)
 {
 /* to be implemented in part (b) */
 }

 public void remove (Object item)
 {
 /* to be implemented in part (c) */
 }

 // other methods to access and display MySet information
}
```

## Examples of MySet Operations

### **Example:**

This is a set of digits [4,7,2,9,3,5]

This is not a set [4,7,2,9,3,5,7] because 7 is stored twice in this data structure.

### **Example:**

MySet before add( 8 ) [4,7,2,9,3,5]  
numItems stores 6

MySet after add( 8 ) [4,7,2,9,3,8,5]  
numItems stores 7

MySet before add( 9 ) [4,7,2,9,3,5]  
numItems stores 6

MySet after add( 9 ) [4,7,2,9,3,5]  
numItems stores 6

### **Example:**

MySet before remove( 2 ) [4,7,2,9,3,5]  
numItems stores 6

MySet after remove( 2 ) [4,7,3,8,5]  
numItems stores 5

MySet before remove( 6 ) [4,7,2,9,3,5]  
numItems stores 6

MySet after remove( 6 ) [4,7,2,9,3,5]  
numItems stores 6

### **Example:**

MySet stores [4,7,2,9,3,5]      contains( 3 )      returns 4  
numItems stores 6

MySet stores [4,7,2,9,3,5]      contains( 8 )      returns -1  
numItems stores 6

**Part (a).**

Write method `contains`, which is described as follows. Method `contains` is a private helper method for other members of the `MySet` class. Method `contains` returns the index value if the parameter item exists in the current `MySet` object and returns -1 otherwise. Logically, `MySet` will behave like a set data structure, however since this is a private method the indexing will not be seen at the client level. The implementation may only use the operations available for the `ArrayList` class.

Complete method `contains` below.

```
// precondition: item is not null.
// postcondition: Returns index location in the set array if item exists,
// and returns -1 otherwise.
private int contains (Object item)
```

**Part (b).**

Write method `add`, which is described as follows. Method `add` checks to see if the parameter element is already present in the `MySet` object. The element is added to the `MySet` object if it is not yet present and `numItems` is incremented. In writing `add` you may call method `contains` specified in part (a). Assume that `contains` works as specified, regardless of what you wrote in part (a). Solutions that reimplement functionality by this method, rather than invoking this method, will not receive full credit.

Complete method `add` below.

```
// precondition: item is not null.
// postcondition: item is added to the set array, if item is not
// contained in set and numItems is incremented.
public void add (Object item)
```

Part (c).

Write method `remove`, which is described as follows. Method `remove` checks to see if the parameter element is present in the `MySet` object. The element is removed from the `MySet` object if it is present and `numItems` is decremented. In writing `remove` you may call method `contains` specified in part (a). Assume that `contains` works as specified, regardless of what you wrote in part (a). Solutions that reimplement functionality by this method, rather than invoking this method, will not receive full credit.

Complete method `remove` below.

```
// precondition: item is not null
// postcondition: item is removed from the set array, if item is
// contained in set and numItems is decremented
public void remove (Object item)
```

# Chapter 17 Questions

## Sample AP Examination II

---

### Section I

**Time** - 1 hour and 15 minutes

**Number of questions** - 40

**Percent of total grade** - 50

01. Consider the following method

```
public static int japan(int p, int q)
{
 int n = 0;
 while (n <= p)
 {
 n++;
 while (p < q)
 {
 p++;
 n++;
 }
 }
 return n;
}
```

What value is returned as a result of the call `japan(15, 7)`?

- (A) 10
- (B) 13
- (C) 15
- (D) 19
- (E) 23

02. Consider the following two methods.

```
// precondition: p and q are positive integers
public static boolean swing(int p, int q)
{
 return (p % q) == (p - (q * p/q));
}

// precondition: p and q are positive integers
public static boolean polka(int p, int q)
{
 return (p % q) < (p - q);
}
```

Assume that the preconditions of samba and mambo are satisfied.

For which values of p and q is it true that `swing(p, q) == polka(p, q)` ?

- (A) For all values of p and q
- (B) For no values of p and q
- (C) For all values of p and q, such that  $p < q$  is false
- (D) For all values of p and q, such that  $p < q$  is true
- (E) For all values of p and q, such that  $p \% q$  is 0

03. Consider the following method.

```
public static void austria(ArrayList<Integer> list)
{
 Iterator<Integer> iter = list.iterator();
 while (iter.hasNext())
 {
 Integer temp = iter.next();
 if (temp.intValue() % 2 != 0)
 iter.remove();
 }
}
```

Which of the following describes the result of the call `austria(list)` ?

- (A) All odd integer values are deleted from the list array.
- (B) All even integer values are replaced with null.
- (C) All odd integer values are replaced with null.
- (D) All even integer values are deleted from the list array.
- (E) list is an empty array.

04. Consider the following code segment.

```
String s1 = "Annapolis";
String s2 = "WestPoint";
String temp = s1 + s2;
System.out.println(temp.substring(s1.length()));
```

What is printed as a result of executing the code segment?

- (A) AnnapolisWestPoint
- (B) Annapolis
- (C) WestPoint
- (D) AnnapolisW
- (E) sWestPoint

05. Consider the following incomplete getMean method.

```
// Precondition: numbers is non-empty array of double numbers.
// Postcondition: getMean returns the mean of the numbers array elements.
public static double getMean(double[] numbers)
{
 /* missing code */
}
```

Which of the following implementation of /\* **missing code** \*/ will satisfy the postcondition of method getMean?

**Implementation 1**

```
double sum = 0;
for (double number : numbers)
 sum += number;
return sum/numbers.length;
```

**Implementation 2**

```
double sum = 0;
for (int k = 0; k < numbers.length; k++)
 sum += numbers[k];
return sum/numbers.length;
```

**Implementation 3**

```
double sum = 0;
for (int k = 0; k <= numbers.length; k++)
 sum += numbers[k];
return sum/numbers.length;
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementation 1 & 2 only
- (E) Implementation 1 & 3 only

06. Consider the following method.

```
public static int columbia(int p, int q)
{
 if (p % q == 0)
 return q;
 else
 return columbia(p-1,q);
}
```

What value is returned as a result of the call `columbia(45,30)`?

- (A) 30
- (B) 15
- (C) 45
- (D) 5
- (E) 1

07. Consider the following two methods.

```
public static int egypt(int p, int q)
{
 if (p % q == 0)
 return q;
 else
 return egypt(q,p % q);
}

public static int congo(int p, int q)
{
 while (p % q != 0)
 {
 int temp = p % q;
 if (temp != 0)
 {
 p = q;
 q = temp;
 }
 }
 return q;
}
```

For which values of `p` and `q` do `egypt(p,q)` and `congo(p,q)` return the same result?

- (A) For all values of `p` and `q`
- (B) For all values, such that `p > q`
- (C) For all values, such that `p <= q`
- (D) For all values, such that `p > 0` and `q > 0`
- (E) For no values of `p` and `q`

08. Consider the following class.

```
class Plips
{
 private int numPlips;

 public Plips(int p)
 {
 numPlips = p;
 }
}
```

Assume that the following code segment appears in a client program.

```
Plips plip1 = new Plips(1000);
Plips plip2 = new Plips(2000);
System.out.println(plip1);
System.out.println(plip2);
```

What is printed as a result of executing the code segment?

- (A) 1000  
2000
- (B) 2000  
1000
- (C) Error message, because there are no methods that accesses the private data in the `Plips` class.
- (D) `Plips@` followed by a hexadecimal number  
`Plips@` followed by the same hexadecimal number
- (E) `Plips@` followed by a hexadecimal number  
`Plips@` followed by a different hexadecimal number

09. Consider the following three classes

```
class Animal
{
 private String type;

 /* constructor */
}

class Mammal extends Animal
{
 /* constructor */
}

class Cat extends Mammal
{
 /* constructor */
}
```

What type of relationship(s) is/are established by these three classes?

- (A) One example of inheritance only
- (B) One example of composition only
- (C) Two examples of inheritance
- (D) Two examples of composition
- (E) One example of inheritance and one example of composition

10. Consider the following code segment.

```
System.out.println(8 + 8 / 4 + 3 * 6);
```

What is printed as a result of executing the code segment?

- (A) 42
- (B) 22
- (C) 28
- (D) 18
- (E) 54

11. Consider the following two methods.

```
public static boolean method1(int n)
{
 System.out.println("Calling method 1");
 return n < 5;
}

public static boolean method2(int n)
{
 System.out.println("Calling method 2");
 return n < 5;
}
```

Assume that the following code segment appears in a client program.

```
for (int n = 4; n < 6; n++)
{
 if (method1(n) || method2(n))
 System.out.println();
}
```

What is printed as a result of executing the code segment?

- (A) Calling method1

Calling method1  
Calling method2

- (B) Calling method1  
Calling method2

- (C) Calling method1  
Calling method1

- (D) Calling method2  
Calling method2

- (E) Calling method1  
Calling method2

Calling method1

12. Consider the following code segment.

```
boolean trueFalse = <true or false>
trueFalse = (trueFalse == true && !trueFalse == false);
System.out.println(trueFalse);
```

What is printed as a result of executing the code segment?

- (A) The code segment always prints true.
- (B) The code segment always prints false.
- (C) The code segment prints true, if the initial value of trueFalse is true and false otherwise.
- (D) The code segment prints true, if the initial value of trueFalse is false and true otherwise.
- (E) The code segment will print true or false, but the actual value cannot be determined.

13. Consider the following method.

```
public static ArrayList<Integer> russia(int[] list1)
{
 ArrayList<Integer> list2 = new ArrayList<Integer>();
 for (int p = 0; p < list1.length; p++)
 {
 int temp = list1[p];
 for (int q = p; q < list1.length; q++)
 {
 if (list1[q] < temp)
 temp = list1[q];
 }
 list2.add(new Integer(temp));
 }
 return list2;
}
```

Assume that the following code segment appears in a client program.

```
int[] intArray = {66,22,11,44,55,33,77};
System.out.println(russia(intArray));
```

What is printed as a result of executing the code segment?

- (A) [11, 22, 33, 44, 55, 66, 77]
- (B) [77, 66, 55, 44, 33, 22, 11]
- (C) [11, 66, 22, 44, 55, 33, 77]
- (D) [66, 22, 44, 55, 33, 77, 11]
- (E) [11, 11, 11, 33, 33, 33, 77]

14. Does the linear search have an advantage over the binary search?
- (A) No, the binary search is far more time efficient than the linear search, especially when the search list is very large.
  - (B) Not really, it is possible that a linear search finds an element quicker than the binary search, but that is only in rare cases.
  - (C) Yes, with any search list that is not sorted, the binary search finds an element much slower than the linear search.
  - (D) Yes, with any search list that is not sorted the binary search may never find an existing element.
  - (E) It strictly depends on the location of the search element. Sometimes the binary search is faster, and at other times the linear search is faster.

15. Consider the Java program statement

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

Which of the following features is displayed by this statement?

- I. Inheritance
- II. Composition
- III. Short Circuiting

- (A) I only
- (B) II only
- (C) III only
- (D) I and II only
- (E) II and III only

16. A search item exists in a sorted array of 5000 elements.

How many comparisons, in a worst-case scenario, will be made by a linear search method?

- (A) 12
- (B) 13
- (C) 2500
- (D) 4999
- (E) 5000

17. Consider the following two `sortList` methods.

```
public static void sortList (int[] list)
{
 boolean sorted = false;
 int p = 1;
 while (!sorted)
 {
 sorted = true;
 for (int q = 0; q < list.length-p; q++)
 if (list[q] < list[q+1])
 {
 sorted = false;
 int temp = list[q];
 list[q] = list[q+1];
 list[q+1] = temp;
 }
 }
}

public static void sortList (double[] list)
{
 for (int p = 1; p < list.length; p++)
 for (int q = 0; q < list.length-p; q++)
 if (list[q] < list[q+1])
 {
 double temp = list[q];
 list[q] = list[q+1];
 list[q+1] = temp;
 }
}
```

Can a class work correctly with both of these `sortList` methods?

- (A) Yes, because the two `sortList` methods have different signatures.
- (B) Yes, because the two `sortList` methods have different source code.
- (C) No, because overloaded methods may not use the same number of parameters.
- (D) No, because overloaded methods may not have identical parameter lists.
- (E) No, because overloaded methods may only use `Object` data types.

18. Consider the following two Boolean expressions.

`p && (a || b)`

`p && a || p && b`

For which values of `a` and `b` are the two expressions equivalent?

- (A) For no values of `a` and `b`
- (B) Only whenever `a` is true
- (C) Only whenever `b` is true
- (D) Only whenever both `a` is true and `b` is true.
- (E) For all values of `a` and `b`

19. Consider the following code segment.

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
for (int k = 1; k <= 10; k++)
{
 int rndInt = (int) (Math.random() * 9000) + 1000;
 numbers.add(new Integer(rndInt));
}
for (Integer number : numbers)
 System.out.println(number);
```

What is printed as a result of executing the code segment?

- (A) Ten random integers `x`, such that `1000 <= x <= 9999`
- (B) Ten random integers `x`, such that `1000 <= x <= 10000`
- (C) Ten random real numbers `x`, such that `1000 < x < 10000`
- (D) Ten random real numbers `x`, such that `1000 <= x < 10000`
- (E) Ten random real numbers `x`, such that `1000 <= x < 9999`

20. A search item exists in a sorted array of 5000 elements.

How many comparisons, in a worst-case scenario, will be made by a binary search method?

- (A) 10
- (B) 13
- (C) 2500
- (D) 4999
- (E) 5000

21. Consider the following method.

```
// precondition: m > 0
public static int mystery(int m)
{
 int counter = 0;
 for (int p = 1; p < m; p++)
 for (int q = 0; q < m; q++)
 counter++;
 return counter;
}
```

What value is returned as a result of the call `mystery(x)` ?

- (A)  $(x - 1)^2$
- (B)  $x^2$
- (C)  $x^2 - 1$
- (D)  $x^2 - x$
- (E)  $(x - 2)^2$

22. The boolean expression

`(A || B) && (A && B)`

is equivalent to

- (A) `!(A || B) && !(A && B)`
- (B) `!((A || B) || (A && B))`
- (C) `(A || B) || (A && B)`
- (D) `!( !(A || B) || !(A && B) )`
- (E) `!(A && B) || !(A || B)`

23. Consider the following method.

```
public static void mystery(int[] list)
{
 int n = list.length;
 for (int k = 0; k < n/2; k++)
 {
 int temp = list[n-k-1];
 list[n-k-1] = list[k];
 list[k] = temp;
 }
}
```

Assume that integer array contains the following values: {11, 22, 33, 44, 55, 66, 77, 88, 99}  
Which of the following represents the contents of list as a result of calling method `mystery`?

- (A) {88, 99, 66, 77, 55, 33, 44, 11, 22}
- (B) {22, 11, 44, 33, 55, 77, 66, 99, 88}
- (C) {99, 88, 77, 66, 55, 44, 33, 22, 11}
- (D) {11, 22, 33, 44, 55, 66, 77, 88, 99}
- (E) {66, 77, 88, 99, 55, 44, 33, 22, 11}

Questions 24-25 refer to the following Student class.

```
class Student
{
 private String name;
 private double gpa;

 public Student (String n, double g)
 {
 name = n;
 gpa = g;
 }

 /* missing equals method */

 /* missing toString method */
}
```

24. Method **equals** must be implemented, such that equality of **Student** objects requires identical names and identical GPAs.

Which of the following three implementations of /\* missing code \*/ will make method **equals** work as intended?

**Implementation 1**

```
public boolean equals(Object source)
{
 Student temp = (Student) source;
 return name.equals(temp.name) && gpa == temp.gpa;
}
```

**Implementation 2**

```
public boolean equals(Object source)
{
 return name.equals(source.name) && gpa == source.gpa;
}
```

**Implementation 3**

```
public int equals (Object source)
{
 Student temp = (Student) source;
 if (name.equals(temp.name) && gpa == temp.gpa)
 return 0;
 else
 return -1;
}
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementations 1 and 2 only
- (E) Implementations 2 and 3 only

25. Assume that the following code segment appears in a client program.  
The code segment is followed by the output that is the result of executing the code segment.

```
Student student1 = new Student("Meg", 3.675);
Student student2 = new Student("Tom", 3.285);
System.out.println(student1);
System.out.println(student2);
```

Name: Meg  
GPA: 3.675

Name: Tom  
GPA: 3.285

Which of the following three implementations of /\* missing code \*/ in `toString` will make a statement like `System.out.println(manageChecksObject)` display output like shown above?

**Implementation 1**

```
public String toString()
{
 return "Name: " + name + "\n" + "GPA: " + gpa;
}
```

**Implementation 2**

```
public boolean toString()
{
 return "Name: " + name + "\n" + "GPA: " + gpa + "\n";
}
```

**Implementation 3**

```
public String toString()
{
 return "Name: " + name + "\n" + "GPA: " + gpa + "\n";
}
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementations 1 and 2 only
- (E) Implementations 2 and 3 only

26. Consider the following method.

```
public static String bizarre (String str)
{
 String temp = "";
 for (int k = 0 ; k < str.length(); k++)
 {
 if (k % 2 != 0)
 temp = temp + " " + str.substring(k);
 }
 return temp;
}
```

What value is returned as a result of the call `bizarre( "HOLLAND" )` ?

- (A) OLLAND LAND ND
- (B) HOLLAND LLAND AND D
- (C) OLLAND LAND ND D
- (D) HOLLAND OLLAND LLAND LAND AND ND D
- (E) HOLLAND

27. Which of the following code statements uses one of the `Math` class methods incorrectly?

- (A) `System.out.println(Math.sqrt(Math.abs(-45)));`
- (B) `System.out.println(Math.abs(Math.pow(-2,3)));`
- (C) `System.out.println(Math.sqrt(Math.pow(13,-2)));`
- (D) `System.out.println(Math.pow(Math.sqrt(25),-3));`
- (E) `System.out.println(Math.random(100));`

28. Consider the following airport organizational design.

Passengers flying from Houston to Berlin enter a terminal with an entrance labeled **Houston Passengers Only**. They enter a corridor without intersections. At the check-in counter, there are agents who only handle passengers flying to Berlin. Luggage is placed on a conveyor belt that only goes to a plane flying to Berlin. Passengers continue to the end of the corridor where a single gate has a plane, which is ready to fly to Berlin. The airport has multiple additional corridors, which are all dedicated for passengers flying to one specified destination.

The stated airport design is an example of using

- (A) pre-OOP program design.
- (B) encapsulation.
- (C) inheritance.
- (D) polymorphism.
- (E) debugging techniques.

29. Consider the following two methods.

```
public static int botswana(int x, int y)
{
 int temp = ghana(x,y);
 return x / temp * y;
}

public static int ghana(int a, int b)
{
 if (a % b == 0)
 return b;
 else
 return ghana(b, a % b);
}
```

What value is returned as a result of the call `botswana(120,108)` ?

- (A) 2160
- (B) 1080
- (C) 120
- (D) 108
- (E) 24

30. Consider the following code segment.

```
ArrayList<String> list1 = new ArrayList<String>();
list1.add("Tom");
list1.add("Sue");
list1.add("Bob");
list1.add("Ann");
ArrayList<String> list2 = list1;
list2.add("Ann");
list2.add("Bob");
list2.add("Sue");
list2.add("Tom");
System.out.println(list1);
```

What is printed as a result of executing the segment?

- (A) [Ann, Bob, Sue, Tom]
- (B) [Tom, Sue, Bob, Ann]
- (C) [Tom, Sue, Bob, Ann, Ann, Bob, Sue, Tom]
- (D) [Ann, Bob, Sue, Tom, Tom, Sue, Bob, Ann]
- (E) [Ann, Bob, Sue, Tom, Ann, Bob, Sue, Tom]

31. Consider the following three classes

```
class Education
{
 private int years;
 private String degree;
 public Education(int y, String d)
 {
 years = y;
 degree = d;
 System.out.println("Education is constructed");
 }
}

class Person
{
 private String name;
 private int age;
 private Education education;
 public Person(String n, int a, int y, String d)
 {
 System.out.println("Person is constructed");
 name = n;
 age = a;
 education = new Education(y,d);
 }
}

class Employee extends Person
{
 private String jobTitle;
 private double salary;
 public Employee(String j, double s, String n, int a, int y, String d)
 {
 super(n,a,y,d);
 jobTitle = j;
 salary = s;
 System.out.println("Employee is constructed");
 }
}
```

Assume that the following code segment appears in a client program. What is printed by the code segment?

```
Employee juan = new Employee("Plumber",45000,"Juan",29,14,"Associates");
```

- (A) Person is constructed  
Education is constructed  
Employee is constructed
- (B) Education is constructed  
Person is constructed  
Employee is constructed
- (C) Employee is constructed  
Person is constructed  
Education is constructed
- (D) Employee is constructed
- (E) Employee is constructed  
Person is constructed

32. Consider the following method.

```
// Precondition: matrix is a "square" matrix
// All elements of matrix are non-zero.
// 0 <= r < matrix.length
// r equals c
public static void reArrange(int matrix[][], int r, int c)
{
 for (int row = 0; row < matrix.length; row++)
 for (int col = 0; col < matrix[0].length; col++)
 if (row == c || col == r)
 matrix[row][col] = 0;
}
```

Which of the following describes the values stored in `matrix` after `reArrange` is called?

- (A) All the values in `matrix` are zero.
- (B) All the values in `row` `r` and also in `col` `c` are zero.
- (C) All the values in `row` `c` and also in `col` `r` are zero.
- (D) Only the value of `matrix[r][c]` is zero.
- (E) All the values in `matrix` are unchanged.

33. Consider the following code segment.

```
for (int k = 1; k <= 10; k++)
{
 int rndInt = (int) (Math.random() * 10;
 System.out.print(rndInt + " ");
}
System.out.println();
```

What is printed as a result of executing the code segment?

- (A) A set of 10 zeros for each separate execution.
- (B) A different set of random integers in the [0..10] range for each separate execution.
- (C) A different set of random integers in the [0..9] range for each separate execution.
- (D) An identical set of random integers in the [0..10] range for each separate execution.
- (E) An identical set of random integers in the [0..9] range for each separate execution.

34. Consider the following interface declaration and abstract class declaration.

```
interface Paris
{
 public Object getData();
 public void setData(Object obj);
 public void showData();
}

abstract class London
{
 abstract public Object getData();
 abstract public void setData(Object obj);
 abstract public void showData();
}
```

Both the `Paris` interface and the `London` abstract class contain the same methods.  
Are there any differences, practically speaking, besides the reserved words `interface` and `class`?

- I. The `Paris` methods are not abstract. The `London` methods are abstract.
  - II. A class implementing `Paris` can also implement another interface, but a class extending `London` cannot extend another class.
  - III. A class extending `London` can also extend another class, but an interface implementing `Paris` cannot implement another interface.
- 
- (A) I only
  - (B) II only
  - (C) III only
  - (D) I and II only
  - (E) I and III only

35. Consider the following class.

```
class Names
{
 private ArrayList<String> nameList;

 public Names (String[] list)
 {
 nameList = new ArrayList<String>();
 for (String name : list)
 nameList.add(name);
 }

 public String toString()
 {
 String temp = "";
 for (String name : nameList)
 temp = temp + name + "\n";
 return temp;
 }
}
```

Assume that the following code segment appears in a client program.  
What is printed by the code segment?

```
String[] x = { "AAA" , "BBB" , "CCC" , "DDD" , "EEE" };
Names list = new Names(x);
System.out.println(list);
```

- (A) null
- (B) [ AAA, BBB, CCC, DDD, EEE ]
- (C) AAA + BBB + CCC + DDD + EEE
- (D) AAA  
    BBB  
    CCC  
    DDD  
    EEE
- (E) [ "AAA" , "BBB" , "CCC" , "DDD" , "EEE" ]

36. Consider the following code statement.

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

Identifier out is

- (A) a reserved word.
- (B) an instance variable.
- (C) a class.
- (D) a class method.
- (E) an object method.

37. Consider the following method.

```
// Precondition: matrix is a "square" matrix
public static void reArrange(int matrix[][])
{
 for (int row = 0; row < matrix.length; row++)
 for (int col = 0; col < matrix.length; col++)
 matrix[row][col] = matrix[col][row];
}
```

Which of the following describes the values stored in matrix after reArrange is called?

- (A) All the values of the matrix rows are reversed.
- (B) All the values of the matrix columns are reversed.
- (C) All the values in the matrix diagonals are reversed.
- (D) All the values in the nth row and the nth column of matrix are the same.
- (E) All the values in matrix are unchanged.

38. Consider the following method.

```
public static void ifElse(int n)
{
 if (n == 4)
 {
 ifElse(n - 1);
 System.out.print(n + " ");
 }
 else if (n == 3)
 {
 ifElse(n - 1);
 System.out.print(n + " ");
 }
 else if (n == 2)
 {
 ifElse(n - 1);
 System.out.print(n + " ");
 }
 else
 System.out.print(n + " ");
}
```

Which of the following is printed as a result of the call `ifElse(4)` ?

- (A) 4 3 2 1 2 3 4
- (B) 1 2 3 4 3 2 1
- (C) 1 2 3 4
- (D) 4 3 2 1
- (E) Continues output due to infinite recursion

39. Consider the following classes.

```
class Plant
{
 private int quantity;

 public Plant (int q)
 {
 quantity = q;
 }
 public int getQuantity()
 {
 return quantity;
 }
}

class Flower extends Plant
{
 private String name;

 public Flower(String n, int q)
 {
 super(q);
 name = n;
 }

 public String toString()
 {
 return getQuantity() + " " + name;
 }
}
```

Assume that the following code segment appears in a client program.  
What is printed by the code segment?

```
Flower dutch = new Flower("Tulips",5000);
System.out.println(dutch);
```

- (A) 5000 Tulips
- (B) 0 Tulips
- (C) 5000 dutch Tulips
- (D) null Tulips
- (E) null dutch Tulips

40. Consider the following method

```
// precondition: n > 0 and a number that can be expressed
// as powers of two, such as 1, 2, 4, 8, 16, etc.
public static int counter(int n)
{
 int p = 0;
 int q = 2;
 while (q <= n)
 {
 q *= 2;
 p++;
 }
 return p;
}
```

What value is returned as a result of the call `counter(n)` ?

- (A) Some value  $v$ , such that  $n/v == 2$
- (B)  $(n - 1)^2$
- (C)  $n^2/2$
- (D)  $2^n$
- (E) Some value  $v$ , such that  $2^v == n$

# Chapter 17 Free Response Questions

## Sample AP Examination II

---

### Section II

**Time** - 1 hour and 45 minutes

**Number of questions** - 4

**Percent of total grade** - 50

#### Question 1.

There are three measures of central tendency in statistics. One of the measures of central tendency is the arithmetic mean, which is calculated by adding all of the numbers and dividing by the quantity of the numbers. Another measure of central tendency is the mode, which is the value in a list with the greatest frequency.

**Note:** Initially, this problem may appear to be a repeat of a free response question of the previous sample exam. The questions are similar, but presented intentionally to test the ability to use both the `ArrayList` class and static arrays.

#### Part (a).

Write method `getMean`, which is described as follows. Method `getMean` returns the average, called the **mean**, of a set of numbers. The **mean** is computed by adding the numbers and dividing the sum by the quantity of the numbers.

For example:

The **mean** of {10, 18, 17} is **15**

The **mean** of {10, 15} is **12.5**

Complete method `getMean` below.

```
// precondition: list is a non-empty integer array
// postcondition: returns the mean of the integer values stored
// by the list array.
public static double getMean(int[] list)
```

**Part (b).**

Write method `getMode`, which is described as follows. Method `getMode` returns a measure, called the mode, of a set of numbers, which is the number with the greatest frequency. For this question the mode will be the first number with the greatest frequency in an ordered list.

For example:

The **mode** of {10, 10, 13, 17, 17, 17, 18, 21} is **17**.

The **mode** of {54, 54, 54, 60, 60, 70, 85, 85} is **54**.

The **mode** of {20, 20, 20, 30, 30, 30, 40, 40, 40} is **20**.

The **mode** of {50} is **50**

In writing method `getMode` you may call method `sortList` specified below.

```
// precondition: list is a non-empty integer array
// postcondition: The integers in list are sorted in ascending order
public static void sortList (int[] list)
```

Complete method `getMode` below.

```
// precondition: list is a non-empty integer array
// The integers in list are randomly ordered
// postcondition: returns the mode value of list
public static int getMode(int[] list)
```

## Question 2.

Consider the following incomplete declaration of a Random1 class. Class Random1 could be an implementation of the standard Java class Random.

```
class Random1
{
 double currentNr;

 public Random1()
 {
 double seed = System.currentTimeMillis();
 currentNr = seed;
 }

 public Random1(double seed)
 {
 currentNr = seed;
 }

 // postcondition: returns a double x, such that 0 <= x < 1
 public double getRandom()
 {
 double nextNr = currentNr + Math.PI;
 nextNr = nextNr * nextNr;
 int temp = (int) nextNr;
 currentNr = nextNr = nextNr - temp;
 return nextNr;
 }

 public int nextInteger(int n)
 {
 /* be implemented in part (a) */
 }
}
```

### Part (a).

Complete method nextInteger below.

```
// precondition: n > 1
// postcondition: returns an int x, such that 0 <= x < n
public int nextInteger(int n)
```

**Part (b).**

Consider the following incomplete class Random2, which is a sub class of Random1.

```
class Random2 extends Random1
{
 public Random2(double seed)
 {
 /* be implemented in part (b) */
 }

 public int nextInteger(int start, int end)
 {
 /* be implemented in part (c) */
 }
}
```

Write constructor Random2, which calls and provides the seed information for constructor Random1.

Complete method Random2 below.

```
// precondition: seed > 0
// postcondition: seed is passed to the super class constructor Random1.
public Random2(double seed)
```

**Part (c).**

Write overloaded method `nextInteger`, which returns an integer in a specified range.

Complete method `nextInteger` below.

```
// precondition: start is an integer
// end is an integer such that end > start
// postcondition: returns a random integer x such that for all x,
// start <= x <= end
public int nextInt(int start, int end)
```

### Question 3.

Encrypting data is an important tool in computer science. A simplistic approach in encoding a message is to shift characters by a number of spaces with wraparound. If the shift is four characters forward then a word, like **SATURDAY**, becomes **WEXYVHEC**. The coded message looks unreadable, but a consistent pattern of shifting characters is quickly broken by programs designed to decode messages.

A better approach is to shift characters in a manner that is not consistent by using a special key. Each character in the original message is still shifted to another character, but the shifting is determined by the character shift-value of a key. The actual shift value is not important. This value can be the Unicode value of the character or it can be some value determined by a special key-value-code. The point is that the original message is now altered in an irregular pattern. Consider the following example with intentionally, small shift values in the key **MONDAY**.

| Key Characters | M | O | N | D | A | Y |
|----------------|---|---|---|---|---|---|
| Shift Value    | 9 | 2 | 0 | 1 | 7 | 5 |

Now imagine that this key is applied to the source message **DISCOMBOBULATED**. The coding process goes through the following steps:

1. Get Unicode value of each character in the source message.  
(Unicode values and ASC values are the same for alpha-numeric characters)
2. Get the shift values for each character in the key.
3. Add the shift value of the key to each corresponding character in the message to get the coded value.
4. The coded message is the set of characters that is associated with the coded values.

| Message       | D        | I        | S        | C        | O        | M        | B        | O        | B        | U        | L        | A        | T        | E        | D        |
|---------------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| ASC value     | 68       | 73       | 83       | 67       | 79       | 77       | 66       | 79       | 66       | 85       | 76       | 65       | 84       | 69       | 68       |
| Key           | M        | O        | N        | D        | A        | Y        | M        | O        | N        | D        | A        | Y        | M        | O        | N        |
| Shift value   | 9        | 2        | 0        | 1        | 7        | 5        | 9        | 2        | 0        | 1        | 7        | 5        | 9        | 2        | 0        |
| Coded value   | 77       | 75       | 83       | 68       | 86       | 82       | 75       | 81       | 66       | 86       | 83       | 70       | 93       | 71       | 68       |
| Coded message | <b>M</b> | <b>K</b> | <b>S</b> | <b>D</b> | <b>V</b> | <b>R</b> | <b>K</b> | <b>Q</b> | <b>B</b> | <b>V</b> | <b>S</b> | <b>F</b> | <b>J</b> | <b>G</b> | <b>D</b> |

Consider the following incomplete declaration of the `Cryptic` class. The constructor initializes the `keyCode` field, which determines shift values. Private method `getCode` converts a single character of a source message into a coded character. Private method `getSource` converts a single character of a coded message into a character that is part of the source message. Method `enCrypt` converts a message into a coded message and method `deCrypt` converts a coded message back into a readable source message.

```
class Cryptic
{
 private String keyCode;

 public Cryptic ()
 {
 /* initializes the keyCode */
 }

 private String getCode(String sourceChar, String keyChar)
 {
 /* code to convert source message character into coded character */
 }

 private String getSource(String codedChar, String keyChar)
 {
 /* code to convert coded character into source message character */
 }

 public String enCrypt(String source, String key)
 {
 /* to be implemented in part (a) */
 }

 public String deCrypt(String coded, String key)
 {
 /* to be implemented in part (b) */
 }
}
```

The code segment below is part of a client program that constructs a `Cryptic` object, converts the source message `DISCOMBOBULATED` into a coded message, and then converts the coded message back into the original message.

| Code segment in client program using the <code>Cryptic</code> class                                                                                                                                                                                                                                                                                      | Execution Output                                                                   |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| <pre>Cryptic secret = new Cryptic(); String source = "DISCOMBOBULATED"; String key = "MONDAY"; System.out.println("source: " + source); System.out.println(); String coded = secret.enCrypt(source,key); System.out.println("coded: " + coded); System.out.println(); source = secret.deCrypt(coded,key); System.out.println("source: " + source);</pre> | <pre>source: DISCOMBOBULATED coded: MKSDVRKQBVSF ]GD source: DISCOMBOBULATED</pre> |

**Part (a).**

Write method enCrypt as described earlier. In writing method enCrypt you may call method getCode specified below.

```
// precondition: sourceChar is not null and is a one-character String
// member of some message to be coded.
// keyChar is not null and is a one-character String
// member of a coding key used to alter sourceChar
// postcondition: returns a one-character String, which is the encrypted
// character of sourceChar
private String getCode(String sourceChar, String keyChar)
```

Complete method enCrypt below.

```
// precondition: source is not null and represents the source message
// to be encrypted
// key is not null and represents the key string that is
// used to encrypt source
// postcondition: returns the encrypted version of source
public String enCrypt(String source, String key)
```

**Part (b).**

Write method deCrypt as described earlier. In writing method deCrypt you may call method getSource specified below.

```
// precondition: codedChar is not null and is a one-character String
// member of an encrypted message to be decoded.
// keyChar is not null and is a one-character String
// member of a coding key used to alter codedChar
// postcondition: returns a one-character String, which is the decrypted
// character of codedChar
private String getSource(String codedChar, String keyChar)
```

Complete method deCrypt below.

```
// precondition: coded is not null and represents the encrypted message
// to be decoded
// key is not null and represents the key string that is
// used to decrypt coded
// postcondition: returns the decrypted version of coded
public String deCrypt(String coded, String key)
```

#### Question 4.

Consider the following incomplete declaration of the Rational class. The purpose of the Rational class is to facilitate arithmetic operations between rational numbers. Operations between fractional numbers are handled by Java in decimal format, like  $0.25 + 0.375 = 0.625$ . The rational class will be able to add and display the same operations in the form of common fractions, like  $\frac{1}{4} + \frac{3}{8} = \frac{5}{8}$ .

The Rational class has a constructor for rational numbers that are 0, rational numbers that are integers like 5, and rational numbers in the format num/den, where den  $\neq 0$ . The `toString` method needs to be redefined so that Rational objects can be displayed as a common fraction in lowest terms. A Rational class would have all the customary arithmetic operations. For this question only the `add` method needs to be implemented.

```
class Rational
{
 private int num;
 private int den;

 public Rational () { num = 0; den = 1; }
 public Rational (int nr) { num = nr; den = 1; }
 public Rational (int n, int d) { num = n; den = d; }

 // precondition: num and den are initialized; den != 0
 // postcondition: returns 0 if num equals 0
 // returns num if den equals 1
 // returns value-of-num/value-of-den if both
 // num and den are non-zero
 public String toString()
 {
 /* to be implemented in part (a) */
 }

 // precondition: n and d are integers greater than 0
 // postcondition: returns the greatest common factor of n and d
 private int getGCF (int n, int d)
 {
 /* to be implemented in part (b) */
 }

 // precondition: r1 and r2 are Rational objects
 // postcondition: r1 is added to r2 according to the common rules
 // of arithmetic for rational numbers and the final
 // object is in lowest terms.
 // The resulting num and den values are stored by the
 // current object
 public void add(Rational r1, Rational r2)
 {
 /* to be implemented in part (c) */
 }

 // additional arithmetic methods
}
```

Part (a).

Write method `toString`, which is described as follows. Method `toString` is redefined to display `Rational` objects in the form of common fractions, `0` is displayed as `0`, integers are displayed with numbers only, like `5` and fractions are displayed like `7/6`. The table below demonstrates the result of redefining the `toString` method for the `Rational` class.

| Code segment in client program using the <code>Rational</code> class | Execution Output  |
|----------------------------------------------------------------------|-------------------|
| <code>Rational nr1 = new Rational();</code>                          | 0                 |
| <code>Rational nr2 = new Rational(5);</code>                         | 5                 |
| <code>Rational nr3 = new Rational(3,4);</code>                       | <code>3/4</code>  |
| <code>Rational nr4 = new Rational(5,10);</code>                      | <code>5/10</code> |
| <code>System.out.println(nr1);</code>                                | <code>5/4</code>  |
| <code>System.out.println(nr2);</code>                                |                   |
| <code>System.out.println(nr3);</code>                                |                   |
| <code>System.out.println(nr4);</code>                                |                   |
| <code>nr1.add(nr3,nr4);</code>                                       |                   |
| <code>System.out.println(nr1);</code>                                |                   |

Complete method `toString` below.

```
// precondition: num and den are initialized; den != 0
// postcondition: returns 0 if num equals 0
// returns num if den equals 1
// returns value-of-num/value-of-den if both
// num and den are non-zero
public String toString()
```

Part (b).

Write method `getGCF` as described earlier. There exist many implementations for computing the Greatest Common Factor. *Euclid's Algorithm* is a 2000-year-old GCF algorithm that is suited very well for the sequential instructions required by a computer program. In the example below, Euclid's Algorithm is used to compute the GCF of **120** and **108**. The result is **12**.

| Algorithm Steps                                                                                                | Sample Problem                                          |
|----------------------------------------------------------------------------------------------------------------|---------------------------------------------------------|
| <b>Step 1:</b><br>Start with two integers                                                                      | integer1 is 120<br>integer2 is 108                      |
| <b>Step 2:</b><br>Divide integer1 by integer2 and compute the remainder.                                       | $120 / 108 = 1$<br>The remainder = 12                   |
| <b>Step 3:</b><br>If the remainder equals 0, you are finished. The GCF is integer2.                            | The remainder is not 0<br>You are not finished.         |
| <b>Step 4:</b><br>If the remainder is not 0 then integer1 becomes integer 2 and integer2 becomes the remainder | integer1 is now 108<br>integer2 is now 12               |
| <b>Step 5:</b><br>Go to <b>Step2</b> below:                                                                    |                                                         |
| <b>Step 2:</b><br>Divide integer1 by integer2 and compute the remainder.<br>Continue with <b>Step 3</b> below. | $108 / 12 = 9$<br>The remainder = 0                     |
| <b>Step 3:</b><br>If the remainder equals 0, you are finished. The GCF is integer2.                            | The remainder is 0<br>You are finished and the GCF = 12 |

Complete method `getGCF` below.

```
// precondition: n and d are integers greater than 0
// postcondition: returns the greatest common factor of n and d
private int getGCF (int n, int d)
```

Part (c).

Write method add as described earlier. Method add not only adds two Rational objects, it also reduces the resulting rational number before it is assigned to the current object. In writing method add, you may call method getGCF specified in part (b). Assume that getGCF works as specified, regardless of what you wrote in part (b).

Complete method add below.

```
// precondition: r1 and r2 are Rational objects
// postcondition: r1 is added to r2 according to the common rules
// of arithmetic for rational numbers and the final
// object is in lowest terms.
// The resulting num and den values are stored by the
// current object
public void add(Rational r1, Rational r2)
```

# Chapter 18 Questions

## Sample AP Examination III

---

### Section I

**Time** - 1 hour and 15 minutes

**Number of questions** - 40

**Percent of total grade** - 50

01. Consider the following method.

```
// precondition: m > 0
public static int mystery(int m)
{
 int counter = 0;
 for (int p = 1; p < m; p++)
 for (int q = p; q < m; q++)
 counter++;
 return counter;
}
```

What value is returned as a result of the call `mystery(8)` ?

- (A) 21
- (B) 28
- (C) 36
- (D) 81
- (E) 90

02. The Boolean expression

`(A && B) && (!A || !B)`

evaluates to

- (A) true in all cases.
- (B) false in all cases.
- (C) true only whenever both A is true and B is true.
- (D) false only whenever both A is false and B is false.
- (E) true only whenever A is true or B is true.

03. Consider the following method.

```
public static void mystery(int[] list)
{
 for (int k = 1; k < list.length; k++)
 if (k % 2 == 0)
 list[k] = list[k-1];
}
```

Assume that list contains the following values: {99,88,77,66,55,44,33,22,11}

Which of the following represents the contents of list as a result of calling method mystery?

- (A) {88,88,66,66,44,44,22,22}
- (B) {22,44,44,66,66,88,88}
- (C) {99,88,88,66,66,44,44,22,22}
- (D) {11,33,33,55,55,77,77,99,99}
- (E) {11,22,33,44,55,66,77,88,99}

04. Consider the following code segment.

```
int p = <some int value>
int q = <an int value, such that p > q>
int r = q - p + 1;
for (int k = 1; k <= 300; k++)
{
 int rndNum = (int) (Math.random() * r) + p;
 System.out.print(rndNum + " ");
}
```

What is the range of random numbers that can be assigned to rndNum?

- (A) [0 .. q-r]
- (B) [0 .. r]
- (C) [p .. q-r]
- (D) [q .. p]
- (E) [p .. q]

Questions 05-06 refer to the following information.

A method, called `getIntCount`, must be implemented to return the total count of `int` values from the starting `min` value parameter to the ending `max` value parameter, inclusive.

1st Example, the call `getIntCount(10, 20)` must return 11, based on values

`{10,11,12,13,14,15,16,17,18,19,20}.`

2nd Example, the call `getIntCount(-4, 7)` must return 12, based on values

`{-4,-3,-2,-1,0,1,2,3,4,5,6,7}.`

05. Consider the following implementation of method `getIntCount`.

```
public static int getIntCount(int min, int max)
{
 int count = max - min + 1;
 return count;
}
```

Method `getIntCount`, as implemented above, does work for many values, but testing the method with the call `System.out.println(getIntCount(Integer.MIN_VALUE, Integer.MAX_VALUE));` fails and prints the wrong value.

Which of following statements correctly explains the problem with the method's implementation?

- (A) `MAX_VALUE` and `MIN_VALUE` are members of the `Integer` class and cannot be assigned to an `int` variable.
- (B) The arithmetic computation is wrong and will not return the correct value.
- (C) The total number of `int` values exceeds the largest `int` value and cannot be stored correctly.
- (D) The required construction of an `Integer` object is missing in method `getIntCount`.
- (E) `MAX_VALUE` and `MIN_VALUE` must be called like methods, not handled like constant field values.

06. Consider the following three implementations of method `getIntCount`.

Note:     The `int` integer data type stores values in 4 bytes for with a 2,147,483,647 value.  
          The `long` integer data type stores values in 8 bytes with a 9,223,372,036,854,775,807 value.

Which of the following methods correctly implements the requirements of method `getIntCount`?

**Implementation 1**

```
public static int getIntCount(int min, int max)
{
 int count = max - min;
 return count;
}
```

**Implementation 2**

```
public static long getIntCount(int min, int max)
{
 long count = max - min + 1;
 return count;
}
```

**Implementation 3**

```
public static long getIntCount(int min, int max)
{
 long count = (long) max - min + 1;
 return count;
}
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementations 1 & 2 only
- (E) Implementations 2 & 3 only

07. Consider the following Student class.

```
class Student implements Comparable
{
 private String name;
 private int age;
 private double gpa;

 public Student (String n, int a, double g)
 {
 name = n;
 age = a;
 gpa = g;
 }

 public int compareTo(Object source)
 {
 /* missing code */
 }
}
```

The Student class implements the Comparable interface.

Student objects need to be compared according to their age standing.

The compareTo method needs to be implemented as follows:

student1.compareTo(student2) returns > 0, if student1's gpa > student2's gpa.  
student1.compareTo(student2) returns 0, if student1's gpa == student2's gpa.  
student1.compareTo(student2) returns < 0, if student1's gpa < student2's gpa.

Which of the following three implementations of /\*missing code \*/ will make method compareTo work as intended?

| <u>Implementation 1</u>                                                                                         | <u>Implementation 2</u>                                                                                                                     | <u>Implementation 3</u>                                                                                                                                 |
|-----------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>if (gpa &gt; source.gpa)     return 1; else if (age == source.age)     return 0; else     return -1;</pre> | <pre>Student temp = (Student)source; if (gpa &gt; temp.gpa)     return 1; else if (gpa == temp.gpa)     return 0; else     return -1;</pre> | <pre>Student temp = (Student)source; if (gpa.compareTo(temp.gpa))      return 1; else if (gpa.equals(temp.gpa))     return 0; else     return -1;</pre> |

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementations 1 and 2 only
- (E) Implementations 2 and 3 only

08. Consider the following code segment and Widget class.

```
Widget.setWidgets(1000);
Widget w1 = new Widget(100);
System.out.print(w1.getWidgets() + " ");
Widget w2 = new Widget(200);
System.out.print(w2.getWidgets() + " ");
Widget w3 = new Widget(300);
System.out.print(w3.getWidgets());
```

```
class Widget
{
 public static int numWidgets;

 public Widget(int n)
 {
 if (n > numWidgets)
 numWidgets = n;
 }

 public static void setWidgets(int n)
 {
 numWidgets = n;
 }

 public int getWidgets()
 {
 return numWidgets;
 }
}
```

What is output as a result of executing the code segment?

- (A) 100 200 300
- (B) 1000 100 200 300
- (C) 1000 1000 1000
- (D) 300 300 300
- (E) 100 100 100

09. Consider the following two methods.

```
// precondition: x > 0
public static int spain(int x)
{
 int temp1 = italy(x);
 int temp2 = italy(x);
 return temp1/temp2;
}

// precondition: n > 0
public static int italy(int n)
{
 if (n == 1)
 return 1;
 else
 return n * italy(n-1);
}
```

What value is returned as a result of the call `spain(q)` ?

- (A) 0
- (B) 1
- (C) ( $q$  factorial)/2
- (D)  $q$  factorial
- (E)  $2q$

10. Consider the following class and incomplete add method.

```
class MyList<E> extends ArrayList<E>
{
 public boolean add(E obj)
 {
 /* missing code */
 }
}
```

The add method implemented for the ArrayList class allows the addition of duplicate elements. This question shows the MyList class as a subclass of the ArrayList class. The MyList subclass must redefine the add method in such a manner that no duplicate elements are added.

Which of the following implementations can be used to replace /\* missing code \*/ so that the redefined add method does not add duplicate elements to its object?

**Implementation 1**

```
if (this.contains(obj))
 return false;
else
{
 super.add(obj);
 return true;
}
```

**Implementation 2**

```
if (contains(obj))
 return false;
else
{
 super.add(obj);
 return true;
}
```

**Implementation 3**

```
if (MyList.contains(obj))
 return false;
else
{
 super.add(obj);
 return true;
}
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementations 1 and 2 only
- (E) Implementations 2 and 3 only

11. Consider the following three classes.

```
class Foundation
{
 public Foundation()
 {
 System.out.println("Foundation is constructed");
 }
}

class Roof
{
 public Roof()
 {
 System.out.println("Roof is constructed");
 }
}

class House
{
 private Foundation foundation;
 private Roof roof;

 public House()
 {
 System.out.println("House is constructed");
 roof = new Roof();
 foundation = new Foundation();
 }
}
```

Assume that the following code segment appears in a client program. What is printed by the code segment?

```
House house = new House();
```

- (A) House is constructed
- (B) Foundation is constructed  
House is constructed  
Roof is constructed
- (C) Foundation is constructed  
Roof is constructed  
House is constructed
- (D) House is constructed  
Roof is constructed  
Foundation is constructed
- (E) Roof is constructed  
House is constructed  
Foundation is constructed

12. Consider the following method.

```
public static void change(int[][] matrix)
{
 int[][] temp = matrix;
 for (int p = 1; p < temp.length; p++)
 {
 for (int q = 0; q < temp[0].length; q++)
 {
 int swap = temp[p][q];
 temp[p][q] = temp[p-1][q];
 temp[p-1][q] = swap;
 }
 }
}
```

Which of the following correctly describes the integers stored by `matrix` after a call to method `change`?

- (A) The top row is moved to the bottom.
- (B) The bottom row is moved to the top.
- (C) All the rows are in reverse order.
- (D) `matrix` is reversed both in rows and columns.
- (E) `matrix` is unchanged.

13. Consider the following code segment.

```
for (int k = 1; k <= 10; k++)
{
 int rndInt = (int) (Math.random() * 900) + k;
 System.out.print(rndInt + " ");
}
```

What is printed as a result of executing the code segment?

- (A) A set of random integers in the [0..899] range.
- (B) A set of random integers in the [1..900] range.
- (C) A set of random integers in the [1..909] range.
- (D) A set of random integers in the [10..909] range.
- (E) A set of random integers in the [10..910] range.

14. Consider the following method.

```
// precondition: list1 is a non-empty array and list2 is an empty array
public static void alterList(ArrayList<String> list1, ArrayList<String> list2)
{
 for (String name1: list1)
 {
 boolean OK = true;
 for (String name2: list2)
 {
 if (name1.equals(name2))
 OK = false;
 }
 if (OK)
 list2.add(name1);
 }
}
```

What is stored in list2 after a call to alterList?

- (A) list2 copies all the values of list1, but stores no duplicate values.
- (B) list2 is a copy of list1.
- (C) list2 stores only any duplicate values of list1.
- (D) list2 stores only any consecutive values of list1 that are duplicates.
- (E) list2 stays an empty list.

15. Consider the following method.

```
// precondition: names is a non-empty array of String values.
public static void croatia(String[] names)
{
 for (String name: names)
 for (int q = 0; q < names.length-1; q++)
 if (names[q].compareTo(names[q+1]) > 0)
 {
 String temp = names[q];
 names[q] = names[q+1];
 names[q+1] = temp;
 }
}
```

How are the elements arranged in names, after a call to method croatia?

- (A) The elements are arranged in reverse order.
- (B) The elements are returned to their original order.
- (C) The elements are sorted in ascending order.
- (D) The elements are sorted in descending order.
- (E) The elements are arranged in a random order.

16. The primary benefit of Object Oriented Programming is greater program reliability.

Polymorphism is one of the features of Object Oriented Programming.

How does polymorphism make a program more reliable?

I. Polymorphism keeps the data and the methods that process the data in the same object.

II. Polymorphism insures that an object is responsible for the processing or behavior of its own object.

III. Polymorphism uses the methods of tested, established classes.

(A) I only

(B) II only

(C) III only

(D) I and II only

(E) II and III only

17. Today's airports do not process their data (luggage and passengers) in an Object Oriented Manner.

Imagine a future airport where passengers and luggage are handled as follows:

1. Passengers enter a closed hallway restricted to their flight destination.

2. Passengers check in their luggage with an agent who only handles one destination.

3. The luggage is placed on a conveyor belt dedicated for one destination only.

Such a future airport resembles Object Oriented Programming by using which one of the following OOP features?

(A) encapsulation

(B) inheritance

(C) composition

(D) polymorphism

(E) debugging

18. Consider the following method.

```
//precondition; n < 0
public static void ifElse(int n)
{
 if (n == 0)
 {
 System.out.print(n + " ");
 ifElse(n + 1);
 }
 else if (n == -1)
 {
 System.out.print(n + " ");
 ifElse(n + 2);
 }
 else if (n == -3)
 {
 System.out.print(n + " ");
 ifElse(n + 3);
 }
 else if (n == -4)
 {
 System.out.print(n + " ");
 ifElse(n + 4);
 }
 else
 System.out.print(n + " ");
}
```

Which of the following is printed as a result of the call `ifElse(-5)`?

- (A) -5
- (B) -5 -4
- (C) -5 -4 -3
- (D) -5 -4 -3 -2
- (E) -5 -4 -3 -2 -1

19. Consider the following two classes.

```
class Fruit
{
 private int quantity;

 public Fruit(int q)
 {
 quantity = q;
 }

 public String toString()
 {
 String temp = quantity + " ";
 return temp;
 }
}

class Apple extends Fruit
{
 private String name;

 public Apple(String n, int q)
 {
 super(q);
 name = n;
 }

 public String toString()
 {
 return super.toString() + name;
 }
}
```

Assume that the following code segment appears in a client program.  
What is printed by the code segment?

```
Apple apple = new Apple("Red Delicious",1000);
System.out.println(apple);
```

- (A) 1000
- (B) Red Delicious
- (C) 1000 Red Delicious Apples
- (D) Red Delicious 1000
- (E) 1000 Red Delicious

20. Consider the following method.

```
// precondition: nr > 0
public static String monkey(int nr)
{
 String temp = "]";
 while (nr > 0)
 {
 temp = (nr % 3) + temp;
 nr /= 3;
 }
 return "[" + temp;
}
```

What string is returned as a result of the call `monkey(n)`?

- (A) A string displaying the multiples of 3 of `nr`, like [ 3 6 9 12 ... ]
- (B) A string containing the sum of the binary numbers of `nr`, like [ 1+2+4+8+16+... ]
- (C) A string containing the base-3 representation of `nr`
- (D) A string containing the base-2 representation of `nr`
- (E) A string containing all the factors of `nr`

Questions 21-22 refer to the following incomplete class declaration.

```
class Game
{
 /* highScore declaration which stores highest score recorded from multiple game objects. */

 private int score;

 public Game()
 {
 playGame();
 if (score > highScore)
 highScore = score;
 }

 public static int getHS()
 {
 return highScore;
 }

 public int getScore()
 {
 return score;
 }

 public void playGame()
 {
 score = (int) (Math.random() * 10000);
 }

 /* declaration of setHS method, which updates the highScore value */
}
```

21. Class Game attribute highScore stores the highest score recorded from multiple game objects. Which of the following declarations satisfies that requirement?

**Declaration I**

private static int highScore;

**Declaration II**

private int highScore;

**Declaration III**

public int highScore;

- (A) Declaration I only
- (B) Declaration II only
- (C) Declaration III only
- (D) Declarations I and II only
- (E) Declarations II and III only

22. Method `setHS`, updates the `highScore` value.  
Which of the following declarations satisfies that requirement?

**Declaration I**

```
private static void setHS(int newScore)
{
 highScore = newScore;
}
```

**Declaration II**

```
public static void setHS(int newScore)
{
 highScore = newScore;
}
```

**Declaration III**

```
public void setHS(int newScore)
{
 highScore = newScore;
}
```

- (A) Declaration I only
- (B) Declaration II only
- (C) Declaration III only
- (D) Declarations I and II only
- (E) Declarations II and III only

23. Consider the following program segment.

```
ArrayList<Integer> numbers1 = new ArrayList<Integer>();
ArrayList<Integer> numbers2 = new ArrayList<Integer>();

for (int k = 1; k <= 15; k++)
{
 int randomInt = (int) (Math.random() * 100);
 numbers1.add(new Integer(randomInt));
}

for (Integer number: numbers1)
 if (number.intValue() % 2 != 0)
 numbers2.add(number);
```

What type of integers are stored in `numbers2` as a result of executing the program segment?

- (A) Only even integer values are stored in `numbers2`.
- (B) Only odd integer values are stored in `numbers2`.
- (C) All even integer values are replaced with null in `numbers2`.
- (D) All odd integer values are replaced with null in `numbers2`.
- (E) All values in `numbers2` are null.

24. Consider the following method.

```
// precondition: s1.length() equals s2.length()
public static String sillyString(String s1, String s2)
{
 String temp1 = s2.substring(0,s1.length());
 String temp2 = s1.substring(0,s2.length());
 String temp3 = temp2 + temp1;
 return temp3;
}
```

What value is returned as a result of the call `sillyString(string1,string2)` ?

- (A) `string1 + string2`
- (B) `string2 + string1`
- (C) `string1`
- (D) `string2`
- (E) `string1 + string2 + string1`

25. Consider the following method and code segment.

```
public static double jack(double seed)
{
 seed = seed + Math.PI;
 seed = Math.sqrt(seed);
 seed = seed - (int) seed;
 return seed;
}

double seed = 1000.0;
for (int k = 1; k < 10; k++)
{
 seed = jack(seed);
}
```

What values are returned as a result of executing the code segment?

- (A) Ten double values such that for each  $x \quad 0 < x < 1$
- (B) Ten double values such that for each  $x \quad 0 < x \leq 1$
- (C) Ten double values such that for each  $x \quad 0 < x < \text{Math.PI}$
- (D) Ten double values such that for each  $x \quad 0 < x \leq \text{Math.PI}$
- (E) Ten double values such that for each  $x \quad 1 < x < \text{Math.PI}$

26. Consider the following method.

```
// Precondition: p > 0; q > 0; p > q
public static int canada(int p, int q)
{
 if (p > q)
 return canada(p-1,q+1);
 else
 return p + q;
}
```

What value is returned as a result of the call `canada(a,b)`?

- (A) a
- (B) b
- (C) a + b
- (D) 1
- (E) 0

27. Consider the following interface and classes declarations.

```
abstract interface XYZ
{
 public abstract int getData();
 public void setData(int n);
 public abstract void showData();
}

abstract class PQR implements XYZ
{
 protected int number;
 public PQR(int n) { number = n; }
 public void setData(int n) { number = n; }
 public void showData() { System.out.println(number); }
}

class ABC extends PQR
{
 public ABC(int n) { super(n); }
}
```

Which one of these three declarations is correct and will compile?

- (A) Only interface XYZ is correct.
- (B) Only interface XYZ and abstract class PQR are correct.
- (C) Only interface XYZ and class ABC are correct.
- (D) Only abstract class PQR and class ABC are correct.
- (E) interface XYZ, abstract class PQR and class ABC are all correct.

28. Consider the following code segment and incomplete method `convertArrays`.

```
double list1[] = {1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9};
ArrayList<Double> list2 = staticToDynamic(list1);
```

Method `staticToDynamic` takes the double values stored in the static `list1` array, adds them to the dynamic `list2` array and returns the dynamic array object.

Which of the following methods correctly implements `convertArrays`?

**Implementation 1**

```
public static ArrayList<Double> staticToDynamic(double[] temp1)
{
 ArrayList<Double> temp2 = new ArrayList<Double>();
 for (double number : temp1)
 {
 temp2.add(number);
 }
 return temp2;
}
```

**Implementation 2**

```
public static ArrayList<Double> staticToDynamic(double[] temp1)
{
 ArrayList<Double> temp2 = new ArrayList<Double>();
 for (int index = 0; index < temp1.length; index++)
 {
 temp2.add(temp1[index]);
 }
 return temp2;
}
```

**Implementation 3**

```
public static ArrayList<Double> staticToDynamic(double[] temp1)
{
 ArrayList<Double> temp2 = new ArrayList<Double>();
 for (int index = 0; index < temp1.length; index++)
 {
 double number = temp1[index];
 temp2.set(index, number);
 }
 return temp2;
}
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementations 1 and 2 only
- (D) Implementations 2 and 3 only
- (E) Implementations 1, 2 and 3

29. Consider the following three classes.

```
class Eye
{
 private String color;

 /* constructor */
}

class Person
{
 private int age;
 private Eye eye;

 /* constructor */
}

class Student extends Person
{
 /* constructor */
}
```

What type of class relationship(s) is/are used by these three classes?

- (A) Inheritance only
- (B) Composition only
- (C) Encapsulation only
- (D) Inheritance and Composition only
- (E) Composition and Encapsulation only

30. Consider the following code segment.

```
double result = (double) 4 + 3 + 2 + 1/4;
System.out.println(result);
```

What is printed as a result of executing the code segment?

- (A) 2.5
- (B) 9.25
- (C) 9.0
- (D) 10.25
- (E) ArithmeticException runtime error

31. Consider the following two methods.

```
public static boolean method1(int n)
{
 System.out.println("Calling method 1");
 return n < 10;
}

public static boolean method2(int n)
{
 System.out.println("Calling method 2");
 return n < 10;
}
```

Assume that the following code segment appears in a client program.

```
for (int n = 9; n < 11; n++)
{
 if (method1(n) || method2(n))
 System.out.println();
}
```

What is printed as a result of executing the code segment?

- (A) Calling method1

Calling method1  
Calling method2

- (B) Calling method1  
Calling method2

- (C) Calling method1  
Calling method1

- (D) Calling method2  
Calling method2

- (E) Calling method1  
Calling method2

Calling method1

32. Consider the following code segment.

```
public static void main(String args[])
{
 boolean bool = <true or false>
 bool = !(bool == true || bool == false);
 System.out.println(bool);
}
```

What is printed as a result of executing the code segment?

- (A) The code segment always prints true.
- (B) The code segment always prints false.
- (C) The code segment prints true, if the initial value of bool is true and false otherwise.
- (D) The code segment prints true, if the initial value of bool is false and true otherwise.
- (E) The code segment will print true or false, but the actual value cannot be determined.

33 Consider the following method.

```
public static void swing(int[] list)
{
 for (int p: list)
 {
 for (int q = 0; q < list.length-1; q++)
 if (list[q] < list[q+1])
 {
 int temp = list[q];
 list[q] = list[q+1];
 list[q+1] = temp;
 }
 }
}
```

Assume that the following code segment appears in a client program.

```
int[] intArray = {66,22,77,11,44,55,33,99,88};
swing(intArray);
for (int k = 0; k < intArray.length; k++)
 System.out.print(intArray[k] + " ");
```

What is printed as a result of executing the code segment?

- (A) 88 99 33 55 44 11 77 22 66
- (B) 99 88 77 66 55 44 33 22 11
- (C) 11 22 33 44 55 66 77 88 99
- (D) 22 66 11 44 55 33 77 88 99
- (E) 66 22 77 11 44 55 33 99 88

34. Which of the following is **false** about using a binary search routine to find an element in an arbitrary array?
- (A) The binary search works correctly, but inefficiently when the array is not sorted.
  - (B) It is possible that the binary search finds the requested element.
  - (C) The binary search may indicate that the search item is not found when it does exist in the array.
  - (D) A binary search can work correctly if the list is sorted in descending order.
  - (E) A binary search can work correctly if the list is sorted in ascending order.
35. Consider an interface, which may only have abstract methods, and an abstract class, which also has only abstract methods. Is there any syntactical difference between using the interface or the abstract class in this situation?
- (A) No, because in both cases a concrete class must implement all the classes.
  - (B) No, because an abstract class with only abstract methods is identical to an interface.
  - (C) Yes, because a concrete class must implement the interface and extends the abstract class.
  - (D) Yes, because the abstract class has a constructor and the interface does not.
  - (E) Yes, because you cannot instantiate an interface, but you can instantiate an abstract class.

36. Consider the following incomplete method `linearSearch`.

```
// precondition: list is sorted in ascending order
// postcondition: linearSearch returns index location if number exists,
// list and -1 otherwise
public int linearSearch(int[] list, int number)
{
 boolean found = false;
 int k = 0;
 /* missing code */
}
```

Which of the following implementations of /\* missing code \*/ takes advantage of a sorted list where the search number is not in the list array?

**Implementation 1**

```
while (k < list.length && !found)
{
 if (list[k] == number)
 found = true;
 else
 k++;
}
if (found)
 return k;
else
 return -1;
```

**Implementation 2**

```
while (k < list.length)
{
 if (list[k] == number)
 found = true;
 else
 k++;
}
if (found)
 return k;
else
 return -1;
```

**Implementation 3**

```
while (k < list.length && !done)
{
 if (list[k] == number)
 {
 found = true;
 done = true;
 }
 else
 {
 if (number < list[k])
 done = true;
 else
 k++;
 }
}
if (found)
 return k;
else
 return -1;
```

- (A) Implementation 1 only
- (B) Implementation 2 only
- (C) Implementation 3 only
- (D) Implementations 1 and 2 only
- (E) Implementations 2 and 3 only

37. Consider the following code segment.

```
int[][] m = { {11,12,13,14,15},
 {16,17,18,19,20},
 {21,22,23,24,25},
 {26,27,28,29,30},
 {31,32,33,34,35} };

for (int r = 0; r < m.length; r++)
{
 for (int c = 0; c < m[0].length; c++)
 if (r == c)
 System.out.print(m[r][c] + " ");
 else
 System.out.print("00 ");
 System.out.println();
}
```

Which is printed as a result of executing the code segment?

- (A) 11 12 13 14 15  
 16 17 18 19 20  
 21 22 23 24 25  
 26 27 28 29 30  
 31 32 33 34 35
- (B) 00 00 00 00 15  
 00 00 00 19 00  
 00 00 23 00 00  
 00 27 00 00 00  
 31 00 00 00 00
- (C) 11 00 00 00 00  
 00 17 00 00 00  
 00 00 23 00 00  
 00 00 00 29 00  
 00 00 00 00 35
- (D) 11 12 13 14 15  
 16 00 00 00 20  
 21 00 00 00 25  
 26 00 00 00 30  
 31 32 33 34 35
- (E) 00 00 00 00 00  
 00 17 18 19 00  
 00 22 23 24 00  
 00 27 28 29 00  
 00 00 00 00 00

38. Consider the following two Boolean expressions.

I.  $a \&\& (a \mid\mid (b \&\& c))$

II.  $a \mid\mid (a \&\& (b \mid\mid c))$

*Blank* dictates that Expression I is always `false` when `a` equals `false`.

*Blank* dictates that Expression II is always `true` when `a` equals `true`.

What is *Blank*?

- (A) short circuiting
- (B) De Morgan's Law
- (C) Boolean distribution
- (D) Boolean inheritance
- (E) Polymorphism

39. Consider the following code segment.

```
ArrayList<String> names = new ArrayList<String>();
names.add("John");
names.add("Greg");
names.add("Maria");
names.add("Heidi");

int n = names.size()-1;
for (int k = 0; k < n; k++)
{
 names.set(k,names.get(k+1));
 names.set(k+1,names.get(k));
}
System.out.println(names);
```

What is output as a result of executing the code segment?

- (A) John John John John
- (B) Greg Greg Greg Greg
- (C) Greg Maria Heidi John
- (D) Greg Maria Heidi Heidi
- (E) Heidi Maria Greg John

40. Consider the following method.

```
// Precondition: list is a non-empty array of int element.
// Postcondition: panda returns true if number exists in numbers,
// and false otherwise
public static boolean panda(int[] numbers, int number)
{
 boolean found = false;
 for (int item: numbers)
 {
 if (number == item)
 found = true;
 else
 found = false;
 }
 return found;
}
```

For which of following executing scenarios is the postcondition of `panda` satisfied?

- (A) `panda` will always execute correctly.
- (B) `panda` will never execute correctly.
- (C) `panda` executes correctly if `number` exists in `numbers`,  
but executes incorrectly when `number` does not exist.
- (D) `panda` executes incorrectly if `number` exists in `numbers`, unless `number` is the last element,  
and executes correctly when `number` does not exist.
- (E) The execution behavior of `panda` cannot be determined with the given information.

# Chapter 18 Free Response Questions

## Sample AP Examination III

---

### Section II

**Time** - 1 hour and 45 minutes

**Number of questions** - 4

**Percent of total grade** - 50

#### Question 1.

This question involves converting numbers from Base16 to Base 10.

##### Part (a).

Write method `fromHexToBin`, which is described as follows. Method `fromHexToBin` converts the base-16 argument and converts the number to its base-2 equivalent. The base-2 number is returned.

In completing method `fromHexToBin` you can use the following table, which details the unique relationship between base-16 and base-2 numbers. Every single base-16 digit corresponds to four base-2 digits.

|         |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |
|---------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| Base 16 | 0    | 1    | 2    | 3    | 4    | 5    | 6    | 7    | 8    | 9    | a    | b    | c    | d    | e    | f    |
| Base-2  | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |

Complete method `fromHexToBin` below.

```
// precondition: hexNum is a String of characters, which represents
// a correct base-16 number.
// postcondition: returns a String of characters, which represents
// the base-2 number equivalent of hexNum.
public static String fromHexToBin(String hexNum)
```

**Part (b).**

Write method `fromBinToDec`, which is described as follows. Method `fromBinToDec` converts the base-2 argument and converts the number to its base-10 equivalent. The base-10 number is returned.

Complete method `fromBinToDec` below.

```
// precondition: binNum is a String of characters, which represents
// a correct base-2 number.
// postcondition: returns an int value, which represents
// the base-10 number equivalent of binNum.
public static int fromBinToDec(String binNum)
```

**Part (c).**

Write method `fromHexToDec`, which is described as follows. Method `fromHexToDec` converts the base-16 argument and converts the number to its base-10 equivalent. The base-10 number is returned.

In writing method `fromHexToDec` you may call methods `fromHexToBin` and `fromBinToDec`, which are specified in Part (a) and Part(b), whether the methods are implemented correctly or not.

Complete method `fromHexToDec` below.

```
// precondition: hexNum is a String of characters, which represents
// a correct base-16 number.
// postcondition: returns an int value, which represents
// the base-10 number equivalent of hexNum.
public static int fromHexToDec(String hexNum)
```

## Question 2.

This question involves using searching methods for specified elements in an array. The *binary search* is far more efficient than the *linear search*, but a sorted array of elements is required.

For this question assume that some class exists with an **IntArray** attribute, which stores **int** values. It is not known if IntArray stores integers in any particular order, but it is possible. Searching for integer values is one of the primary functions of the class. The class

Part (a).

Write method `linearSearch`, which is described as follows. Method `linearSearch` compares the provided search number argument `sn` with element in the `intArray`. If the element exists in the array the index position is returned and `-1` otherwise.

Complete method `linearSearch` below.

```
// precondition: intArray is a non-empty static array of int values in
// the same class as linearSearch.
// sn is an integer value, which may exist in intArray.
// postcondition: returns the index position of sn in intArray, if
// sn exist and -1 otherwise.
public int linearSearch(int sn)
```

### Part (b).

The binary search is also part of the class, but it can only be called if `intArray` is a sorted array of numbers. Write method `isSorted`, which is described as follows.

Method `isSorted` determines if the elements of `intArray` are in ascending order.

Complete method `isSorted` below.

```
// precondition: intArray is a non-empty static array of int values in
// the same class as IsSorted.
// postcondition: returns true if intArray is sorted in ascending
// order, and false otherwise.
public boolean isSorted()
```

### Part (c).

Write method `binarySearch`, which is described as follows. Method `binarySearch` compares the provided search number argument `sn` with element in the `intArray`. If the element exists in the array the index position is returned and `-1` otherwise. You may assume that method `isSorted` was called before `binarySearch` is called to insure that the `intArray` list is sorted in ascending order.

Complete method `binarySearch` below.

```
// precondition: intArray is a non-empty static array of int values in
// the same class as binarySearch.
// intArray is sorted in ascending order.
// sn is an integer value, which may exist in intArray.
// postcondition: returns the index position of sn in intArray, if
// sn exist and -1 otherwise.
public int binarySearch(int sn)
```

### Question 3.

Consider the following incomplete declaration of a `StringList` class. The intent of the `StringList` class is to facilitate data processing of string information stored in a `strings` array. For this question data processing will be limited to three actions, which are adding strings, sorting strings and displaying strings.

```
public class StringList
{
 private ArrayList<String> strings;
 private int size;

 public StringList()
 {
 strings = new ArrayList<String>();
 size = 0;
 }

 public void add(String str)
 {
 /* to be implemented by part (a) */
 }

 private void sortStrings()
 {
 /* to be implemented by part (b) */
 }

 public String toString()
 {
 /* to be implemented by part (c) */
 }
}
```

**Part (a).**

Write method `add`, which is described as follows. Method `add` adds new elements to the `strings` array. The `strings` array is a dynamic array implemented with an object of the `ArrayList` class.

Complete method `add` below.

```
// postcondition: the String value of str is added to the strings array.
public void add(String str)
```

**Part (b).**

Write method `sortStrings`, which is described as follows. Method `sortStrings` arrange the elements of the `strings` array in ascending order.

Complete method `sortStrings` below.

```
// precondition: strings is a non-empty dynamic array of String values.
// postcondition: the strings array is sorted in ascending order.
private void sortStrings()
```

**Part (c).**

Write method `toString`, which is described as follows. Method `toString` enables the display of the `StringList` object.

If the strings array contains the following string value:

{Tom, Sue, Joe,Meg},

then method `toString` will return

[Tom, Sue, Joe, Meg]

In writing method `toString` you may call method `sortStrings`, which is specified in part (a), whether the method is implemented correctly or not.

Complete method `toString` below.

```
// precondition: strings is a non-empty dynamic array of String values.
// postcondition: the elements in the strings array are sorted in
// ascending order.
// returns a string in the format:
// [string1, string2, string3,]
private void toString()
```

#### Question 4.

Consider the following incomplete declaration of a `Matrix` class. The intent of the `Matrix` class is to facilitate mathematical operations performed on two-dimensional, rectangular arrays. `Matrix` objects can be constructed with row and column size information only or with a provided integer array parameter. A `Matrix` class can have many useful operations. For this question the focus will be on writing methods for matrix multiplication and matrix display.

Method `multiply` takes two `Matrix` objects and returns the product. Method `okSize` is a private helper method for method `multiply` and determines if the provided `Matrix` objects can be multiplied. You will also need to redefine the `toString` method to display the elements of a `Matrix` object.

```
public class Matrix
{
 private int twoD[][];
 private int numRows;
 private int numCols;

 public Matrix(int r, int c)
 {
 numRows = r;
 numCols = c;
 twoD = new int[numRows][numCols];
 }

 public Matrix(int array[][])
 {
 twoD = array;
 numRows = twoD.length;
 numCols = twoD[0].length;
 }

 public String toString()
 {
 /* to be implemented by part (a) */
 }

 private boolean okSize(Matrix m1, Matrix m2)
 {
 /* to be implemented by part (b) */
 }

 public void multiply(Matrix m1, Matrix m2)
 {
 /* to be implemented by part (c) */
 }

 // other methods used by the Matrix class
}
```

**Part (a).**

Redefine the `toString` method as discussed earlier. Method `toString` must return a string that will display a `Matrix` in two-dimensional array format. The following table shows the result of several different calls made by `System.out.println(/* Matrix Object */)` to the `toString` method based on the `String` argument of the constructor.

The code segment below is part of a client program that constructs and displays three `Matrix` objects.

| Code segment in client program using the <code>Matrix</code> class                                                                                                                                                             | Execution Output        |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------|
| <code>int[][] mat1 = {{1,2,3},<br/>                 {4,5,6},<br/>                 {7,8,9}};</code>                                                                                                                             | 1 2 3<br>4 5 6<br>7 8 9 |
| <code>int[][] mat2 = {{1,0,0},<br/>                 {0,1,0},<br/>                 {0,0,1}};</code>                                                                                                                             | 1 0 0<br>0 1 0<br>0 0 1 |
| <code>Matrix m1 = new Matrix(mat1);<br/>Matrix m2 = new Matrix(mat2);<br/>Matrix m3 = new Matrix(3,3);</code>                                                                                                                  | 0 0 0<br>0 0 0<br>0 0 0 |
| <code>System.out.println(m1);<br/>System.out.println();<br/>System.out.println(m2);<br/>System.out.println();<br/>System.out.println(m3);<br/>m3.multiply(m1,m2);<br/>System.out.println();<br/>System.out.println(m3);</code> | 1 2 3<br>4 5 6<br>7 8 9 |

Complete method `toString` below.

```
// precondition: twoD is a nonempty two-dimensional integer array
// postcondition: returns a string in rectangular matrix format.
public String toString()
```

### Part (b).

Matrix multiplication involves a process of adding the products of elements of rows in one matrix times corresponding elements of columns in a second matrix. This process is demonstrated by the matrix multiplication example below.

$$\begin{array}{cc} a & b \\ c & d \end{array} * \begin{array}{cc} e & f \\ g & h \end{array} = \begin{array}{cc} ae + bg & af + bh \\ ce + dg & cf + dh \end{array}$$

It is not necessary that matrices are square matrices or identical matrices. Two matrices can be multiplied as long as the number of columns in the first matrix are equal to the number of rows in the second matrix. The next example involves a **2 x 3** matrix and a **3 x 2** matrix.

$$\begin{array}{ccc} a & b & c \\ d & e & f \end{array} * \begin{array}{cc} g & h \\ i & j \\ k & l \end{array} = \begin{array}{cc} ag + bi + ck & ah + bj + cl \\ dg + ei + fk & dh + ej + fl \end{array}$$

Observe that the **2 x 3** matrix is multiplied by a **3 x 2** matrix and creates a **2 x 2** matrix product. A general formula can be stated about the matrix multiplication, which is:

An **a x b** matrix times a **b x c** matrix becomes an **a x c** matrix.

This formula shows that there are special size requirements for all three matrices involved in the multiplication process. The column size (**b**) of the first matrix must be the same as the row size (**b**) of the second matrix. Furthermore, the product matrix must have row size (**a**) and column size (**c**).

Write method `okSize`, which is described as follows. Method `okSize` is used to see if the three matrices involved in multiplication follow the correct rules of row and column size. Method `okSize` returns true if the number of columns of `m1` equals the number of rows of `m2` and the product matrix has the same number of rows as `m1` and the same number of columns as `m2`.

Complete method `okSize` below.

```
// precondition: m1 and m2 are not null
// postcondition: returns true if m1 columns equals m2 rows and
// m1 rows equals object's rows and m2 columns
// equals object's columns
private boolean okSize(Matrix m1, Matrix m2)
```

**Part (c).**

Write method `multiply` as described earlier. In writing method `multiply`, you may call method `okSize`, as specified in part (b). Assume that `okSize` works as specified, regardless of what you wrote in part (b).

Complete method `multiply` below.

```
// precondition: m1 and m2 are not null
// postcondition: the product of the m1 and m2 is stored in the
// current object, if possible, otherwise the
// current object remains unchanged
public void multiply(Matrix m1, Matrix m2)
```