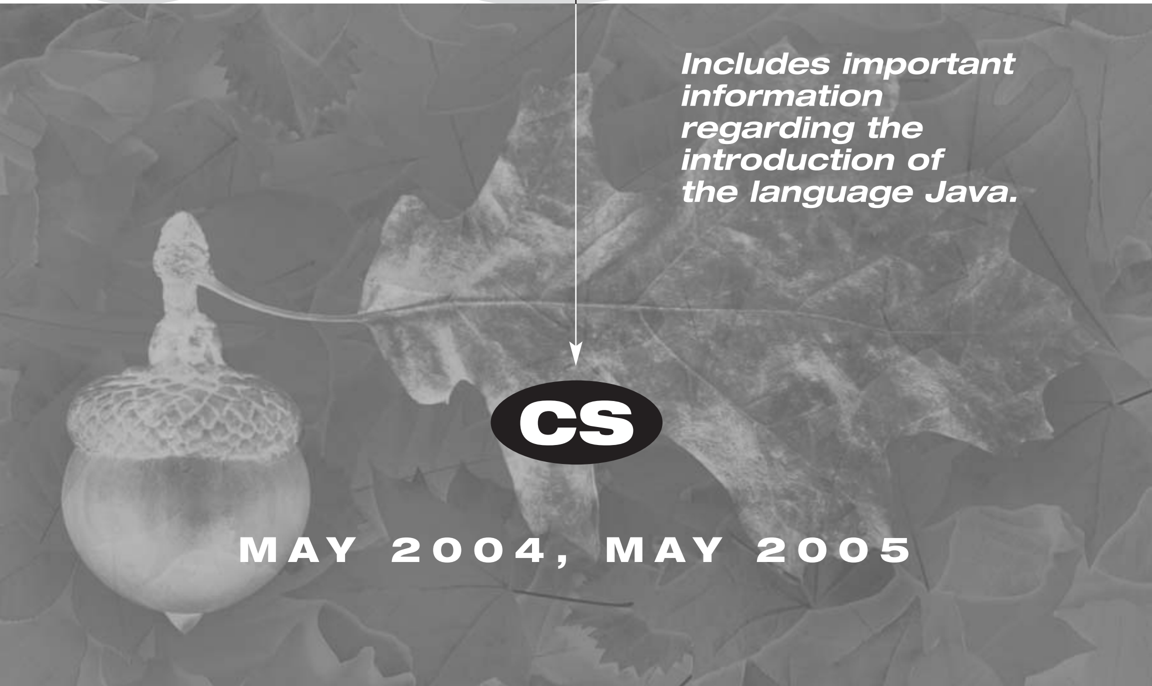
The College Board

AP® ADVANCED PLACEMENT PROGRAM®

Course Description

COMPUTER SCIENCE



Includes important information regarding the introduction of the language Java.

CS

MAY 2004, MAY 2005

Contents

Welcome to the AP® Program

The Advanced Placement Program® (AP®) is a collaborative effort between motivated students, dedicated teachers, and committed high schools, colleges, and universities. Since its inception in 1955, the Program has allowed millions of students to take college-level courses and exams, and to earn college credit or placement while still in high school.

Most colleges and universities in the U.S., as well as colleges and universities in 21 other countries, have an AP policy granting incoming students credit, placement, or both on the basis of their AP Exam grades. Many of these institutions grant up to a full year of college credit (sophomore standing) to students who earn a sufficient number of qualifying AP grades.

Each year, an increasing number of parents, students, teachers, high schools, and colleges and universities turn to AP as a model of educational excellence.

More information about the AP Program is available at the back of this Course Description and at AP Central™, the College Board's online home for AP professionals (apcentral.collegeboard.com). Students can find more information at the AP student site (www.collegeboard.com/apstudents).

AP Courses

Thirty-four AP courses in a wide variety of subject areas are currently available. Developed by a committee of college faculty and AP teachers, each AP course covers the breadth of information, skills, and assignments found in the corresponding college course. See page 2 for a list of the AP courses and exams that are currently offered.

AP Exams

Each AP course has a corresponding exam that participating schools worldwide administer in May. Except for Studio Art, which is a portfolio assessment, AP Exams contain multiple-choice questions and a free-response section (either essay or problem-solving).

AP Exams represent the culmination of AP courses, and are thus an integral part of the Program. As a result, many schools foster the expectation that students who enroll in an AP course will go on to take the corresponding AP Exam. Because the College Board is committed to providing homeschooled students and students whose schools do not offer AP access to the AP Exams, it does not require students to take an AP course prior to taking an AP Exam.

AP Courses and Exams

Art
Art History
Studio Art (Drawing Portfolio)
Studio Art (2-D Design Portfolio)
Studio Art (3-D Design Portfolio)

Biology

Calculus
Calculus AB
Calculus BC

Chemistry

Computer Science
Computer Science A
Computer Science AB

Economics
Macroeconomics
Microeconomics

English
English Language and Composition
English Literature and Composition

Environmental Science

French
French Language
French Literature

German Language

Government and Politics
Comparative Government and Politics
United States Government and Politics

History
European History
United States History
World History

Human Geography

Latin
Latin Literature
Latin: Vergil

Music Theory

Physics
Physics B
Physics C: Electricity and Magnetism
Physics C: Mechanics

Psychology

Spanish
Spanish Language
Spanish Literature

Statistics

AP Computer Science

Introduction

AP Computer Science courses and examinations will be administered using the Java programming language beginning with the 2003-04 academic year and the 2004 examinations.

This Course Description contains many revisions from previous years due to the change in the computer language used to illustrate computer science concepts covered in the AP Computer Science courses. The topic outline has been revised to reflect new topics supported by the language and new sample questions are included. Read the following information carefully to verify that your course offerings contain appropriate materials for student preparation for the exams.

The Advanced Placement Program offers two computer science courses: Computer Science A and Computer Science AB. The content of Computer Science A is a subset of the content of Computer Science AB. Computer Science A emphasizes object-oriented programming methodology with a concentration on problem solving and algorithm development and is meant to be the equivalent of a first-semester college-level course in Computer Science. It also includes the study of data structures, design, and abstraction, but these topics are not covered to the extent that they are in Computer Science AB. Computer Science AB includes all the topics of Computer Science A, as well as a more formal and in-depth study of algorithms, data structures, design, and abstraction. For example, binary trees are studied in Computer Science AB but not in Computer Science A. For a listing of the topics covered, see the AP Computer Science topic outline on pages 8–13.

Computer Science A may be appropriate for schools offering an AP Computer Science course for the first time, for schools whose faculty members have not yet developed sufficient expertise to cover the material in Computer Science AB, or for schools wishing to offer a choice of courses.

The nature of both AP courses is suggested by the words “computer science” in their titles. Their presence indicates a disciplined approach to a more broadly conceived subject than would a descriptor such as “computer programming.” There are no computing prerequisites for either AP course. Each is designed to serve as a first course in computer science for students with no prior computing experience.

Because of the diversity of introductory computer science courses currently offered by colleges and universities, the outline of topics described here may not match any sequence of courses exactly. The Association for Computing Machinery (ACM) and the Institute of Electrical and Electronic

Engineers (IEEE) Computer Society have published standards for the content of a college-level program in computer science that include recommendations for topics to be covered in the first two years of college. The AP Computer Science A course is compatible with those topics that are covered in a typical CS1 course as described in the example curricula in the ACM/IEEE guidelines. The additional topics in the AP Computer Science AB course are consistent with a CS2 course in those sample curricula. Some colleges and universities may organize their curricula in alternative ways so that the topics of the AP Computer Science A and AB courses are spread over the first three or four college courses, with other topics from computer science interspersed.

Either AP Computer Science course can be offered by any secondary school that has faculty who possess the necessary expertise and have access to appropriate computing facilities. It should be emphasized that these courses represent college-level achievement for which most colleges and universities can be expected to grant advanced placement and credit. Placement and credit are granted by institutions in accordance with their own policies, not by those of the College Board or the AP Program.

The Courses

The AP Computer Science courses are introductory courses in computer science. Because the development of computer programs to solve problems is a skill fundamental to the study of computer science, a large part of the course is built around the development of computer programs or parts of programs that correctly solve a given problem. The course also emphasizes the design issues that make programs understandable, adaptable, and, when appropriate, reusable. At the same time, the development of useful computer programs and classes is used as a context for introducing other important concepts in computer science, including the development and analysis of algorithms, the development and use of fundamental data structures, and the study of standard algorithms and typical applications. In addition, an understanding of the basic hardware and software components of computer systems and the responsible use of these systems are integral parts of the course. The topic outline on pages 8–13 summarizes the content of the AP Computer Science curriculum.

Goals

The goals of an AP course in computer science are comparable to those in the introductory sequence of courses for computer science majors offered in college and university computer science departments. It is not expected, however, that all students in an AP Computer Science course will major in computer science at the university level. An AP Computer Science course is intended to serve both as an introductory course for computer science majors and as a course for people who will major in other disciplines that require significant involvement with technology. It is not a substitute for the usual college-preparatory mathematics courses.

The following goals apply to both of the AP Computer Science courses when interpreted within the context of the specific course.

- Students should be able to design and implement computer-based solutions to problems in a variety of application areas.
- Students should be able to use and implement well-known algorithms and data structures.
- Students should be able to develop and select appropriate algorithms and data structures to solve problems.
- Students should be able to code fluently in an object-oriented paradigm using the programming language Java. Students are expected to be familiar with and be able to use standard Java library classes from the AP Java subset.
- Students should be able to read and understand a large program consisting of several classes and interacting objects. Students should be able to read and understand a description of the design and development process leading to such a program. (An example of such a program is the *AP Marine Biology Simulation Case Study.*)
- Students should be able to identify the major hardware and software components of a computer system, their relationship to one another, and the roles of these components within the system.
- Students should be able to recognize the ethical and social implications of computer use.

Computer Language

The content of the college-level introductory programming course has evolved significantly over the years. Starting as a treatment merely of language features, it eventually incorporated first the notions of procedures and procedural abstraction, then the use of modules and data abstraction. At most institutions, the current introductory programming course takes an object-oriented approach to programming that is based on encapsulating procedures and data and creating programs with interacting objects. The AP Computer Science courses have evolved to incorporate this approach.

Current offerings of the AP Computer Science Examination require the use of Java. Those sections of the exam that require the reading or writing of actual programs will use Java. The exam will not cover all the features of Java; it will be consistent with the AP Java subset. (See Appendix A.) The AP Java subset can be found in the Computer Science section of AP Central™ (apcentral.collegeboard.com).

Equipment

Students should have access to a computer system that represents relatively recent technology. The system should be able to compile, in a matter of seconds, programs of size comparable to the *AP Marine Biology Simulation Case Study*, and response time should be reasonably rapid. This will require large hard disk drives either on individual machines or shared via a network.

Each student in the course should have a minimum of three hours per week alone on a computer throughout the academic year; additional time is desirable. This access can be made available at any time during the school day or after school and need not be made available to all students in the AP course simultaneously. It should be stressed that (1) this requirement represents a bare minimum of access; and (2) this time is not instructional time at a computer with the teacher or a tutor, but is time that the student spends alone at a computer in addition to the instructional time. Schools that do not allow their facilities to be used after school hours may wish to reevaluate such a policy in light of the needs of their students who take an AP Computer Science course.

Schools offering AP Computer Science will need to have Java software and enough memory in their lab machines so that students will be able to compile and run Java programs efficiently. Both free and commercial Java systems are available from a variety of sources. At a minimum, the hardware configuration will need large hard drives and sufficient memory to support current operating systems and compilers.

Prerequisites

The necessary prerequisites for entering either of the AP Computer Science courses include knowledge of mathematics at the level of a second course in algebra and experience in problem solving. A student in either AP Computer Science course should be comfortable with functions and the concepts found in the uses of functional notation, such as `f(x) = x + 2` and `f(x) = g(h(x))`. It is important that students and their advisers understand that any significant computer science course builds upon a foundation of mathematical reasoning that should be acquired before attempting such a course.

Schools that offer Computer Science AB may have students who do well on the topics covering programming methodology but do not deal as well with the more advanced material on data structures and analysis of algorithms. Such students might be advised to concentrate instead on the topics listed for Computer Science A and to take the Computer Science A examination.

Some schools may offer only Computer Science A and encourage students who move at a faster pace to study the topics covered by the Computer Science AB outline. They can then take the AP Examination in Computer Science AB rather than the examination in Computer Science A. Other schools may offer both courses and restrict enrollment in Computer Science AB to students who have some prior programming experience. Some schools may offer each course for a full year, while others cover each in one semester. This will vary according to the background of the students and the teacher.

One prerequisite for an AP Computer Science course, competence in written communication, deserves special attention. Documentation plays a central role in the programming methodology that forms the heart of an AP Computer Science course. Students should have already acquired facility in written communication before entering such a course.

Teaching the Courses

The teacher should be prepared to present a college-level first course in computer science. Each AP Computer Science course is more than a course on programming. The emphasis in these courses is on procedural and data abstraction, object-oriented programming and design methodology, algorithms, and data structures.

Because of the dynamic nature of the computer science field, AP Computer Science teachers will continually need to update their skills. Some resources that may assist teachers in professional development are AP Computer Science workshops and Summer Institutes, and Web sites such as AP Central. For information on workshops, teachers should contact their regional College Board office or go to AP Central.

One particular area of change is the evolution of programming languages and programming paradigms. Teachers should endeavor to keep current in this area by investigating different programming languages.

Topic Outline

Following is an outline of the major topics covered by the AP Examinations in Computer Science. This outline is intended to define the scope of the course, but not necessarily the sequence. The topics in the right-hand column will not be tested on the Computer Science A examination.

I. Object-Oriented Program Design

The overall goal for designing a piece of software (a computer program) is to correctly solve the given problem. At the same time, this goal should encompass specifying and designing a program that is understandable, can be adapted to changing circumstances, and has the potential to be reused in whole or in part. The design process needs to be based on a thorough understanding of the problem to be solved.

Computer Science A and AB	***Computer Science AB only***
A. Program design	
1. Read and understand a problem description, purpose, and goals.	1. Specify the purpose and goals for a problem.
2. Apply data abstraction and encapsulation.	
3. Read and understand class specifications and relationships among the classes (“is-a,” ”has-a” relationships).	3. Decompose a problem into classes, define relationships and responsibilities of those classes.
4. Understand and implement a given class hierarchy.	
5. Identify reusable components from existing code using classes and class libraries.	

Computer Science A and AB	*Computer Science AB only*
B. Class design	
1. Design and implement a class.	1. Design and implement a set of interacting classes.
2. Design an interface.	
3. Choose appropriate data representation and algorithms.	3. Choose appropriate advanced data structures and algorithms.
4. Apply functional decomposition.	
5. Extend a given class using inheritance.	

II. Program Implementation

The overall goals of program implementation parallel those of program design. Classes that fill common needs should be built so that they can be reused easily in other programs. Object-oriented design is an important part of program implementation.

Computer Science A and AB	*Computer Science AB only*
A. Implementation techniques	
1. Methodology	
a. Object-oriented development	
b. Top-down development	
c. Encapsulation and information hiding	
d. Procedural abstraction	
B. Programming constructs	
1. Primitive types vs. objects	
2. Declaration	
a. Constant declarations	
b. Variable declarations	
c. Class declarations	
d. Interface declarations	
e. Method declarations	
f. Parameter declarations	

Computer Science A and AB	***Computer Science AB only***
3. Console output (System.out.print/println)	
4. Control	
a. Methods	
b. Sequential	
c. Conditional	
d. Iteration	
e. Recursion	
C. Java library classes (included in the A level AP Java Subset)	C. Java library classes (included in the AB level AP Java Subset)

III. Program Analysis

The analysis of programs includes examining and testing programs to determine whether they correctly meet their specifications. It also includes the analysis of programs or algorithms in order to understand their time and space requirements when applied to different data sets.

Computer Science A and AB	***Computer Science AB only***
A. Testing	
1. Test classes and libraries in isolation	
2. Identify boundary cases and generate appropriate test data	
3. Perform integration testing	
B. Debugging	
1. Categorize errors: compile-time, run-time, logic	
2. Identify and correct errors	
3. Techniques: use a debugger, add extra output statements, hand-trace code	
C. Understand and modify existing code	
D. Extend existing code using inheritance	
E. Understand error handling	
1. Understand runtime exceptions	
	2. Throw runtime exceptions

Computer Science A and AB | *Computer Science AB only*

F. Reason about programs
   1. Pre- and post-conditions
   2. Assertions

   (AB only) 3. Invariants

G. Analysis of algorithms
   1. Informal comparisons of running times
   2. Exact calculation of statement execution counts

   (AB only) 3. Big-Oh notation
   (AB only) 4. Worst-case and average-case time and space analysis

H. Numerical representations and limits
   1. Representations of numbers in different bases
   2. Limitations of finite representations (e.g., integer bounds, imprecision of floating-point representations, and round-off error)

IV. Standard Data Structures

Data structures are used to represent information within a program. Abstraction is an important theme in the development and application of data structures.

Computer Science A and AB	*Computer Science AB only*
A. Simple data types (int, boolean, double)	
B. Classes	
C. One-dimensional arrays	
	D. Two-dimensional arrays
	E. Linked lists (singly, doubly, circular)
	F. Stacks
	G. Queues
	H. Trees
	I. Heaps
	J. Priority queues
	K. Sets
	L. Maps

V. Standard Algorithms

Standard algorithms serve as examples of good solutions to standard problems. Many are intertwined with standard data structures. These algorithms provide examples for analysis of program efficiency.

Computer Science A and AB

A. Operations on A-level data structures previously listed
   1. Traversals
   2. Insertions
   3. Deletions

B. Searching
   1. Sequential
   2. Binary

C. Sorting
   1. Selection
   2. Insertion
   3. Mergesort

Computer Science AB only

A. Operations on AB-level data structures previously listed
   1. Traversals
   2. Insertions
   3. Deletions
   4. Iterators

   3. Hashing

   4. Quicksort
   5. Heapsort

VI. Computing in Context

A working knowledge of the major hardware and software components of computer systems is necessary for the study of computer science, as is the awareness of the ethical and social implications of computing systems. These topics need not be covered in detail, but should be considered throughout the course.

Computer Science A and AB	***Computer Science AB only***
A. Major hardware components	
1. Primary and secondary memory	
2. Processors	
3. Peripherals	
B. System software	
1. Language translators/compilers	
2. Virtual machines	
3. Operating systems	
C. Types of systems	
1. Single-user systems	
2. Networks	
D. Responsible use of computer systems	
1. System reliability	
2. Privacy	
3. Legal issues and intellectual property	
4. Social and ethical ramifications of computer use	

Commentary on the Topic Outline

The AP Computer Science (APCS) course is an introductory course in computer science. Because the design and implementation of computer programs to solve problems are skills that are fundamental to the study of computer science, a large part of the APCS course is built around the development of computer programs that correctly solve a given problem. These programs should be understandable, adaptable, and, when appropriate, reusable. At the same time, the design and implementation of computer programs is used as a context for introducing other important aspects of computer science, including the development and analysis of algorithms, the development and use of fundamental data structures, the study of standard algorithms and typical applications, and the use of logic and formal methods. In addition, an understanding of the basic hardware and software components of computer systems and the responsible use of these systems are integral parts of the course. The topic outline summarizes the content of the APCS curriculum. In this section, we provide more details about the topics in the outline.

I. Object-Oriented Program Design

Computer Science involves the study of complex systems. Computer software is a part of a complex system. To understand the development of computer software, we need tools that can make sense of that complexity. Object-oriented design and programming form an approach that enables us to do that, based on the idea that a piece of software, just like a computer itself, is composed of many interacting parts.

The novice will not start by designing a whole program, but rather by studying programs already developed, then writing or modifying parts of a program to add to or change its functionality. Only later in the first course will a student get to the point of working from a specification to develop a design for a program or part of a program.

In an object-oriented approach, the fundamental part of a program is an object, an entity that has state (stores some data) and operations that access or change its state and that may interact with other objects. Objects are defined by classes; a class specifies the components and operations of an object and each object is an instance of a class.

A. Program Design

A student in a first computer science course (APCS A) would learn to work with the design of a program, but would not be expected to develop a full program design. However, this student should be able to work from a given design to develop the parts of the program. This would include an understanding of how to apply the data abstractions covered in the first course (classes and one-dimensional arrays).

A student in the first course should be able to understand the relationships among the different classes that comprise a program. One such relationship is an inheritance hierarchy, where a subclass inherits characteristics from a superclass, thereby creating an "is-a" relationship. For example if we have a class `Elevator` with a superclass `PersonalTransport` and subclasses `ExpressElevator` and `FreightElevator`, as shown in Figure 1, then we could say that an `ExpressElevator` is-a(n) `Elevator`, an `Elevator` is-a `PersonalTransport`, and so on. If it does not make sense to express the natural relationship in terms of A is-a B, then it is not correct to use inheritance to make A a subclass of B.

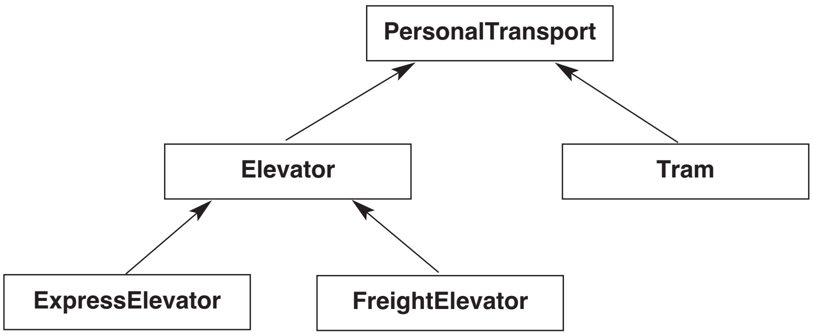


Figure 1

The other common relationship among classes is composition. One class has one or more instances of another class as attributes. For example, an `Elevator` would have `ElevatorDoors`. `ElevatorDoors` would be a separate class, and the `Elevator` class would have one or more instances of it among its attributes. This is an example of a "has-a" relationship: an `Elevator` has-a(n) `ElevatorDoor`. It would not make sense to say an elevator is an elevator door, so to try to implement an `Elevator` class by inheriting from an `ElevatorDoor` class would be incorrect, even if it might be technically possible.

Students in the first course should be able to distinguish between these different relationships among classes. They should also be able to implement a class inheritance hierarchy when given the specifications for the classes involved — which classes are subclasses of other classes.

Students in the second course (APCS AB) should learn to work from a given problem statement to define the purpose and goals of a program intended to solve that problem. They then should be able to decompose the problem into interacting objects and specify the classes needed to define those objects, as well as the relationships among those classes.

An important skill when working with computer programs is to be able to recognize the appropriate use of components from libraries. The APCS curriculum specifies the classes from the Java libraries with which students should be familiar, and students should be able to recognize the appropriate use of these classes. In addition, students should recognize the possibilities of reuse of components of their own code or other examples of code, such as the *AP Marine Biology Simulation Case Study*, in different programs.

B. Class Design

A fundamental part of the development of an object-oriented program is the design of a class. Students in the first course should be able to design a class — write the class declaration including the instance variables and the method signatures (the method bodies would comprise the implementation of this design) — when they are given a description of the type of entity the class represents. Such a description would include the data that must be represented by the class and the operations that can be applied to that data. These operations range from simple access to the data or information that can be derived from the data, to operations that change the data (which stores the state) of an instance of the class. The design of a class includes decisions on appropriate data structures for storing data and algorithms for operations on that data. The decomposition of operations into subsidiary operations, functional decomposition, is part of the design process. An example of the process of designing a class is given in the sample free-response question which documents the logical considerations for designing a savings account class.

A student in the second course (AB) should be able to develop a design for a set of interacting classes, given the specifications for those classes and their relationships. This student will also have a much broader set of data structures and algorithms to use in design decisions.

Given a design for a class, either their own or one provided, students in the first course should then be able to implement the class (more details on program implementation follow). They should also be able to extend a given class using inheritance, thereby creating a subclass with modified or additional functionality.

An *interface* is a specification for a set of operations that a class must implement. In Java, there is a specific construct, the `interface`, that can be specified for this purpose, so that another class can be specified to *implement* that interface. Students in both the A and AB courses should be able to design an interface by declaring all its methods, given a specification of the operations that these methods represent.

Design as an Examination Topic

As noted in the topic outline, the A examination may include questions that ask about the design as well as the implementation of classes or a simple hierarchy of classes. The AB examination may include questions that ask about the design of multiple classes that specify interacting objects, as well as the implementation of such classes.

A design question would provide students with a description of the type of information and operations on that information that an object should encapsulate. Students would then be required to provide part or all of an interface or class declaration to define such objects. An example of this type of question appears as one of the sample free-response questions for Computer Science A (see page 72).

A design question may require a student to develop a solution that includes the following:

- appropriate use of inheritance from another class using keyword `extends`
- appropriate implementation of an interface using keyword `implements`
- declaration of constructors and methods with
  - meaningful names
  - appropriate parameters
  - appropriate return types
- appropriate data representation
- appropriate designation of data and methods as `public` or `private`
- all data should be `private`
- all client accessible operations should be specified as `public` methods

A design question might only require that a student specify the appropriate constructor and method signatures (access specifier, return type, method identifier, parameter list), and not require that the body of the constructors or methods be implemented. A question focusing on a simple class hierarchy might also require implementation of the body of some or all methods for some of the classes.

II. Program Implementation

In order to implement a program, one must understand the fundamental programming constructs of the language, as well as understand the design of the program. The fundamental principles of encapsulation and information hiding should be applied when implementing classes and data structures. A good program will often have components that can be used in other programs.

There are topics not included in the course outline that will be part of any introductory course. For example, input and output must be part of a course on computer programming. However, in a modern object-oriented approach to programming, there are many ways to handle input and output, including console based character I/O, graphical user interfaces and applets. Consequently, the APCS curriculum does not prescribe any particular approach and will not test the details of input and output (except for the basic console output, `System.out.print/ln` in Java), so that teachers may use an approach that fits their own style and whatever textbook and other materials they use.

A. Implementation Techniques

A variety of implementation techniques are used for organizing the code as you develop classes and methods to implement the design. Object-oriented development starts with the design process that breaks down a problem into its constituent parts that are then represented by objects (that are defined by class declarations). In the implementation phase, the operations on those objects are the methods that must be implemented for each class. In addition, during the development of code, we might discover additional classes that can be useful. These might be classes found in a Java library, or they might be auxiliary classes that we develop.

For example, in the sample free-response question on instant runoff voting, we might have first designed this program using two classes, one responsible for all the voter's ballots, `VoterBallots`, and a second responsible for the overall election, `InstantRunoff.` During the implementation of this program, we realize that we need to have a class to represent each individual voter's ballot. In the example as given in the question, we chose to define our own new class, `Ballot`, to take on this responsibility. Another choice would have been to use the class

`ArrayList` from the Java library (see appendices). It is often the case that we discover the need for additional classes during the implementation phase of programming. This is a form of top-down programming when we develop subsidiary classes to make the representation of each individual class simpler. The process of organizing some information and the operations on that information into one unit, a class declaration, is called *encapsulation*. The technique of keeping the data representation hidden from the client by specifying it `private` is called *information hiding*.

Another form of top-down development occurs as we define the methods for classes. If the code for a method is long and complex, we often break out coherent parts of that code into subsidiary helper methods. Such methods are normally declared `private`, since they are not part of the client interface for a class. Another reason for encapsulating a piece of code into a helper method is that it is used in more than one place. Rather than repeat code, it is better to abstract it into a single method and call that method. This process of abstracting pieces of code into methods is called *procedural abstraction*.

The *AP Marine Biology Simulation Case Study* provides examples of the procedural abstraction that we have described. Rather than repeat code in each constructor for the `Fish` class, the private helper method `initialize` is used to set the values of the instance variables. In the `move` method in the `Fish` class top-down design has been used to break the process into cohesive units using the private helper methods `nextLocation`, `emptyNeighbors`, `changeLocation`, and `changeDirection.`

B. Programming Constructs

Programming constructs are the tools of the trade. One needs to understand the different programming constructs, variables, control structures, etc., in order to create a program. These are tied closely to the language used to teach the course. The different programming constructs are common to most languages that use a given paradigm for program design, usually differing only in syntax and some details. The abilities one gains by learning to implement programs in Java would carry over quite easily to any other object-oriented programming language such as Smalltalk, C#, Eiffel, or C++.

The basic constructs for storing information are variables that are either primitive types or objects. The primitive types included in the APCS curriculum include Boolean, integer, and real, represented in Java as types `boolean`, `int`, and `double.` Other Java types such as `float` (single precision floating point representation of real numbers) and `char` (characters) are not included in the testable subset, but may, of course be covered in an APCS course.

A class declaration defines a type and an instance of that class is called an object. In Java, an object variable is a reference to that object, so that when one object variable is assigned to another, they both refer to the same object — an example of aliasing. (Technically, an object variable contains the address where the object itself is stored in memory. The same effect can be obtained in C++ by using reference variables.)

A declaration assigns an identifier to a construct and defines that construct. A variable declaration indicates the type of the variable, which may be one of the primitive types or a reference to a class. A constant is declared in the same way, but may not change value once assigned; in Java a constant is indicated by the keyword `final.` *Class* declarations specify a new type of object, while *interface* declarations specify a type that encompasses only the specified methods. A class can *implement* an *interface* if it defines all the methods specified by the interface. An object of such a class is type-compatible with the interface.

The encapsulation of actions, procedural abstraction, is accomplished with *methods*. A method specifies some code to be executed when it is called. A method declaration must include the access specifier (`public` or `private`) [other options are not included in the testable subset], the return type (`void` if nothing is to be returned), the identifier for the method and the parameter list for the method. Most methods are instance methods that are called with a reference to an object of the given class followed by the method name, using the "." notation. Methods that are `static` can be called with a reference to the class name, as for example, the methods from the class `Math` such as `Math.sqrt.`

The parameter list contains the types and identifiers of the parameters needed for a method. Java has much simpler parameters than some languages, e.g., Pascal and C++. All parameters in Java are value parameters. However, this can be confusing, since this means different things for primitive and object types. For primitive types, the fact that parameters are always value parameters means that a copy of the value of the actual parameter passed is used within the method code. If a variable was used as the actual parameter for the method call (any expression that evaluates to the correct type could be used), then that variable is unchanged when the call is completed.

For an object, the fact that parameters are always value parameters means that a reference to an object is passed by value to the method. (This is <u>not</u> the same as passing a parameter by reference.) This means that the variable that was used as the actual parameter must be unchanged — it still refers to the same object. However, within the method, the reference to that object can be used to call its methods and these may change the internal state of that object. Thus, although the actual parameter is unchanged and refers to the same object, that object may have its internal state changed.

The APCS curriculum does not prescribe any specific method of input and output that should be used in teaching an APCS course, except that students should know how to use the simple Java console output `System.out.print/ln` that writes output to the console. Of course, any introductory Computer Science course will include input and output, but the means for doing this can be quite varied. Since it really has little impact on the fundamental issues of Computer Science that the course is about, the specific means used are left to the discretion of the teacher. One might use applets with graphical input and output, an application that uses a graphical user interface, such as the one supplied with the *AP Marine Biology Simulation Case Study* or textual input and output. One probably will use input from and output to text files as well. Teachers may well choose to use simplified libraries that come with a textbook or that can be found on the Internet for input and output, rather than the relatively low-level constructs that are in the standard Java libraries.

Program statements are executed sequentially unless that sequence is altered through the use of a control construct. The most common control construct is the method call. Objects that are instances of library classes, as well as objects that instantiate user defined classes, are manipulated using calls to the object methods. These methods may simply return information about the object (accessor methods) or may change the state of the object (modifier or mutator methods). When teaching the object-oriented design and programming paradigm, it is essential that the use of objects defined both by library classes and by user defined classes, and the transfer of control implied by method calls for those objects be taught right from the start.

Within sequences of code, there are two types of constructs that can change the normally sequential execution of statements, conditionals, and iteration. These control constructs are common to virtually all imperative programming languages. Conditional control most commonly takes the form of `if ... else` statements. Typical `if` constructs including a simple `if` with no `else` clause, `if ... else`, nested `if ... else` clauses and the common multi-part `if` with mutually exclusive conditions given in the form

```
if (...)
  {...}
else if (...)
  {   }
else if (...)
  {...}
...
else
  {...}
```

Java also includes a `switch` statement. The syntax of the `switch` statement is rather primitive and it is not included in the AP Java subset, although teachers are free to include it in their courses if they so desire. It adds no essential logic to the programming tools, however.

Iteration is accomplished by loop constructs that include the `for` loop, the `while` loop, and the `do ... while` loop (the `do ... while` loop is not in the AP Java subset). The syntax of these loops in Java is identical to their syntax in C++, and the logic behind them is found in most programming languages.

Another common means of getting repetitive behavior from a program is the use of recursion. A recursive method is one that may call itself with different parameters or that through a sequence of method calls eventually causes another call to itself. Of course, a fundamental issue for recursion is that there must be an end to the calls back to the same method in order to avoid an infinite sequence of recursive calls. Consequently, any recursive method must include a "base case," a case when no further recursive calls are made. The base case is commonly determined by a conditional construct that makes no recursive call based on some condition among the parameters, such as when a numeric parameter reaches a certain value. A valid recursion must always have a base case and must always have logic that makes successive recursive calls progress toward the base case in some fashion. In an object-oriented design, the base case can be realized without the explicit appearance of a conditional expression in the recursive method. Using polymorphism of methods, where the method actually called is determined by a subclass of a given class or interface, we may find a sequence of calls to objects that have a recursive version of the method, but that eventually call the method on an object of a different subtype that has a non-recursive version of the method. Usually this situation is created by using a conditional when the objects are constructed.

C. Java Library Classes

An important aspect of modern programming is the existence of extensive libraries that supply many common classes and methods. One part of learning the skill of programming is to learn about available libraries and their appropriate use. For the AP Computer Science course, specific parts of the standard Java libraries are required for both the A course and the AB course. These are specified in the AP Java subset for the A and AB courses (see appendices).

III. Program Analysis

We need to be able to analyze programs both from the point of view of correctness and also to understand their efficiency for solving problems. An important part of program analysis is the testing of programs and parts of programs. Unit testing refers to tests for small parts of programs such as individual methods or a single class that might have a number of methods. Integration testing is the testing of larger units of a program that are composed of several smaller units: several classes involving a large number of methods. At the high end, integration testing refers to the testing of a complete program or software system.

In order to do useful testing we cannot just randomly run programs or pieces of programs with arbitrary data. Data for testing must be selected to reflect a range of typical cases, including the different variations in the data that can occur, as well as boundary cases, cases that are at the extremes of valid data, and erroneous cases, where the code should fail, but with appropriate error messages or exceptions. The *AP Marine Biology Simulation Case Study* has many examples of testing and a good discussion of the selection of appropriate test data.

"Debugging" refers to the discovery and correction of errors in a program. These errors can be discovered through testing or through careful analysis. There are three categories of errors — compile-time, run-time, and logic errors. Compile-time errors are discovered by the compiler and include errors in the correct formation of expressions and statements in the programming language, i.e., syntax errors. Another type of compile time error is a mismatch of actual with formal parameters in method calls. Some compilers also pick up errors in logic such as uninitialized variables.

A run-time error is an error that occurs when the program is running and usually causes it to terminate abnormally, that is to stop running, but may also result in a program-controlled error condition. Runtime errors are often discovered during testing. The following are some typical run-time errors.

- an arithmetic error such as division by zero
- an out-of-bounds array index
- an attempt to cast an object to a type that does not apply to that object
- an attempt to access methods for a null object variable

Because many runtime errors are dependent on the data used, the construction of a good set of test data is important for discovering these errors. Some runtime errors may cause the program to "crash" (terminate abnormally) or to enter an infinite loop or infinite recursion. These types of errors are particularly difficult to discover.

A logic error is a flaw in the logic of a program. The program runs and produces output, but because the logic of the computation is faulty, the output or actions of the program are sometimes incorrect. A simple example would be a method that is intended to return the maximum from a set of values, but because of an incorrect loop condition does not check the last value and therefore sometimes fails. Testing with data for which the correct result is already known is an important technique for uncovering logic errors.

Some of the techniques for finding and correcting errors, for "debugging" a program or segment of a program, include hand-tracing code, adding extra output statements to trace the execution of a program, or using a debugger to provide information about the program as it runs and when it crashes. Students should be encouraged to experiment with available debugging facilities. However, these will not be tested since they vary from system to system.

Students should be able to read and modify code for a program. They should also be able to extend existing code by taking a given class declaration and declaring a new class using inheritance to add or change the given class' functionality. The *AP Marine Biology Simulation Case Study* contains examples of using inheritance to create new classes.

A common way of handling error conditions that are anticipated in programs is to use exceptions. An exception signals an error in a program. Students in the APCS A course should understand runtime exceptions — exceptions that need not be "caught" and that cause the program to stop. Here are some common runtime exceptions in Java:

- `NullPointerException`, indicating an attempt to reference an object using an object variable that is null
- `IllegalArgumentException`, indicating an argument to a method that is not legal for that method
- `ArrayIndexOutOfBoundsException`, indicating an attempt to access an element that is not within an array's range of indexes
- `ArithmeticException`, such as division by zero for integers
- `ClassCastException`, which occurs when an attempt is made to cast a variable to a class that it does not match

Students in the APCS AB course should also be able to write code to throw runtime exceptions under appropriate circumstances, such as an `IllegalArgumentException.`

Formal methods enable us to reason about programs and verify that they are correct by proof, rather than by tracing and testing. When possible, formal program verification is preferred and becomes essential for life-critical software. Testing can only prove the existence of bugs, it can never prove there are no bugs in software because for any useful program there are too many possible execution paths to test.

In the AP Computer Science curriculum, we introduce some basic ideas of formal methods. One of these is the use of preconditions and postconditions for methods. A precondition is an assertion that should be true when the method is called. A postcondition is an assertion that will be true when the method completes execution, if the precondition was true when it was called. Preconditions and postconditions form a contract that the method should fulfill.

An assertion is a logical statement that may be true or false. In a computer program an assertion made at a certain point in a program expresses a logical condition that should be true at that point if the program is working correctly. For example, the precondition for a method is an assertion that should be true whenever that method begins execution.

A loop invariant is an assertion that should be true every time a loop condition is checked to determine whether another iteration of the loop will be executed. Students in the APCS AB course should understand the use of loop invariants for showing that loops do the intended calculation. By combining a loop invariant with the exit condition for a loop, it is possible to prove what the loop does. By using loop invariants one can often prove that a method that uses these loops satisfies the precondition and postcondition contract: if the precondition is true, then the postcondition is true. This is a small step toward formal verification of programs that students are likely to see in more advanced courses.

An important part of analyzing programs is the analysis of the efficiency for a program, both in terms of the time needed for the program to execute for a given data set and in terms of the space (memory) needed. In the APCS A course, students should be able to make informal comparisons of running times of different pieces of code, for example by counting the number of loop iterations needed for a computation. In the APCS AB course students learn about asymptotic analysis of algorithms: how the algorithms behave as the data sets get larger and larger. Asymptotic analysis uses the "Big-Oh" notation to derive a bound for an algorithm's running time in terms of standard functions such as n, n^2, $log(n)$, etc. Students in the AB course should understand asymptotic analysis of running times for the worst case and average case (when it can be reasonably defined) for standard searching and sorting algorithms. Students should also be able to analyze a given algorithm of moderate complexity. In addition, these students should be able to make a similar analysis of the space (memory) needed to carry out a given algorithm.

Many programs involve numerical computations and therefore are limited by the finite representations of numbers in a computer. Students should understand the representation of positive integers in different bases, particularly decimal, binary, hexadecimal, and octal. They should also understand the consequences of the finite representations of integer and real numbers, including the limits on the magnitude of numbers represented, the imprecision of floating point computation, and round-off error.

IV. Standard Data Structures

There are a number of standard data structures used in programming. Students should understand these data structures and their appropriate use. For the AP Computer Science A and AB courses, students need to be able to use the standard representations of integers, real numbers and Boolean (logical) variables. In Java these are represented as follows:

- `int`, a 32-bit signed integer representation;
- `double`, a 64-bit floating point representation of real numbers;
- `boolean`, a Boolean value that may be true or false

The other primitive types in Java, `char` and `float,` are not part of the AP Java subset, but may be useful in an APCS course.

Classes enable us to define new types that encapsulate both data and operations (methods). A class may be used to declare a simple container for related data (a record) with the associated accessor methods. A simple example from the *AP Marine Biology Simulation Case Study* is the class `Location.` `Location` encapsulates two integer values representing the coordinates of a position in a two-dimensional grid. These coordinates are then accessed by the methods `row()` and `col()`, as well as a `toString()` method that returns a string representation of the `Location`.

A class may also be used to define a more complex object that contains information (its state) and has complex behavior defined by its methods that can change that state and interact with other objects. An example of such a class is the `Fish` class from the *AP Marine Biology Simulation Case Study*. An instance of `Fish` has state that includes its `Color` and `Location`, and has behaviors defined by the act method and subsidiary methods that can change that state. Indeed, an object-oriented program is built from such interacting classes. Students in both the APCS A and AB courses must be able to work with the data structures defined by classes.

Students are responsible for understanding the Java `String` class. Instances of the `String` class represent strings of characters. Although strings are objects, the binary operator + can be applied to strings, and returns the concatenation of the two string arguments. Other methods that apply to the `String` class are given in the AP Java subset (see appendices).

There are several data structures that collect homogeneous items into a list. The most fundamental is a one-dimensional array. Students in the APCS A course should be comfortable working with one-dimensional arrays and should be familiar with two representations for them. The first is a built-in Java array, declared as follows:

```
int[] intArray = new int[10];
```

The above declaration will create a new array of ten integers and assign it to the variable `intArray`. Any type of array, including a user defined class, could be declared. For example, the following statement declares an array of `numFish` `Fish` (from the *AP Marine Biology Simulation Case Study*)

```
Fish[] fishList = new Fish[numFish];
```

Once created, a Java array has fixed size (given by the public constant for the object, `length`). In this regard a Java array is like a primitive array in other languages such as Pascal or C/C++. However, a Java array is an object, which means an array variable is a *reference* to the array. In general, a Java array is appropriate when the list of items is fixed in size.

Java has a second data structure that represents a one-dimensional array, the class `ArrayList` from the `java.util` package (library). An `ArrayList` is a structure that can be accessed by index, as can an array. However, an `ArrayList` is dynamic, its size can change. In fact, the method `add` with just an object parameter will append the given object to the `ArrayList`, adjusting its size as needed. An `ArrayList` stores references to instances of an `Object` — that is, any object. It cannot be given a specific type. Consequently, when an item is accessed from an `ArrayList` it must usually be downcast to the specific type that is needed to access its methods. An `ArrayList` also does not use the bracket notation, but instead is accessed using the `get`, `set` and `add` methods.

Students in both the APCS A and AB courses should be familiar with both Java arrays and the structure `ArrayList.` They should be able to use either in a program and should be able to select the most appropriate one for a given application. The methods for `ArrayList` for which students are responsible are specified in the AP Java subset (see appendices).

The APCS AB course has a major focus on abstract data types and data structures. Consequently, there are several additional data structures that these students must understand. The following abstract data types should be covered in the APCS AB course:

- Lists
- Stacks
- Queues
- Priority Queues
- Sets
- Maps

There are a number of data structures that can be used to implement these abstractions. They include, of course, the one-dimensional arrays that are part of the APCS A course. Other fundamental data structures are the following:

- two-dimensional arrays;
- linked lists, including singly, doubly, and circular;
- trees, including binary trees;
- heaps (and their standard array implementation)
- hash tables

Students should be able to implement the abstract data types with appropriate data structures.

In addition, many implementations of abstract data types are given as part of the Java libraries. Those that students are required to understand are specified in the AP Java subsets for the A and AB courses. These include `ArrayList` for the A course and the following for the AB course:

- `List` interface
- `LinkedList`
- `Set` interface
- `HashSet`
- `TreeSet`
- `Map` interface
- `HashMap`
- `TreeMap`
- `Iterator` interface
- `ListIterator` interface

These are all found in the `java.util` package.

The Java library structures `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`, `HashMap`, and `TreeMap` which implement the data types `List`, `Set`, and `Map`, all have specifications about their runtime in the standard documentation. For example, the documentation specifies that the `get` and `set` operations for an `ArrayList` can be done in constant time. On the other hand, the documentation indicates that the operations for `LinkedList` take time for a doubly linked list implementation, so a `set` operation would take `O(n)` for a list of `n` items in the worst case. Students should understand these time estimates so that they can estimate the asymptotic time for operations involving these data structures.

The following is a list of some useful information that can be found in the Java documentation:

1. Adding an item to the end of an `ArrayList` takes `O(1)` time. Adding one item to the front of an `ArrayList` of size `n` is `O(n)`.
2. `LinkedList` is implemented as a doubly linked list with head and tail links. Searches for the kth item in the list start at the end of the list that is closer.
3. `TreeSets` and `TreeMaps` are implemented as balanced binary trees, so `add`, `contains`, and `remove` for `TreeSet` and `put`, `get`, and `containsKey` for `TreeMap` are all `O(log n)` worst-case.
4. Operations on `HashSet` and `HashMap` are `O(1)` expected time, but could be `O(n)` worst-case time
5. `Iterator` and `ListIterator` for `List` return the elements in the order they appear in the list.
6. Items inserted in `TreeSet` and keys inserted in `TreeMap` must be `Comparable` (in the AP Java subset).
7. `Iterator` for `TreeSet` returns the elements in the order specified by `compareTo.`
8. `Iterator` for `HashSet` returns elements in an arbitrary order.
9. An `Iterator` or `ListIterator` can iterate through all elements in a collection in `O(n)` time. For a `HashSet`, `n` is the maximum size the `HashSet` has ever been; for other Collection classes, `n` is the current size. For a `TreeSet` iterator, the first call to `next()` takes `O(log n)` time.

See the AP Java subset for details, including the methods that students are required to know. Note there are three interfaces that will be used to ask questions about the stack, queue, and priority queue data types called `Stack`, `Queue`, and `PriorityQueue`, respectively. These are supplied for the APCS course so as to have a consistent set of methods for testing these concepts. AB students should know that some classes implement some interfaces (e.g., `LinkedList implements List`).

V. Standard Algorithms

Both the APCS A and AB courses cover several standard algorithms. These serve as good solutions to standard problems. These algorithms, many of which are intertwined with data structures, provide excellent examples for the analysis of program efficiency. Programs implementing standard algorithms also serve as good models for program design.

The A course includes standard algorithms for accessing arrays including traversing an array, inserting into and deleting from an array. Students should also know the two standard searches, sequential search and binary search, and the relative efficiency of each. Finally, there are three standard sorts that are required for the A course, the two most common quadratic sorts — Selection sort and Insertion sort — and the more efficient Merge sort. Of course, the latter implies that students should know the merge algorithm for sorted lists.

Students in the APCS A course are not required to know the asymptotic (Big-Oh) analysis of these algorithms, but they should understand that merge sort is advantageous for large data sets and should be familiar with the differences between selection and insertion sort.

An important part of the APCS AB course is the understanding and analysis of algorithms associated with the standard data structures. These include traversing the structures so as to access all elements, and adding and removing elements from the structures. Iterators are an abstraction of the process of traversing a structure. Java has both `Iterator` and `ListIterator` interfaces that provide this abstraction, and students should be familiar with their use.

In addition to the searching and sorting algorithms listed for the APCS A course, students in the AB course should also understand Quicksort and Heapsort. For all these standard algorithms, students in the AB course should understand the asymptotic complexity.

A hash table provides a structure for which each insertion and search operation can be carried out in constant time. Students in the AB course should understand hash tables, as well as being able to use the Java library implementations `HashSet` and `HashMap.`

VI. Computing in Context

Students should learn about the main hardware components of a computer *system*. A detailed study of hardware and how it interfaces with software goes beyond the bound of APCS, but students should be familiar with *processors*, the hierarchy of *primary and secondary memory*, and the *peripheral devices* that are used for communications with a computer (keyboard, mouse, screen, printer), for communications among computers (network connections, modems), and for off-line data storage and retrieval (floppy disk, CD). Students will normally become familiar with these through the use of their own systems for programming and other applications.

The APCS course should include discussion on how the main *software* components of a system enable the user to interact with the computer. Programs in a high level language, such as the one used for the APCS course, must be translated into machine executable code. These language translators can be *interpreters*, which translate and execute code one line at a time, or *compilers*, which translate whole programs or program components into a separate file of machine-executable code. The latter enables separate compilation (separately compiling components of a larger program), and also facilitates optimization of code. Separate compilation makes it possible to build libraries of precompiled modules that can be reused in many programs.

Java is a hybrid relative to language translation. It was developed to make transmitting programs between computers easier and to make it possible to run the same code on different types of computers. When we compile a Java class — it is always compiled one class at a time — we produce a class file that contains "Java byte code." This Java byte code is machine independent, so to run it we must have a "Java Virtual Machine" (JVM). The JVM interprets the code on whatever machine is being used. There are JVMs for many machines so that the same Java byte code can be run on all these machines. Some JVMs go one step further and incorporate some compiler technology into their translation process so as to speed up execution, but these details need not concern us here.

Operating Systems mediate between the computer user and the hardware. An operating system provides the code that (1) enables the user to interact with the computer through various peripheral devices, (2) manages the file system for the storage of programs and data files, and (3) manages the interaction of the user with various application programs, including the programming environment, compiler, JVM, and their own programs.

Students should be familiar with the basic computer systems that can be used. These include *single-user* or *stand-alone systems* (such as a personal computer, or a larger special-purpose machine with no connections to other machines) and *computer networks*, where many machines are interconnected and certain machines may provide different services (such as file storage, computational resources, or printing) to other machines on the network. Of course many computers are connected to the Internet, so students should be familiar with the benefits and pitfalls of working on the Internet.

Given the tremendous impact computers and computing have on almost every aspect of society, it is important that intelligent and responsible attitudes about the use of computers be developed as early as possible. The applications of computing that are studied in an APCS course provide opportunities to discuss the impact of computing. Typical issues include:

- the impact of applications using databases, particularly over the Internet, on an individual's right to privacy,
- the economic and legal impact of viruses and other malicious attacks on computer systems,
- the need for fault-tolerant and highly reliable systems for life-critical applications and the resulting need for software engineering standards,
- the intellectual property rights of writers, musicians, computer programmers, and fair use of intellectual property

Attitudes are acquired, not taught. Hence, references to responsible use of computer systems should be integrated into an APCS course wherever appropriate, rather than taught as a separate unit. Participating in an APCS course provides an opportunity to discuss such issues as the responsible use of a system and respect for the rights and property of others. Students should learn to take responsibility for the programs they write and for the consequences of the use of their programs.

Case Studies

A case study is a document that includes the statement of a problem, one or more programs that solve the problem, and a written description of one expert's path from problem statement to solution program(s). The write-up describes the choices made for design and implementation and the justification for the choices that were made.

Case studies provide a vehicle for presenting many of the topics of the AP Computer Science courses. They provide examples of good style, programming language constructs, fundamental data structures, algorithms, and applications. Large programs give the student practice in the management of complexity and motivate the use of certain programming practices (including decomposition into classes, use of inheritance and interfaces, message passing between interacting objects, and selection of data structures tailored to the needs of the classes) in a much more complete way than do small programs.

Case studies also allow the teacher to show concretely the design and implementation decisions leading to the solution of a problem and thus to focus more effectively on those aspects of the programming process. This approach gives the student a model of the programming process as well as a model program. The use of case studies also gives the student a context for seeing the importance of good design when a program is to be modified.

The AP Computer Science Examinations will include questions based on the case study described in the document *AP Marine Biology Simulation Case Study*. These questions may explore design choices, alternative choices of data structures, extending a class via inheritance, etc., in the context of a large program without requiring large amounts of reading during the exam. Both the A and AB examinations will contain at least five multiple-choice questions and one free-response question covering material from the case study. Printed excerpts from the case study programs will accompany the examinations.

Questions will deal with activities such as the following:

a. modifying the procedural and data organization of the case study program to correspond to changes in the program specification;
b. extending the case study program by writing new code (including new methods for existing classes, new subclasses extending existing classes, and new classes);
c. evaluating alternatives in the representation and design of objects and classes;
d. evaluating alternative incremental development strategies;
e. understanding how the objects/classes of the program interact; and
f. developing test data.

Sample questions for the *AP Marine Simulation Biology Case Study* appear in the teacher's manual. The text and code for the *AP Marine Biology Simulation Case Study* are available for downloading from AP Central.

The Examinations

The AP Examinations for Computer Science A and Computer Science AB are each three hours long and seek to determine how well students have mastered the concepts and techniques contained in the respective course outlines. Before the examination date, students must decide which of the two examinations they will take. In most cases, students will prepare during the year for one examination or the other. Some students enrolled in the AB course, however, may not feel comfortable with some of its more advanced topics. Such students might prefer to take the Computer Science A examination.

Each examination consists of two sections: a multiple-choice section (40 questions in 1 hour and 15 minutes), which tests proficiency in a wide variety of topics, and a free-response section (4 questions in 1 hour and 45 minutes), which requires the student to demonstrate the ability to solve problems involving more extended reasoning.

The multiple-choice and the free-response section of both AP Computer Science Examinations require students to demonstrate their ability to design, write, analyze, and document programs and subprograms.

Minor points of syntax are not tested on the examinations. All code given in the exams is consistent with the AP Java subset. All student responses involving code must be written in Java. Students are expected to be familiar with and able to use the standard Java classes listed in the AP Java subset. For both the multiple-choice and the free-response sections of the examinations, a quick reference to both the case study and the classes in the AP Java subset will be provided.

In the determination of the grade for each examination, the multiple-choice section and the free-response section are given equal weight. Because each examination is designed for full coverage of the subject matter, it is not expected that many students will be able to correctly answer all the questions in either the multiple-choice section or the free-response section.

Computer Science A: Sample Multiple-Choice Questions

Following is a representative set of questions. Questions marked with an asterisk are also representative of AB examination questions. The answer key for the Computer Science A multiple-choice questions is on page 59. In this section of the examination, as a correction for haphazard guessing, one-fourth of the number of questions answered incorrectly will be subtracted from the number of questions answered correctly. The AP Computer Science A Examination will include at least five multiple-choice questions based on the *AP Marine Biology Simulation Case Study*. (See the teacher's manual for the *AP Marine Biology Simulation Case Study* for examples.)

Directions: Determine the answer to each of the following questions or incomplete statements, using the available space for any necessary scratchwork. Then decide which is the best of the choices given and fill in the corresponding oval on the answer sheet. No credit will be given for anything written in the examination booklet. Do not spend too much time on any one problem.

Notes:

- Assume that the classes listed in the Quick Reference sheet have been imported where appropriate. A Quick Reference to the AP Java classes is included as part of the exam.
- Assume that declarations of variables and methods appear within the context of an enclosing class.
- Assume that method calls that are not prefixed with an object or class name appear within the context of the class in which the method is declared.
- Unless otherwise noted in the question, assume that parameters in method calls are not `null.`

1. Consider the following code segment.

```
for (int k = 0; k < 20; k = k + 2)
{
  if (k % 3 == 1)
    System.out.print(k + " ");
}
```

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

(A) `4  16`
(B) `4  10  16`
(C) `0  6  12  18`
(D) `1  4  7  10  13  16  19`
(E) `0  2  4  6  8  10  12  14  16  18`

2. Consider the following code segment.

```
ArrayList list = new ArrayList();

list.add(new Integer(1));
list.add(new Integer(2));
list.add(new Integer(3));
list.set(2, new Integer(4));
list.add(2, new Integer(5));
list.add(new Integer(6));
System.out.println(list);
```

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

(A) `[1, 2, 3, 4, 5]`
(B) `[1, 2, 4, 5, 6]`
(C) `[1, 2, 5, 4, 6]`
(D) `[1, 5, 2, 4, 6]`
(E) `[1, 5, 4, 3, 6]`

*3. Consider the following data field and method.

```
private ArrayList nums;

// precondition: nums.size() > 0;
//   nums contains Integer objects
public void numQuest()
{
  int k = 0;
  Integer zero = new Integer(0);

  while (k < nums.size())
  {
    if (nums.get(k).equals(zero))
      nums.remove(k);

    k++;
  }
}
```

Assume that `ArrayList` `nums` initially contains the following `Integer` values.

```
[0, 0, 4, 2, 5, 0, 3, 0]
```

What will `ArrayList` `nums` contain as a result of executing `numQuest` ?

(A) `[0, 0, 4, 2, 5, 0, 3, 0]`
(B) `[4, 2, 5, 3]`
(C) `[0, 0, 0, 0, 4, 2, 5, 3]`
(D) `[3, 5, 2, 4, 0, 0, 0, 0]`
(E) `[0, 4, 2, 5, 3]`

4. Consider the following declaration for a class that will be used to represent points in the *xy*-coordinate plane.

```
public class Point
{
  private int myX;     // coordinates
  private int myY;

  public Point( )
  {
    myX = 0;
    myY = 0;
  }

  public Point(int a, int b)
  {
    myX = a;
    myY = b;
  }

  // ... other methods not shown
}
```

The following incomplete class declaration is intended to extend the above class so that two-dimensional points can be named.

```
public class NamedPoint extends Point
{
  private String myName;

  // constructors go here

  // ... other methods not shown
}
```

Consider the following proposed constructors for this class.

```
  I. public NamedPoint()
     {
       myName = "";
     }

 II. public NamedPoint(int d1, int d2, String name)
     {
       myX = d1;
       myY = d2;
       myName = name;
     }

III. public NamedPoint(int d1, int d2, String name)
     {
       super(d1, d2);
       myName = name;
     }
```

Which of these constructors would be legal for the `NamedPoint` class?

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

5. Consider the following output.

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

Which of the following code segments will produce this output?

(A)
```
for (int j = 1; j <= 5; j++)
{
  for (int k = 1; k <= 5; k++)
  {
    System.out.print(j + " ");
  }
  System.out.println();
}
```

(B)
```
for (int j = 1; j <= 5; j++)
{
  for (int k = 1; k <= j; k++)
  {
    System.out.print(j + " ");
  }
  System.out.println();
}
```

(C)
```
for (int j = 1; j <= 5; j++)
{
  for (int k = 5; k >= 1; k–)
  {
    System.out.print(j + " ");
  }
  System.out.println();
}
```

```
(D) for (int j = 1; j <= 5; j++)
    {
      for (int k = 5; k >= j; k–)
      {
        System.out.print(j + " ");
      }
      System.out.println();
    }

(E) for (int j = 1; j <= 5; j++)
    {
      for (int k = j; k <= 5; k++)
      {
        System.out.print(k + " ");
      }
      System.out.println();
    }
```

6. A car dealership needs a program to store information about the cars for sale. For each car, they want to keep track of the following information: number of doors (2 or 4), whether the car has air conditioning, and its average number of miles per gallon. Which of the following is the best design?

   (A) Use one class, `Car`, which has three data fields: `int numDoors`, `boolean hasAir`, and `double milesPerGallon`.

   (B) Use four unrelated classes: `Car`, `Doors`, `AirConditioning`, and `MilesPerGallon`.

   (C) Use a class `Car` which has three subclasses: `Doors`, `AirConditioning`, and `MilesPerGallon`.

   (D) Use a class `Car`, which has a subclass `Doors`, with a subclass `AirConditioning`, with a subclass `MilesPerGallon`.

   (E) Use three classes: `Doors`, `AirConditioning`, and `MilesPerGallon`, each with a subclass `Car`.

7. Consider the following class declaration.

```
public class SomeClass implements Comparable
{
  // ... other methods not shown
}
```

Which of the following method signatures of `compareTo` will satisfy the `Comparable` interface requirement?

```
  I. public int compareTo(Object other)
 II. public int compareTo(SomeClass other)
III. public boolean compareTo(Object other)
```

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

Questions 8 – 9 refer to the following incomplete class declaration.

```
public class TimeRecord
{
  private int hours;
  private int minutes;  // 0<=minutes<60

  public TimeRecord(int h, int m)
  {
    hours = h;
    minutes = m;
  }

  // postcondition: returns the
  //   number of hours
  public int getHours()
  { /* implementation not shown */ }

  // postcondition: returns the number
  //   of minutes; 0 <= minutes < 60
  public int getMinutes()
  { /* implementation not shown */ }


  // precondition: h >= 0; m >= 0
  // postcondition: adds h hours and
  //   m minutes to this TimeRecord
  public void advance(int h, int m)
  {
    hours = hours + h;
    minutes = minutes + m;

    /* missing code */
  }

  // ... other methods not shown

}
```

8. Which of the following can be used to replace `/* missing code */` so that `advance` will correctly update the time?

   (A) `minutes = minutes % 60;`

   (B) `minutes = minutes + hours % 60;`

   (C)
   ```
   hours = hours + minutes / 60;
   minutes = minutes % 60;
   ```

   (D)
   ```
   hours = hours + minutes % 60;
   minutes = minutes / 60;
   ```

   (E) `hours = hours + minutes / 60;`

9. Consider the following declaration that appears in a client program.

```
TimeRecord[] timeCards = new TimeRecord[100];
```

Assume that `timeCards` has been initialized with `TimeRecord` objects. Consider the following code segment that is intended to compute the total of all the times stored in `timeCards`.

```
TimeRecord total = new TimeRecord(0,0);

for (int k = 0; k < timeCards.length; k++)
{
  /* missing expression */ ;
}
```

Which of the following can be used to replace `/* missing expression */` so that the code segment will work as intended?

(A) `timeCards[k].advance()`

(B) `total += timeCards[k].advance()`

(C)
```
total.advance(timeCards[k].hours,
              timeCards[k].minutes)
```

(D)
```
total.advance(timeCards[k].getHours(),
              timeCards[k].getMinutes())
```

(E)
```
timeCards[k].advance(timeCards[k].getHours(),
                     timeCards[k].getMinutes())
```

*10. Consider the following incomplete method, `calcTotal`, which is intended to return the sum of all the integer values represented by the elements in the `ArrayList` `list` of `Integer` objects.

```
public int calcTotal(ArrayList list)
{
  int total = 0;

  for (int index = 0;
           index < list.size(); index++)
  {
    /* missing code */
  }

  return total;
}
```

Which of the following can be used to replace `/* missing code */` so that `calcTotal` will work as intended?

```
  I. total += list.get(index);
 II. total += (Integer) list.get(index);
III. total += ((Integer) list.get(index)).intValue();
```

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

Questions 11 – 12 refer to the following information.

Consider the following data field and method `findLongest` with line numbers added for reference. Method `findLongest` is intended to find the longest consecutive block of the value `target` occurring in the array nums; however, `findLongest` does not work as intended.

For example, if the array `nums` contains the values
`[7, 10, 10, 15, 15, 15, 15, 10, 10, 10, 15, 10, 10]`,
the call `findLongest(nums, 10)` should return 3, the length of the longest consecutive block of 10's.

```
        private int[] nums;

        public int findLongest(int target)
        {
        int lenCount = 0;
        int maxLen = 0;

Line 1:   for (int k = 0; k < nums.length; k++)
Line 2:   {
Line 3:     if (nums[k] == target)
Line 4:     {
Line 5:       lenCount++;
Line 6:     }
Line 7:     else
Line 8:     {
Line 9:       if (lenCount > maxLen)
Line 10:      {
Line 11:        maxLen = lenCount;
Line 12:      }
Line 13:    }
Line 14:  }
Line 15:  if (lenCount > maxLen)
Line 16:  {
Line 17:    maxLen = lenCount;
Line 18:  }
Line 19:  return maxLen;
        }
```

*11. The method `findLongest` does not work as intended.
Which of the following best describes the value returned by a call to `findLongest` ?

(A) It is the length of the shortest consecutive block of the value `target` in `nums`.

(B) It is the length of the array `nums`.

(C) It is the number of occurrences of the value `target` in `nums`.

(D) It is the length of the first consecutive block of the value `target` in `nums`.

(E) It is the length of the last consecutive block of the value `target` in `nums`.

*12. Which of the following changes should be made so that method `findLongest` will work as intended?

(A) Insert the statement `lenCount = 0;` between lines 2 and 3.

(B) Insert the statement `lenCount = 0;` between lines 8 and 9.

(C) Insert the statement `lenCount = 0;` between lines 10 and 11.

(D) Insert the statement `lenCount = 0;` between lines 11 and 12.

(E) Insert the statement `lenCount = 0;` between lines 12 and 13.

*13. Consider the following data field and method.

```
private int[] myStuff;

// precondition:  myStuff contains
//   integers in no particular order
public int mystery(int num)
{
  for (int k = myStuff.length - 1; k >= 0; k--)
  {
    if (myStuff[k] < num)
    {
      return k;
    }
  }

  return -1;
}
```

Which of the following best describes the contents of `myStuff` after the following statement has been executed?

```
int m = mystery(n);
```

(A) All values in positions `0` through `m` are less than `n`.

(B) All values in positions `m+1` through `myStuff.length-1` are less than n.

(C) All values in positions `m+1` through `myStuff.length-1` are greater than or equal to `n`.

(D) The smallest value is at position `m`.

(E) The largest value that is smaller than `n` is at position `m`.

14. Consider the following method.

```
// precondition: x >= 0
public void mystery(int x)
{
  System.out.print(x % 10);

  if ((x / 10) != 0)
  {
    mystery(x / 10);
  }

  System.out.print(x % 10);
}
```

Which of the following is printed as a result of the call `mystery(1234)` ?

(A) `1441`
(B) `3443`
(C) `12344321`
(D) `43211234`
(E) Many digits are printed due to infinite recursion.

15. Consider the following two classes.

```
public class Base
{
  public void methodOne()
  {
    System.out.print("A");
    methodTwo();
  }

  public void methodTwo()
  {
    System.out.print("B");
  }
}

public class Derived extends Base
{
  public void methodOne()
  {
    super.methodOne();
    System.out.print("C");
  }

  public void methodTwo()
  {
    super.methodTwo();
    System.out.print("D");
  }
}
```

Assume that the following declaration appears in a client program.

```
Base b = new Derived();
```

What is printed as a result of the call `b.methodOne()` ?

(A) `AB`
(B) `ABC`
(C) `ABCD`
(D) `ABDC`
(E) Nothing is printed due to infinite recursion.

*16. Consider the following declarations.

```
Integer valueOne, valueTwo;
```

Assume that `valueOne` and `valueTwo` have been properly initialized. Which of the following is equivalent to the expression below?

```
valueOne.intValue() == valueTwo.intValue()
```

(A) `valueOne == valueTwo`
(B) `valueOne.compareTo(valueTwo)`
(C) `valueOne.equals(valueTwo) == 0`
(D) `valueOne.compareTo(valueTwo) == 0`
(E) `valueOne.intValue().equals(valueTwo.intValue())`

*17. Consider the following recursive method.

```
public static int mystery(int n)
{
  if (n == 0)
    return 1;
  else
    return 3 * mystery(n - 1);
}
```

What value is returned as a result of the call `mystery(5)` ?

(A) 0
(B) 3
(C) 81
(D) 243
(E) 6561

*18. Consider the following data field and method.

```
private int[] arr;

// precondition: arr.length > 0
public int checkArray()
{
  int loc = arr.length / 2;

  for (int k = 0; k < arr.length; k++)
  {
    if (arr[k] > arr[loc])
      loc = k;
  }

  return loc;
}
```

Which of the following is the best postcondition for `checkArray` ?

(A) Returns the index of the first element in array `arr` whose value is greater than `arr[loc]`

(B) Returns the index of the last element in array `arr` whose value is greater than arr[loc]

(C) Returns the largest value in array `arr`

(D) Returns the index of the largest value in array `arr`

(E) Returns the index of the largest value in the second half of array `arr`

*19. Consider the following data field and method.

```
private int[] arr;

// precondition: arr contains no duplicates,
//   the elements in arr are in sorted order,
//   0 ≤ low ≤ arr.length;
//   low - 1 ≤ high < arr.length
public int mystery(int low, int high, int num)
{
  int mid = (low + high) / 2;

  if (low > high)
  {
    return low;
  }
  else if (arr[mid] < num)
  {
    return mystery(mid + 1, high, num);
  }
  else if (arr[mid] > num)
  {
    return mystery(low, mid - 1, num);
  }
  else              // arr[mid] == num
  {
    return mid;
  }
}
```

What is returned by the call
`mystery(0, arr.length - 1, num)` ?

(A) The number of elements in `arr` that are less than `num`

(B) The number of elements in `arr` that are less than or equal to `num`

(C) The number of elements in `arr` that are equal to `num`

(D) The number of elements in `arr` that are greater than `num`

(E) The index of the middle element in `arr`

20. Assume the following declarations have been made.

```
private String s;
private int n;

public void changer(String x, int y)
{
  x = x + "peace";
  y = y * 2;
}
```

Assume `s` has the value `"world"` and `n` is `6`. What are the values of `s` and `n` after the call `changer(s, n)` ?

```
        s          n

(A) world          6
(B) worldpeace     6
(C) world         12
(D) worldpeace    12
(E) peace         12
```

21. At a certain high school students receive letter grades based on the following scale.

Numeric Score	Letter Grade
93 or above	A
From 84 to 92 inclusive	B
From 75 to 83 inclusive	C
Below 75	F

Which of the following code segments will assign the correct string to `grade` for a given integer score ?

```
  I. if (score >= 93)
       grade = "A";
     if (score >= 84 && score <= 92)
       grade = "B";
     if (score >= 75 && score <= 83)
       grade = "C";
     if (score < 75)
       grade = "F";

 II. if (score >= 93)
       grade = "A";
     if (84 <= score <= 92)
       grade = "B";
     if (75 <= score <= 83)
       grade = "C";
     if (score < 75)
       grade = "F";

III. if (score >= 93)
       grade = "A";
     else if (score >= 84)
       grade = "B";
     else if (score >= 75)
       grade = "C";
     else
       grade = "F";
```

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

Answers to Computer Science A Multiple-Choice Questions

1 – B	4 – D	7 – A	10 – C	13 – C	16 – D	19 – A
2 – C	5 – D	8 – C	11 – C	14 – D	17 – D	20 – A
3 – E	6 – A	9 – D	12 – E	15 – D	18 – D	21 – D

Sample Free-Response Questions

Following is a representative set of questions. Questions marked with an asterisk are also representative of AB examination questions. The AP Computer Science A Examination will include one free-response question based on the *AP Marine Biology Simulation Case Study*. (See the teacher's manual for the *AP Marine Biology Simulation Case Study* for examples.)

Directions: SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN JAVA.

Notes:

- Assume that the classes listed in the Quick Reference sheet have been imported where appropriate. A Quick Reference to the AP Java classes is included as part of the exam.
- Assume that declarations of variables and methods appear within the context of an enclosing class.
- Assume that method calls that are not prefixed with an object or class name appear within the context of the class in which the method is declared.
- Unless otherwise noted in the question, assume that parameters in method calls are not `null.`
- Unless otherwise noted, assume that methods are called only when their preconditions are satisfied.

1. In an instant runoff election there are two or more candidates and there are many voters. Each voter votes by submitting a ballot that is an <u>ordered list of all the candidates</u>, where the first name listed is the voter's first choice, the second name is the voter's second choice, and so on. There are no ties allowed on a voter's ballot.

   The election is decided by the following process.

   - Initially, all candidates are placed on the current candidate list.
   - As long as there are two or more candidates on the current candidate list, the following steps are repeated.
     1. Each ballot is examined for candidates on the current candidate list and a vote is counted for the current candidate that appears earliest in the list of names on the ballot. (On the first pass, this will be the first name on the ballot. In subsequent passes, it might not be the first name on the ballot. See the illustrations below.)
     2. The candidate(s) with the fewest votes is (are) eliminated from the current candidate list.
   - The last remaining candidate is the winner. If there is none, the election is not decisive.

   For example, suppose there are four candidates in the election: Chris, Jamie, Pat, and Sandy. Each ballot has these four names listed in order of the voter's preference, with the first choice appearing first in the list. Assume that seven ballots were submitted as shown in the following table.

Current Candidate List: Chris, Jamie, Pat, Sandy

Voter	Ballot	First Choice from Current Candidate List
0	Chris, Jamie, Pat, Sandy	Chris
1	Chris, Pat, Sandy, Jamie	Chris
2	Chris, Sandy, Pat, Jamie	Chris
3	Pat, Jamie, Sandy, Chris	Pat
4	Pat, Sandy, Chris, Jamie	Pat
5	Sandy, Pat, Jamie, Chris	Sandy
6	Jamie, Sandy, Pat, Chris	Jamie

In the first pass, Chris has 3 votes, Pat has 2 votes, Sandy has 1 vote, and Jamie has 1 vote. Jamie and Sandy are tied for the fewest votes; so both are eliminated, leaving Chris and Pat on the current candidate list. Voter preferences for these two candidates are shown in the following table.

Current Candidate List: Chris, Pat

Voter	Ballot	First Choice from Current Candidate List
0	Chris, Jamie, Pat, Sandy	Chris
1	Chris, Pat, Sandy, Jamie	Chris
2	Chris, Sandy, Pat, Jamie	Chris
3	Pat, Jamie, Sandy, Chris	Pat
4	Pat, Sandy, Chris, Jamie	Pat
5	Sandy, Pat, Jamie, Chris	Pat
6	Jamie, Sandy, Pat, Chris	Pat

In the second pass, Chris has 3 votes and Pat has 4 votes. Chris has fewest votes and is eliminated. Pat is the only remaining candidate and is therefore the winner of the election.

A ballot is modeled with the following partial class declaration.

```
public class Ballot
{
  // postcondition: returns the first
  //   choice candidate for this Ballot
  //   from those on the candidateList
  public String firstChoiceFrom(
                ArrayList candidateList)
  { /* code not shown */ }

  // ... constructors, other methods,
  //     and private data not shown
}
```

The `Ballot` method `firstChoiceFrom` returns the name of the candidate from `candidateList` that appears first on this ballot.

The set of ballots for all voters in an election is modeled with the following partial class declaration.

```
public class VoterBallots
{
  private ArrayList ballotList;
  // each entry is an instance of Ballot
  // representing one voter's ballot

  // precondition: candidate appears in
  //   candidateList
  // postcondition: returns the number
  //   of times that candidate is first
  //   among those on candidateList for
  //   elements of ballotList
  private int numFirstVotes(String candidate,
                      ArrayList candidateList)
  { /* to be implemented in part (a) */ }
  // precondition: each String in
  //   candidateList appears exactly
  //   once in each Ballot in ballotList
  // postcondition: returns a list of
  //   those candidates tied with the
  //   fewest first choice votes
  public ArrayList candidatesWithFewest(
                   ArrayList candidateList)
  { /* to be implemented in part (b) */ }

  // ... constructor(s) and other
  //     methods not shown
}
```

An instant runoff election is represented by the class `InstantRunoff` that encapsulates the process of selecting a winner by repeatedly applying the `VoterBallots` method `candidatesWithFewest` to a list of candidates that is reduced until only the winner remains. This class is not shown here.

(a) Write the `VoterBallots` method `numFirstVotes`. Method `numFirstVotes` should return the number of times `candidate` appears first, among those elements that are on `candidateList`, in elements of `ballotList`.

Complete method `numFirstVotes` below.

```
 // precondition: candidate appears in
 //   candidateList
 // postcondition: returns the number
 //   of times that candidate is first
 //   among those on candidateList for
 //   elements of ballotList
 private int numFirstVotes(String candidate,
                     ArrayList candidateList)
```

(b) Write the `VoterBallots` method `candidatesWithFewest.` Method `candidatesWithFewest` should count the number of times each `String` in the list `candidateList` appears first in an element of `ballotList`, and return an `ArrayList` of all those `Strings` that are tied for the smallest count.

In writing method `candidatesWithFewest` you may use the `private` helper method `numFirstVotes` specified in part (a). Assume that `numFirstVotes` works as specified, regardless of what you wrote in part (a). Solutions that reimplement functionality provided by this method, rather than invoking it, will not receive full credit.

Complete method `candidatesWithFewest` below.

```
// precondition: each String in
//   candidateList appears exactly
//   once in each Ballot in ballotList
// postcondition: returns a list of
//   those candidates tied with the
//   fewest first choice votes
// public ArrayList candidatesWithFewest(
                    ArrayList candidateList)
```

2. Consider the following incomplete declaration of a `LineEditor` class that allows insertions and deletions in a line of text. The line of text is stored internally as a `String.` The `insert` operation takes a `String` and inserts it into the line of text at the given index. The `delete` operation takes a String parameter and removes the first occurrence (if any) of that string from the line of text. The `deleteAll` operation removes all occurrences (if any) of a given `String` from the line of text, including any that are formed as a result of the deletion process.

```
public class LineEditor
{
  private String myLine;

  // precondition:   str is not null;
  //   0 <= index <= myLine.length()
  // postcondition: str has been inserted
  //   into myLine at position index;
  //   no characters from myLine have
  //   been overwritten
  public void insert(String str, int index)
  { /* to be implemented in part (a) */ }


  // precondition:   str is not null
  // postcondition: if str is found in
  //   myLine, the first occurrence
  //   of str has been removed from myLine;
  //   otherwise, myLine is left unchanged
  public void delete(String str)
  { /* to be implemented in part (b) */ }


  // precondition:  str is not null
  // postcondition: all occurrences of str
  //   have been removed from myLine;
  //   myLine is otherwise unchanged
  public void deleteAll(String str)
  { /* to be implemented in part (c) */ }
```

```
    // ... constructor and other methods
    //    not shown
  }
```

(a) Write the `LineEditor` method `insert` as described at the beginning of the question. The following tables show the result of several different calls to `insert`.

Method call: `insert("A.P.", 0)`

`myLine` before the call	`myLine` after the call
`"Computer Science"`	`"A.P.Computer Science"`

Method call: `insert(" is best", 16)`

`myLine` before the call	`myLine` after the call
`"Computer Science"`	`"Computer Science is best"`

Method call: `insert("Java", 4)`

`myLine` before the call	`myLine` after the call
`"Computer Science"`	`"CompJavauter Science"`

Complete method `insert` below.

```
// precondition: str is not null;
//    0 <= index <= myLine.length()
// postcondition: str has been inserted
//    into myLine at position index;
//    no characters from myLine have
//    been overwritten
public void insert(String str, int index)
```

(b) Write the `LineEditor` method `delete` as described at the beginning of the question. The following table shows the result of several different calls to `delete`.

Method call: `delete("Com")`

`myLine` before the call	`myLine` after the call
`"Computer Science"`	`"puter Science"`

Method call: `delete("ter Sc")`

`myLine` before the call	`myLine` after the call
`"Computer Science"`	`"Compuience"`

Method call: `delete("c")`

`myLine` before the call	`myLine` after the call
`"Computer Science"`	`"Computer Sience"`

Method call: `delete("Java")`

`myLine` before the call	`myLine` after the call
`"Computer Science"`	`"Computer Science"`

Complete method `delete` below.

```
// precondition:  str is not null
// postcondition: if str is found in
//   myLine, the first occurrence
//   of str has been removed from myLine;
//   otherwise, myLine is left unchanged
public void delete(String str)
```

(c) Write the `LineEditor` method `deleteAll` as described at the beginning of the question. The following table shows the result of several different calls to `deleteAll`.

Method call: `deleteAll("ing")`

`myLine` before the call	`myLine` after the call
`"string programming"`	`"str programm"`

Method call: `deleteAll("r")`

`myLine` before the call	`myLine` after the call
`"string programming"`	`"sting pogamming"`

Method call: `deleteAll("aba")`

`myLine` before the call	`myLine` after the call
`"abababa"`	`"b"`

Method call: `deleteAll("oh-la")`

`myLine` before the call	`myLine` after the call
`"ooh-lah-lah"`	`"h"`

Method call: `deleteAll("zap")`

`myLine` before the call	`myLine` after the call
`"pizza pie"`	`"pizza pie"`

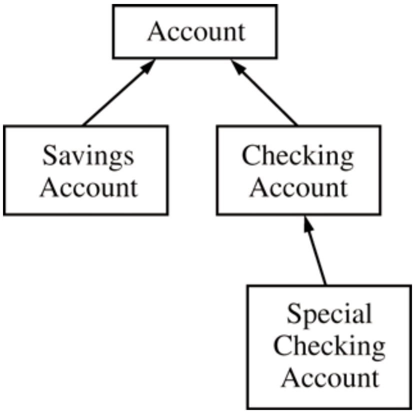
In writing `deleteAll`, you may call any of the methods in the `LineEditor` class, including `insert` and `delete` from parts (a) and (b). Assume that these methods work as specified, regardless of what you wrote in parts (a) and (b). Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit.

Complete method `deleteAll` below.

```
// precondition:  str is not null
// postcondition: all occurrences of str
//    have been removed from myLine;
//    myLine is otherwise unchanged
public void deleteAll(String str)
```

Note: The following question is somewhat longer than what may appear on the AP Computer Science Examinations. In particular, a question of this type appearing on the AP Computer Science A Exam might be limited to two parts.

*3. Consider the problem of modeling bank accounts. A diagram of the class hierarchy used to represent bank accounts is shown below.



The abstract class `Account` models a bank account with the following data and operations.

Data

- the identity number for the account (the identity number is never changed once the account has been constructed)
- the balance in the account (the balance can change as a result of some operations)

Operations

- create an account with a given identity number and initial balance
- return the identity number
- return the current balance
- deposit some positive amount into the account, increasing the balance
- decrease the balance by a specified positive amount; if the amount is greater than the balance, throw an `IllegalArgumentException`
- return the monthly interest due

An implementation for this class is shown below.

```
public abstract class Account
{
  private int idNumber;
    // identity number for this account
  private double balance;
    // current balance for this account

  // precondition:  startBal >= 0.0
  // postcondition: An Account with
  //   identity number idNum and
  //   current balance of startBal
  //   has been created
  // exceptions: If startBal < 0.0, an
  //   IllegalArgumentException is thrown
  public Account(int idNum, double startBal)
  { /* code not shown */ }
  // postcondition: returns the identity
  //   number for this account
  public int idNumber()
  { /* code not shown */ }

  // postcondition: returns the current
  //   balance for this account
  public double currentBalance()
  { /* code not shown */ }

  // precondition:  amount >= 0.0
  // postcondition: the current balance of
  //   this account has been increased
  //   by amount;
  // exceptions: if amount < 0.0, then
  //   current balance is unchanged and an
  //   IllegalArgumentException is thrown
  public void deposit(double amount)
  { /* code not shown */ }

  // precondition:  0.0 <= amount <= balance
  // postcondition: the current balance of
  //   this account has been decreased
  //   by amount;
  // exceptions: if amount < 0.0 or if
  //   amount > balance, then current
  //   balance is unchanged and an
  //   IllegalArgumentException is thrown
  public void decreaseBalance(double amount)
  { /* code not shown */ }

  // postcondition: returns the monthly
  //   interest due for this account
  public abstract double monthlyInterest();
}
```

(a) A savings account at a bank “is-a” bank account and is modeled by the class `SavingsAccount.` A savings account has all the characteristics of a bank account. In addition, a savings account has an interest rate, and the interest due each month is calculated from that interest rate. The operations for a savings account that differ from those specified in the class `Account` are the following.

- create a new savings account with a given annual interest rate, as well as the parameters required for all accounts
- withdraw a positive amount that does not exceed the current balance, decreasing the balance by the amount withdrawn
- calculate the monthly interest by multiplying the current balance by the annual interest rate divided by twelve

Write the complete definition of the class `SavingsAccount`, including the implementation of methods.

(b) A checking account at a bank “is-a” bank account and is modeled by the class `CheckingAccount.` A checking account has all the characteristics of a bank account. In addition, a checking account can have checks written on it. Each check written decreases the account by the amount of the check plus a per-check charge. The operations for a checking account that differ from those specified in the class `Account` are the following.

- create a new checking account with a given per-check charge, as well as the parameters required for all accounts
- clear a check for a given amount by decreasing the balance by the amount of the check plus the per-check charge
- compute and return the monthly interest

A declaration of the class `CheckingAccount` is shown below.

```
public class CheckingAccount extends Account
{
  private double checkCharge;

  public CheckingAccount(int idNum,
                         double startBal,
                         double chkCharge)
  {
    super(idNum, startBal);
    checkCharge = chkCharge;
  }

  public void clearCheck(double amount)
  {
    decreaseBalance(amount + checkCharge);
  }

  public double monthlyInterest()
  { /* code not shown */ }
}
```

A special checking account "is-a" checking account and is modeled by the class `SpecialCheckingAccount.` A special checking account has all the characteristics of a checking account. In addition, a special checking account has a minimum balance and an annual interest rate. When the balance is above the minimum balance, the per-check charge is not deducted from the balance when a check is cleared. Otherwise, a check is cleared just as it is for a checking account. In addition, when the balance is above the minimum balance when interest is calculated, interest due is calculated on the current balance. Otherwise, the interest due is the same as for a checking account. The operations for a special checking account that differ from those specified in the class `CheckingAccount` are the following.

- create a new special checking account with a given minimum balance and interest rate, as well as the parameters required for a checking account
- clear a check for a given amount according to the rules above
- calculate the monthly interest by multiplying current balance by the annual interest rate divided by twelve if the current balance is above the minimum; otherwise, calculate the interest as it is done for a checking account

Write the complete definition of the class `SpecialCheckingAccount`, including the implementation of its methods.

(c) Consider the class `Bank` partially specified below.

```
public class Bank
{
  private ArrayList accounts;
    // all accounts in this bank
    // accounts has no null entries

  // postcondition: for each account in
  //   this bank, the monthly interest
  //   due has been deposited into
  //   that account
  public void postMonthlyInterest()
  {
    // to be implemented in this part
  }

  // ... constructors and other methods
  //     not shown
}
```

Write the `Bank` method `postMonthlyInterest`, which is described as follows. For each account in this bank, `postMonthlyInterest` should calculate the monthly interest and deposit that amount into the account.

In writing `postMonthlyInterest`, you may use any of the public methods of class `Account` or its subclasses. Assume these methods work as specified. Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit.

Complete method `postMonthlyInterest` below.

```
// postcondition: for each account in
//   this bank, the monthly interest
//   due has been deposited into
//   that account
public void postMonthlyInterest()
```

Suggested Solutions to Free-Response Questions

Note: There are many correct variations of these solutions.

Question 1

(a)

```
private int numFirstVotes(String candidate,
                          ArrayList candidateList)
{
  int numVotes = 0;

  for (int v = 0; v < ballotList.size(); v++)
  {
    Ballot voterBallot =
                  (Ballot) ballotList.get(v);
    String first =
            voterBallot.firstChoiceFrom(candidateList);

    if (candidate.equals(first))
      numVotes++;
  }
  return numVotes;
}
```

(b)

```
public ArrayList candidatesWithFewest(
                    ArrayList candidateList)
{
  int[] votes = new int[candidateList.size()];
  int minVotes = ballotList.size();

for (int c = 0; c < candidateList.size(); c++)
  {
    String candidate =
              (String) candidateList.get(c);
    votes[c] =  numFirstVotes(candidate,
                             candidateList);
    if (votes[c] < minVotes)
      minVotes = votes[c];
  }

  ArrayList result = new ArrayList();
  for (int c = 0; c < candidateList.size(); c++)
  {
    if (votes[c] == minVotes)
      result.add(candidateList.get(c));
  }

  return result;
}
```

(b) Alternate solution

```
public ArrayList candidatesWithFewest(
                     ArrayList candidateList)
{
  ArrayList result = new ArrayList();
  int minVotes = ballotList.size() + 1;

  for (int c = 0; c < candidateList.size(); c++)
  {
    String candidate =
              (String) candidateList.get(c);
    int thisVotes = numFirstVotes(candidate,
                                  candidateList);
    if (thisVotes < minVotes)
    {
      minVotes = thisVotes;
      result = new ArrayList();
    }
    if (thisVotes == minVotes)
      result.add(candidateList.get(c));
  }

  return result;
}
```

Question 2

(a)

```
public void insert(String str, int index)
{
  myLine = myLine.substring(0, index) + str
           + myLine.substring(index);
}
```

(b)

```
public void delete(String str)
{
  int index = myLine.indexOf(str);

  if (index != -1)
  {
    myLine = myLine.substring(0, index)
     + myLine.substring(index + str.length());
  }
}
```

(c)

```
public void deleteAll(String str)
{
  while (myLine.indexOf(str) != -1)
  {
    delete(str);
  }
}
```

Question 3

(a)

```
public class SavingsAccount extends Account
{
  private double intRate;
            // annual interest rate for this account

  public SavingsAccount(int idNum,
                        double balance,
                        double rate)
  {
    super(idNum, balance);
    intRate = rate;
  }

  public void withdraw(double amount)
  {
    decreaseBalance(amount);
  }

  public double monthlyInterest()
  {
    return (currentBalance() * (intRate / 12.0));
  }
}
```

(b)

```
public class SpecialCheckingAccount
              extends CheckingAccount
{
  private double minBalance;
  private double intRate;

  public SpecialCheckingAccount(int idNum,
           double startBal, double chkCharge,
           double minBal, double rate)
  {
    super(idNum, startBal, chkCharge);
    minBalance = minBal;
    intRate = rate;
  }

  public void clearCheck(double amount)
  {
    if (currentBalance() >= minBalance)
      decreaseBalance(amount);
    else
      super.clearCheck(amount);
  }

  public double monthlyInterest()
  {
    if (currentBalance() >= minBalance)
      return  (currentBalance() * (intRate / 12.0));
    else
      return super.monthlyInterest();
  }
}
```

(c)

```
public void postMonthlyInterest()
{
  double interest;

  for (int k = 0; k < accounts.size(); k++)
  {
    Account acct = (Account) accounts.get(k);
    interest = acct.monthlyInterest();
    acct.deposit(interest);
  }
}
```

Computer Science AB: Sample Multiple-Choice Questions

Following is a representative set of questions. Questions marked with an asterisk in the Computer Science A questions are also representative of Computer Science AB questions. The answer key for Computer Science AB multiple-choice questions is on page 103.) In this section of the examination, as a correction for haphazard guessing, one-fourth of the number of questions answered incorrectly will be subtracted from the number of questions answered correctly. The AP Computer Science AB Examination will include at least five multiple-choice questions based on the *AP Marine Biology Simulation Case Study.* (See the teacher's manual for the *AP Marine Biology Simulation Case Study* for examples.)

Directions: Determine the answer to each of the following questions or incomplete statements, using the available space for any necessary scratchwork. Then decide which is the best of the choices given and fill in the corresponding oval on the answer sheet. No credit will be given for anything written in the examination booklet. Do not spend too much time on any one problem.

Notes:

- Assume that the classes listed in the Quick Reference sheet have been imported where appropriate. A Quick Reference to the AP Java classes is included as part of the exam.
- Assume that the implementation classes are used for any questions referring to linked lists or trees and that the interfaces for stacks, queues, and priority queues behave as specified.
- Assume that declarations of variables and methods appear within the context of an enclosing class.
- Assume that method calls that are not prefixed with an object or class name appear within the context of the class in which the method is declared.
- Unless otherwise noted in the question, assume that parameters in method calls are not `null`.

1. Consider the following code segment.

```
int [][] mat = new int [3][4];

for (int row = 0; row < mat.length; row++)
{
  for (int col = 0; col < mat[0].length; col++)
  {
    if (row < col)
      mat[row][col] = 1;
    else if (row == col)
      mat[row][col] = 2;
    else
      mat[row][col] = 3;
  }
}
```

What are the contents of `mat` after the code segment has been executed?

(A)
```
2   1   1
3   2   1
3   3   2
3   3   3
```

(B)
```
2   3   3
1   2   3
1   1   2
1   1   1
```

(C)
```
2   3   3   3
1   2   3   3
1   1   2   3
```

(D)
```
2   1   1   1
3   2   1   1
3   3   2   1
```

(E)
```
1   1   1   1
2   2   2   2
3   3   3   3
```

2. Which of the following best describes the data structure represented by `java.util.LinkedList` ?

   (A) A singly linked list with a reference to the first node only

   (B) A singly linked list with references to the first and last nodes

   (C) A doubly linked list with a reference to the first node only

   (D) A doubly linked list with references to the first and last nodes only

   (E) A doubly linked list with reference to the first, middle, and last nodes

3. Consider the following code segment.

```
Queue que = new ListQueue();
  // ListQueue implements Queue

Object obj;

que.enqueue("a");
que.enqueue("b");
que.enqueue("c");
obj = que.peekFront();
que.enqueue(obj + "d");
que.enqueue(obj + "x" + obj);

while (! que.isEmpty())
{
  System.out.print(que.dequeue() + " ");
}
```

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

(A) `a b c ad`

(B) `b c ad axa`

(C) `axa ad c b a`

(D) `ad axa b c a`

(E) `a b c ad axa`

4. Consider the following declarations.

```
Stack s = new ListStack();
  // ListStack implements Stack

Queue q = new ListQueue();
  // ListQueue implements Queue
```

Assume that `s` is initially empty and that `q` initially contains the following strings.

```
   W X Y Z
   ↑     ↑
 front  back
```

Consider the following code segment.

```
while (!q.isEmpty())
  s.push(q.dequeue());

while (!s.isEmpty())
  q.enqueue(s.pop());
```

Which of the following best describes stack `s` and queue `q` after the code segment has been executed?

(A) Stack `s` is empty and queue `q` contains `W`, `X`, `Y`, `Z`, in that order, with `W` at the front of the queue.

(B) Stack `s` is empty and queue `q` contains `Z`, `Y`, `X`, `W`, in that order, with `Z` at the front of the queue.

(C) Stack `s` contains `Z`, `Y`, `X`, `W`, in that order, with `Z` at the top of the stack, and queue `q` is empty.

(D) Stack `s` contains `Z`, `Y`, `X`, `W`, in that order, with `Z` at the top of the stack; and queue `q` contains `W`, `X`, `Y`, `Z`, in that order, with `W` at the front of the queue.

(E) Stack `s` contains `Z`, `Y`, `X`, `W`, in that order, with `Z` at the top of the stack; and queue `q` contains `Z`, `Y`, `X`, `W`, in that order, with `Z` at the front of the queue.

5. Consider the following code segment.

```
for (int j = 1; j <= n; j++)
{
  for (int k = 1; k <= n; k = k * 2)
  {
    System.out.println(j + " " + k);
  }
}
```

Of the following, which best characterizes the running time of the code segment?

(A) $O(\log n)$

(B) $O(n)$

(C) $O(n \log n)$

(D) $O(n^2)$

(E) $O(n!)$

6. The following integers are inserted into an empty binary search tree in the following order.

```
26   20   37   31   22   18   25   29   19
```

Which traversal of the tree would produce the following output?

```
26   20   37   18   22   31   19   25   29
```

(A) Preorder

(B) Inorder

(C) Postorder

(D) Reverse postorder

(E) Level-by-level

7. Consider the following methods.

```
public List process1(int n)
{
  ArrayList someList = new ArrayList();

  for (int k = 0; k < n; k++)
    someList.add(new Integer(k));

  return someList;
}

public List process2(int n)
{
  ArrayList someList = new ArrayList();

  for (int k = 0; k < n; k++)
   someList.add(k, new Integer(k));

  return someList;
}
```

Which of the following best describes the behavior of `process1` and `process2` ?

(A) Both methods produce the same result and take the same amount of time.

(B) Both methods produce the same result, and `process1` is faster than `process2`.

(C) The two methods produce different results and take the same amount of time.

(D) The two methods produce different results, and `process1` is faster than `process2`.

(E) The two methods produce different results, and `process2` is faster than `process1`.

8. Consider the following data field and incomplete method. Method `hasItem` should return `true` if `item` is found in `myList`; otherwise, it should return false.

```
private List myList;

public boolean hasItem(Object item)
{
  Iterator itr = myList.iterator();

  while ( /* condition */ )
  {
    if (itr.next().equals(item))
        return true;
  }

  return false;
}
```

Which of the following expressions can be used to replace `/* condition */` so that `hasItem` will work as intended?

(A) `itr.hasNext()`

(B) `myList.hasNext()`

(C) `! itr.hasNext()`

(D) `! myList.hasNext()`

(E) `itr != null`

9. Consider the following method.

```
public static void mystery(ListNode p)
{
  if (p != null)
  {
    mystery(p.getNext().getNext());
    p.setNext(p.getNext().getNext());
  }
}
```

What changes does mystery make to the list whose first node is `p` ?

(A) It makes no changes to the list.

(B) It removes the first, third, and all odd nodes from the list.

(C) It removes the second, fourth, and all even nodes from the list.

(D) It removes all nodes except the first node of the list.

(E) If the number of nodes in the list is odd, it will cause a `NullPointerException`; otherwise,
it removes half of the nodes from the list.

10. Consider the following partial class declaration.

```
public class LList
{
  private ListNode front;

  public LList()
  {
    front = null;
  }

  public void addToLList(Comparable obj)
  {
    front = addHelper(front, obj);
  }

  private ListNode addHelper(
          ListNode list, Comparable obj)
  {
    if (list == null ||
        obj.compareTo(list.getValue()) < 0)
    {
      list = new ListNode(obj, list);
      return list;
    }
    else
    {
      list.setNext(addHelper(
                      list.getNext(), obj));
      return list;
    }
  }

  // ... other methods and data not shown
}
```

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

```
LList list = new LList();

list.addToLList("manager");
list.addToLList("boy");
list.addToLList("girl");
list.addToLList("anyone");
list.addToLList("place");
list.addToLList("vector");
```

What values are in `list` after the code segment has been executed?

(A) `[anyone, boy, girl, manager, place, vector]`

(B) `[manager, boy, girl, anyone, place, vector]`

(C) `[vector, place, anyone, girl, boy, manager]`

(D) `[vector, place, manager, girl, boy, anyone]`

(E) Nothing is in `list` because a `NullPointerException` was thrown during the execution.

11. Consider the following data field and incomplete method, `partialSum`, which is intended to return an integer array sum such that for all `i`, `sum[i]` is equal to `arr[0] + arr[1] + ... + arr[i]`. For instance, if `arr` contains the values `{ 1, 4, 1, 3 }`, the array `sum` will contain the values `{ 1, 5, 6, 9 }`.

```
private int[] arr;

public int[] partialSum()
{
  int[] sum = new int[arr.length];

  for (int j = 0; j < sum.length; j++)
    sum[j] = 0;

  /* missing code */

  return sum;
}
```

The following two implementations of `/* missing code */` are proposed so that `partialSum` will work as intended.

Implementation 1

```
for (int j = 0; j < arr.length; j++)
  sum[j] = sum[j - 1] + arr[j];
```

Implementation 2

```
for (int j = 0; j < arr.length; j++)
  for (int k = 0; k <= j; k++)
    sum[j] = sum[j] + arr[k];
```

Which of the following statements is true?

(A) Both implementations work as intended, but implementation 1 is faster than implementation 2.

(B) Both implementations work as intended, but implementation 2 is faster than implementation 1.

(C) Both implementations work as intended and are equally fast.

(D) Implementation 1 does not work as intended, because it will cause an `ArrayIndexOutOfBoundsException`.

(E) Implementation 2 does not work as intended, because it will cause an `ArrayIndexOutOfBoundsException`.

Questions 12 – 13 refer to the following method.

```
private static void sort(List theList)
{
  Iterator itr = theList.iterator();

  PriorityQueue pq = new PriorityQueueImpl();
  // PriorityQueueImpl implements PriorityQueue

  while (itr.hasNext())
  {
    pq.add(itr.next())
    itr.remove(); // removes last item
                  // just seen by itr
  }

  while ( !pq.isEmpty())
    theList.add(pq.removeMin());
}
```

12. If the parameter is a `LinkedList` in <u>sorted</u> order, and the `PriorityQueueImpl` is implemented as a min-heap, which data structure operation dominates the running time of the sort?

    (A) `pq.add`

    (B) `itr.remove`

    (C) `pq.isEmpty`

    (D) `pq.removeMin`

    (E) `theList.add`

13. If the parameter is an `ArrayList`, and the `PriorityQueueImpl` is implemented as a min-heap, which data structure operation dominates the running time of the sort?

    (A) `pq.add`

    (B) `itr.remove`

    (C) `pq.isEmpty`

    (D) `pq.removeMin`

    (E) `theList.add`

14. An electronic mail application includes a class for handling mail aliases. A mail alias is a single name that represents a collection of mail addresses. For example, someone might set up the following mail aliases.

Alias	**Collection of mail addresses**
`family`	`myMom@business.com, myDad@isp.net, mySis@school.edu`
`buddies`	`pat@school1.edu, chris@school2.edu, taylor@school3.edu`
`teammates`	`jamie@myschool.edu, alex@myschool.edu`

Which of the following describes the best choice of data structures for representing an individual's mail aliases in the electronic mail application?

(A) Represent the mail aliases as a two-dimensional array of strings with the alias name in one column and the collection of mail addresses for that alias as a single string in a second column.

(B) Represent the mail aliases as an array of linked lists of strings, where the first element in each linked list is the alias name and the remaining elements in the linked list are the mail addresses for that alias.

(C) Represent the mail aliases as an `ArrayList` of `Maps`, where each `Map` has one key, the alias name, and one value, a set of strings representing the mail addresses for that alias.

(D) Represent the mail aliases as a `Map`, using the alias names as the keys and sets of mail addresses as the values.

(E) Represent the mail aliases as a stack of linked lists, where the first element in each linked list is the alias and the remaining elements in the linked list are the mail addresses for that alias.

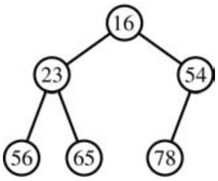
15. A min-heap with no duplicate values is a binary tree in which the value in each node is smaller than the values in its children's nodes.

    Suppose the following values are inserted sequentially into an empty heap in the following order.

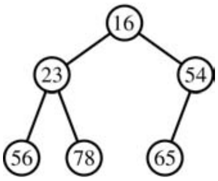
    ```
    23, 56, 54, 16, 78, 65
    ```

    What will the contents of the min-heap be after the values are inserted?

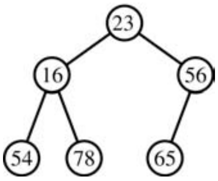
(A)



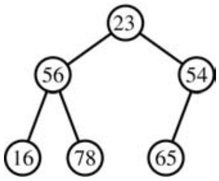
(B)



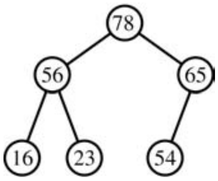
(C)



(D)



(D)



Answers to Computer Science AB Multiple-Choice Questions

1 – D	4 – B	7 – A	10 – A	13 – B
2 – D	5 – C	8 – A	11 – D	14 – D
3 – E	6 – E	9 – E	12 – D*	15 – B

* The standard array implementation of a heap should be assumed unless otherwise explictly stated.

Sample Free-Response Questions

Following is a representative set of questions. The AP Computer Science AB Examination will include one free-response question based on the *AP Marine Biology Simulation Case Study.* (See the teacher's manual for the *AP Marine Biology Simulation Case Study* for examples.)

Directions: SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN JAVA.

Notes:

- Assume that the classes listed in the Quick Reference sheet have been imported where appropriate. A Quick Reference to the AP Java classes is included as part of the exam.
- Assume that the implementation classes are used for any questions referring to linked lists or trees and that the interfaces for stacks, queues, and priority queues behave as specified.
- Assume that declarations of variables and methods appear within the context of an enclosing class.
- Assume that method calls that are not prefixed with an object or class name appear within the context of the class in which the method is declared.
- Unless otherwise noted in the question, assume that parameters in method calls are not `null`.
- Unless otherwise noted, assume that methods are called only when their preconditions are satisfied.

1. Consider using a *radix sort* to order a list of nonnegative integers. A radix sort makes as many passes through the list as there are digits in the largest number to be sorted. For example, if the largest integer in the list was 492, then the algorithm would make three passes through the list to sort it.

   Assume that the list of numbers to be sorted is in an integer array called `nums`. In each pass through the list, the radix sort algorithm sorts the numbers based on a different digit, working from the least to the most significant digit. To do this, it uses an intermediate data structure, `queues`, an array of ten queues. Each number is placed into the queue corresponding to the value of the digit being examined. For example, in the first pass the digit in the ones place is considered, so the number 345 would be enqueued into `queues[5]`. The number 260 would be enqueued into `queues[0]`. In each pass, the algorithm moves the numbers to be sorted from `nums` to the array of queues and then back to `nums`, as described below. After the last pass, the integers in `nums` are in order from smallest to largest.

   Radix Sort Algorithm:

   In each pass through the list, do the following two steps.

   Step 1

   Taking each integer from `nums` in order, insert the integer into the queue corresponding to the value of the digit currently being examined. If the integer being examined does not have a digit at a given place value, 0 is assumed for that place value. For example, 95 has no digit in the hundreds place, so, when examining the hundreds digit, the algorithm would assume the value in the hundreds place is zero and enqueue 95 into `queues[0]`.

   Step 2

   After all integers have been inserted into the appropriate queues, the array `nums` is filled from beginning to end by emptying the queues into `nums`, starting with the integers in `queues[0]` and proceeding sequentially through `queues[9]`.

For example, assume that `nums` contains the integers 380, 95, 345, 382, 260, 100, and 492. The sort will take three passes, because the largest integer in `nums` has 3 digits. The following diagram shows the sorting process. (For passes II and III, only the nonempty queues are shown in order to save space.)

```
   nums        queues                                    nums
  Before       After  Front              Rear            After
   Pass        Step 1   ↓                  ↓             Step 2

                           Pass I

[0]     380   [0]     380     260     100    [0]     380
[1]      95   [1]                            [1]     260
[2]     345   [2]     382     492            [2]     100
[3]     382   [3]                            [3]     382
[4]     260   [4]                            [4]     492
[5]     100   [5]      95     345            [5]      95
[6]     492   [6]                            [6]     345
              [7]
              [8]
              [9]

                           Pass II

[0]     380   [0]     100                    [0]     100
[1]     260   [4]     345                    [1]     345
[2]     100   [6]     260                    [2]     260
[3]     382   [8]     380     382            [3]     380
[4]     492   [9]     492      95            [4]     382
[5]      95                                  [5]     492
[6]     345                                  [6]      95

                           Pass III

[0]     100   [0]      95                    [0]      95
[1]     345   [1]     100                    [1]     100
[2]     260   [2]     260                    [2]     260
[3]     380   [3]     345     380     382    [3]     345
[4]     382   [4]     492                    [4]     380
[5]     492                                  [5]     382
[6]      95                                  [6]     492
```

The radix sort is implemented using the following class declaration.

```
public class RadixSort
{
  // precondition:  number >= 0; k >= 0
  // postcondition: returns kth digit
  //   of number, where k = 0 is the
  //   least significant digit
  private static int getDigit(int number, int k)
  { /* implementation not shown */ }

  // precondition:  nums.length > 0;
  //   all values in nums are nonnegative;
  //   k >= 0, where k represents the
  //   position of the digit used to
  //   determine queue placement
  // postcondition: returns an array of
  //   10 queues as described in the
  //   example; the total number of values
  //   in the array of queues is equal to
  //   nums.length
  private static Queue[] itemsToQueues(
                         int[] nums, int k)
  { /* to be implemented in part (a) */ }

  // precondition:  queues.length is 10;
  //   numVals is the number of values
  //   in all 10 queues
  // postcondition: returns an array that
  //   contains the integers from queues[0]
  //   through queues[9] in the order in
  //   which they were stored in the
  //   queues;
  //   each queue in queues is empty
  private static int[] queuesToArray(
                       Queue[] queues, int numVals)
  { /* to be implemented in part (b) */ }
  // precondition:  nums.length > 0;
  //   all values in nums are nonnegative;
  //   the largest value in nums has
  //   numDigits digits
  // postcondition: returns an array of all
  //   the values found in nums, sorted in
  //   nondecreasing order
  public static int[] sort(int[] nums, int numDigits)
  { /* to be implemented in part (c) */ }
}
```

Assume that the following class is available for defining `Queue` objects.

```
public class ListQueue implements Queue
{ /* implementation not shown */ }
```

(a) Write the `RadixSort` method `itemsToQueues`, which is described as follows. Method `itemsToQueues` corresponds to step 1 of each pass of the radix sort algorithm, creating the intermediate array of ten queues. Each integer in `nums` is inserted into the queue corresponding to the value of the digit currently being examined. If an integer does not have a digit at the given place value, 0 is assumed for that place value. The digit being examined is found in the `k`th position of the number. Integers are processed in the order in which they occur in `nums`.

In writing `itemsToQueues`, you may call the `RadixSort` method `getDigit`, which returns the `k`th digit of its parameter, `number`. The least significant digit is indicated by a value of 0 for `k`. If `k` is greater than the number of digits in number, then `getDigit` returns 0.

The following table illustrates the results of several calls to `getDigit`.

number	k	getDigit(number, k)
95	0	5
95	1	9
95	2	0

You do not need to implement `getDigit`.

Complete method `itemsToQueues` below.

```
// precondition: nums.length > 0;
//   all values in nums are nonnegative;
//   k >= 0, where k represents the
//   position of the digit used to
//   determine queue placement
// postcondition: returns an array of
//   10 queues as described in the
//   example; the total number of values
//   in the array of queues is equal to
//   nums.length
private static Queue[] itemsToQueues(
                          int[] nums, int k)
```

(b) Write the `RadixSort` method `queuesToArray`, which is described as follows. Method `queuesToArray` corresponds to step 2 of each pass of the radix sort algorithm, creating a new list from the values in the array of queues.

Complete method `queuesToArray` below.

```
// precondition:  queues.length is 10;
//   numVals is the number of values
//   in all 10 queues
// postcondition: returns an array that
//   contains the integers from queues[0]
//   through queues[9] in the order in
//   which they were stored in the
//   queues;
//   each queue in queues is empty
private static int[] queuesToArray(
                     Queue[] queues, int numVals)
```

(c) Write the `RadixSort` method `sort`, as started below. In writing `sort`, you may call methods `getDigit`, `itemsToQueues`, and `queuesToArray`. Assume that `itemsToQueues` and `queuesToArray` work as specified, regardless of what you wrote in parts (a) and (b). Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit.

Complete method `sort` below.

```
// precondition:  nums.length > 0;
//   all values in nums are nonnegative;
//   the largest value in nums has
//   numDigits digits
// postcondition: returns an array of all
//   the values found in nums, sorted in
//   nondecreasing order
public static int[] sort(int[] nums, int numDigits)
```

2. Consider the following interface `CityInfo` that will be used to represent cities in the United States. Each city is represented by its name and the name of the state in which it is located.

   ```
   public interface CityInfo
   {
     String city();
     String state();
   }
   ```

   The following class, `States`, will be used to store states and their respective cities. Information from `CityInfo` objects will be stored in this class as a `TreeMap.` In the `TreeMap`, the keys are the state names, and for each key the corresponding value is a `Set` of the cities in that state.

```
public class States
{
  private Map theMap;

  public States() { theMap = new TreeMap(); }

  // postcondition: Information from theCity
  //   has been added to the Map
  public void addCityToMap(CityInfo theCity)
  { /* to be implemented in part (a) */ }


  public void printOneState(String theState)
  { /* to be implemented in part (b) */ }


  public void printAllStates()
  { /* to be implemented in part (c) */ }


  // ... other methods not shown
}
```

For example, assume that a `States` object, `stateMap`, has been initialized with the following `CityInfo` objects.

```
[Albany,NY] [Miami,FL] [Hamilton,NY]
[Jacksonville,FL] [Dallas,TX]
```

The following table represents the entries in `stateMap`.

Key	**Value**
`FL`	`[Miami, Jacksonville]`
`NY`	`[Albany, Hamilton]`
`TX`	`[Dallas]`

(a) Write the `States` method `addCityToMap`, which is described as follows. Method `addCityToMap` takes one parameter: a new `CityInfo` object, and updates `theMap` to include the information encapsulated in the `CityInfo` object. Method `addCityToMap` should run in $O(\log n)$ expected time where n is the number of states in `theMap.`

The following tables show the result of two sequential calls to `addCityToMap`, when applied to the object `stateMap` shown at the beginning of the question. Assume that `city1` has been defined as the `CityInfo` object `[Albany,GA]` and `city2` has been defined as the `CityInfo` object `[Houston,TX].`

<u>Result of the call</u>

`stateMap.addCityToMap(city1);`

Key	**Value**
FL	[Miami, Jacksonville]
GA	[Albany]
NY	[Albany, Hamilton]
TX	[Dallas]

<u>Result of the call</u>

`stateMap.addCityToMap(city2);`

Key	**Value**
FL	[Miami, Jacksonville]
GA	[Albany]
NY	[Albany, Hamilton]
TX	[Dallas, Houston]

Complete method `addCityToMap` below.

```
// postcondition: information from theCity
//   has been added to theMap
public void addCityToMap(CityInfo theCity)
```

(b) Write method `printOneState`, which is described as follows. Method `printOneState` takes a `String` representing a state that is in `theMap.` It prints the name of the state and a list of cities in the state. The output should not include `[]`, and the cities should each be separated by a blank space. Method `printOneState` should run $O(c + \log n)$ in time, where n is the number of states in `theMap` and c is the number of cities associated with the given state.

For example, if `stateMap` contains the entries shown at the beginning of the question, the call `stateMap.printOneState("FL")` will result in the following output.

```
FL Miami Jacksonville
```

Complete method `printOneState` below. A solution that creates an unnecessary instance of any Collection class will not receive full credit.

```
public void printOneState(String theState)
```

(c) Write method `printAllStates`, which is described as follows. Method `printAllStates` outputs the cities in each state in the format shown in part (b). The states should be listed in alphabetical order. Method `printAllStates` should run in $O(c + n \log n)$ time, where n is the number of states in `theMap` and c is the total number of cities stored in `theMap`.

For example, if the `States` object `stateMap` has the following entries,

Key	**Value**
FL	[Miami, Jacksonville]
GA	[Albany]
NY	[Albany, Hamilton]
TX	[Dallas, Houston]

then the call `stateMap.printAllStates()` will produce the following output.

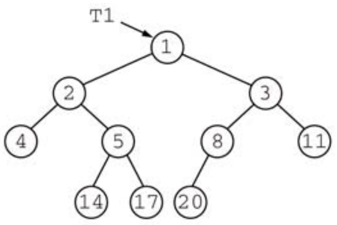
```
FL Miami Jacksonville
GA Albany
NY Albany Hamilton
TX Dallas Houston
```

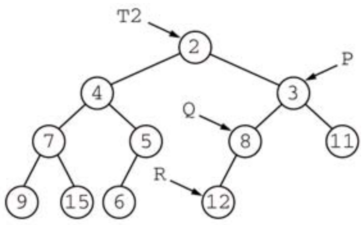
In writing `printAllStates`, you may call `printOneState` as specified in part (b). Assume that `printOneState` works as specified, regardless of what you wrote in part (b). Solutions that reimplement functionality provided by this method, rather than invoking this method, will not receive full credit.

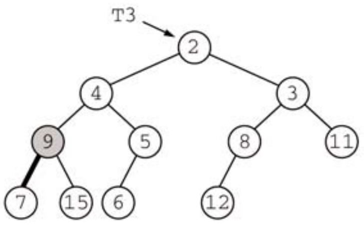
Complete method `printAllStates` below. A solution that creates an unnecessary instance of any Collection class will not receive full credit.

```
public void printAllStates()
```

3. A min-heap with no duplicate values is a binary tree in which the value in each node is smaller than the values in its children's nodes. A tree consisting of zero or one node is by default a min-heap. For example, trees `T1` and `T2` are min-heaps, but tree `T3` is not, because the shaded node has a larger value than one of its children (as shown by the thick line).





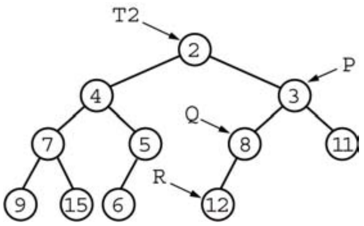


Consider the `MinHeap` class as shown below. The nodes of the min-heap will be represented by the `TreeNode` implementation class as shown in the Quick Reference. You may assume that the values stored in the `TreeNode` objects are `Comparable` objects.

```
public class MinHeap
{
  private TreeNode root;

  private TreeNode smallerChild(TreeNode t)
  { /* to be implemented in part (a) */ }

  private boolean isHeapOrdered(TreeNode t)
  { /* to be implemented in part (b) */ }

  // precondition: t is not null
  private TreeNode removeMinHelper(TreeNode t)
  { /* to be implemented in part (c) */ }

  public Object removeMin()
  {
    if (root == null)
      return null;
    else
    {
      Object result = root.getValue();
      root = removeMinHelper(root);
      return result;
    }
  }

  // ... other methods not shown
}
```

(a) Write method `smallerChild`, which is described as follows. `smallerChild` returns a reference to the child node of `t` that contains the smaller value (or `null` if `t` is a leaf node). If `t` has only one child, `smallerChild` should return a reference to that child. If `t` is `null`, `smallerChild` should return `null`.

The following diagram is repeated for your convenience.



The following table shows the results of several calls to `smallerChild`.

Method call	**Return value**
`smallerChild(T2)`	`P`
`smallerChild(P)`	`Q`
`smallerChild(Q)`	`R`
`smallerChild(R)`	`null`
`smallerChild(null)`	`null`

Complete method `smallerChild below.`

```
private TreeNode smallerChild(TreeNode t)
```

(b) Write method `isHeapOrdered`, which is described as follows. `isHeapOrdered` returns true if `t` is a min-heap and false otherwise.

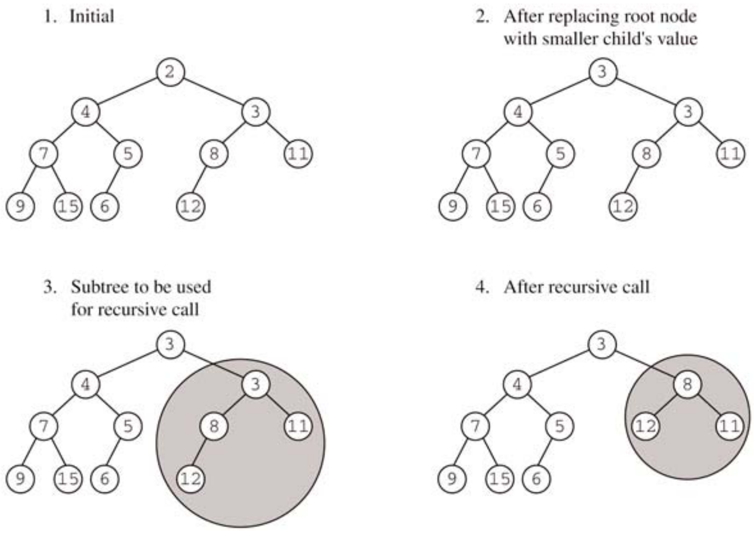
In writing `isHeapOrdered`, you may call method `smallerChild` specified in part (a). Assume that `smallerChild` works as specified, regardless of what you wrote in part (a). Solutions that reimplement functionality provided by this method, rather than invoking this method, will not receive full credit.

Complete method `isHeapOrdered` below.

```
private boolean isHeapOrdered(TreeNode t)
```

(c) The `removeMinHelper` method removes the minimum value from a min-heap and returns the modified min-heap. To do so, replace the value in the root node with the smaller of its children's values and then recursively call `removeMinHelper` on the subtree rooted at the smaller child. If the min-heap consists of a single node, that node is removed and `null` is returned.

For example, the following diagram illustrates the steps to restore the heap after removing 2 from the root node.


1. Initial
2. After replacing root node with smaller child's value
3. Subtree to be used for recursive call
4. After recursive call

Write method `removeMinHelper`, which is described as follows. `removeMinHelper` should remove the minimum item from the heap and return the resulting heap.

In writing `removeMinHelper`, you may call method `smallerChild` specified in part (a). Assume that `smallerChild` works as specified, regardless of what you wrote in part (a). Solutions that reimplement functionality provided by this method, rather than invoking this method, will not receive full credit.

Complete method `removeMinHelper` below.

```
// precondition: t is not null
private TreeNode removeMinHelper(TreeNode t)
```

Suggested Solutions to Free-Response Questions

Note: There are many correct variations of these solutions.

Question 1

(a)

```
private static Queue[]
             itemsToQueues(int[] nums, int k)
{
  Queue[] queues = new Queue[10];

  for (int j = 0; j < queues.length; j++)
    queues[j] = new ListQueue();

  for (int j = 0; j < nums.length; j++)
  {
    int index = getDigit(nums[j], k);
    queues[index].enqueue(new Integer(nums[j]));
  }

  return queues;
}
```

(b)

```
private static int[] queuesToArray(
                Queue[]queues, int numVals)
{
  int index = 0;
  int[] nums = new int[numVals];

  for (int j = 0; j < queues.length; j++)
  {
    while (!queues[j].isEmpty())
{
    Integer val = (Integer) queues[j].dequeue();
    nums[index] = val.intValue();
    index++;
    }
  }

  return nums;
}
```

(c)

```
public static int[] sort(int[] nums, int numDigits)
{
  Queue[] qlist;

  for (int j = 0; j < numDigits; j++)
  {
    qlist = itemsToQueues(nums, j);
    nums = queuesToArray(qlist, nums.length);
  }

  return nums;
}
```

Question 2

(a)

```
public void addCityToMap(CityInfo theCity )
{
  Set cities = (Set) theMap.get(theCity.state());

  if (cities == null)
  {
    cities = new HashSet();
    theMap.put(theCity.state(), cities);
  }

  cities.add(theCity.city());
}
```

Commentary: Note that a `TreeSet` is not acceptable in place of a `HashSet` because it does not meet the requirements for expected running time as stated in the problem.

(b)

```
public void printOneState(String theState)
{
  System.out.print(theState);

  Set cities = (Set) theMap.get(theState);

  Iterator itr = cities.iterator();
  while (itr.hasNext())
    System.out.print(" " + itr.next());

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

Commentary: The call to `theMap.get` takes $O(\log n)$ time. Iterating through a `HashSet` takes $O(c)$ time. See the commentary on the topic outline for more information.

(c)

```
public void printAllStates()
{
  Iterator itr = theMap.keySet().iterator();

  while (itr.hasNext())
    printOneState((String) itr.next());
}
```

Question 3

(a)

```
private TreeNode smallerChild(TreeNode t)
{
  if (t == null)
    return null;
  else if (t.getLeft() == null)
    return t.getRight();
  else if (t.getRight() == null)
    return t.getLeft();
  else
  {
    Comparable left =
       (Comparable) (t.getLeft().getValue());
    Comparable right =
       (Comparable) (t.getRight().getValue());
    if (left.compareTo(right) < 0)
      return t.getLeft();
    else
      return t.getRight();
  }
}
```

(b)

```
private boolean isHeapOrdered(TreeNode t)
{
  TreeNode smaller = smallerChild(t);
  if (smaller == null)
    return true;
  else
  {
    Comparable temp = (Comparable) (t.getValue());
    return
      (temp.compareTo(smaller.getValue()) < 0)
      && isHeapOrdered(t.getLeft())
      && isHeapOrdered(t.getRight());
  }
}
```

(c)

```
// precondition: t is not null
private TreeNode removeMinHelper(TreeNode t)
{
  TreeNode smaller = smallerChild(t);
  if (smaller == null)
    return null;

  t.setValue(smaller.getValue());
  if (smaller == t.getLeft())
    t.setLeft(removeMinHelper(t.getLeft()));
  else
    t.setRight(removeMinHelper(t.getRight()));

  return t;
}
```

Appendix A

AP Computer Science Java Subset

The AP Java subset is intended to outline the features of Java that may appear on AP Computer Science Examinations. The AP Java subset is NOT intended as an overall prescription for computer science courses — the subset itself will need to be supplemented in order to cover a typical introductory curriculum. For example, I/O is essential to programming and can be done in many different ways. Because of this, specific I/O features are not tested on the AP Computer Science Exam.

This appendix describes the Java subset that students will be expected to understand when they take the AP Computer Science Exam. A number of features are also mentioned that are potentially relevant in a CS1/2 course but are not specifically tested on the AP Computer Science Exam.

The three principles that guided the formulation of the subset were as follows:

1. Enable the test designers to formulate meaningful questions
2. Help students with test preparation
3. Enable instructors to follow a variety of approaches in their courses

To help students with test preparation, the AP Java subset was intentionally kept small. Language constructs and library features were omitted that did not add significant functionality and that can, for the formulation of exam questions, be expressed by other mechanisms in the subset. For example, inner classes or the `StringBuffer` class are not essential for the formulation of exam questions — the exam uses alternatives that can be easily understood by students. Of course, these constructs add significant value for programming. Omission of a feature from the AP Java subset does not imply any judgment that the feature is inferior or not worthwhile.

The AP Java subset gives instructors flexibility in how they use Java in their courses. For example, some courses teach how to perform input/output using streams or readers/writers, others teach graphical user interface construction, and yet others rely on a tool or library that handles input/output. For the purpose of the AP Computer Science Exam, these choices are incidental and are not central for the mastery of computer science concepts. The AP Java subset does not address handling of user input at all. That means that the subset is not complete. To create actual programs, instructors need to present additional mechanisms in their classes.

The following section contains the language features that may be tested on the AP Computer Science Exam. A summary table is provided that outlines the features that are tested on the A and AB exams, the AB exam only, and those features that are useful but are not tested on either exam. A list specifying which Standard Java classes and methods will be used on the exam is available at AP Central. There will be no extra AP classes provided as part of the subset.

Language Features

1. The primitive types `int`, `double`, and `boolean` are part of the AP Java subset. The other primitive types `short`, `long`, `byte`, `char`, and `float` are not in the subset. In particular, students need not be aware that strings are composed of `char` values. Introducing `char` does not increase the expressiveness of the subset. Students already need to understand string concatenation, `String.substring`, and `String.equals.` Not introducing `char` avoids complexities with the char/int conversions and confusion between `"x"` and `'x'`.
2. Arithmetic operators: `+, -, *, /, %` are part of the AP Java subset.
3. The increment/decrement operators `++` and `--` are part of the AP Java subset. These operators are used only for their side effect, not for their value. That is, the postfix form (for example, `x++`) is always used, and the operators are not used inside other expressions. For example, `a[x++]` is not used.

4. The assignment operator = is part of the AP Java subset. The combined arithmetic/assignment operators +=, −=, `*=`, `/=`, `%=` are part of the AP Java subset although they are used simply as a shorthand and will not be used in the adjustment part of a `for` loop.

5. Relational operators ==, `!=`, <, <=, >, >= are part of the AP Java subset.

6. Logical operations `&&`, `||`, `!` are part of the AP Java subset. Students need to understand the "short circuit" evaluation of the `&&` and `||` operators. The logical `&`, `|` and `^` and the bit operators <<, >>, > > >, `&`, `~`, `|`, `^` are not in the subset.

7. The ternary `?:` operator is not in the subset.

8. The numeric casts `(int)` and `(double)` are part of the AP Java subset. Since the only primitive types in the subset are `int`, `double`, and `boolean`, the only required numeric casts are the cast `(int)` and the cast `(double)`. Students are expected to understand "truncation towards 0" behavior as well as the fact that positive floating-point numbers can be rounded to the nearest integer as `(int)(x + 0.5)`, negative numbers as `(int)(x − 0.5)`.

9. String concatenation + is part of the AP Java subset. Students are expected to know that concatenation converts numbers to strings and invokes `toString` on objects. String concatenation can be less efficient than using the `StringBuffer` class. However, for greater simplicity and conceptual clarity, the `StringBuffer` class is not in the subset.

10. The escape sequences inside strings `\\`, `\"`, `\n` are part of the AP Java subset. The `\t` escape and Unicode `\uxxxx` escapes are not in the subset. The `\'` escape is only necessary inside character literals and is not in the subset.

11. User input is not part of the AP Java subset. There are many possible ways for supplying user input; e.g., by reading from a `BufferedReader` that is wrapped around `System.in`, reading from a stream (such as a file or a URL), or from a dialog box. There are advantages and disadvantages to the various approaches. In particular, reading from `System.in` is both fraught with complexities (two nested readers and the handling of checked exceptions) and considered old fashioned by some instructors. The exam does not prescribe any one approach. Instead, if reading input is necessary, it will be indicated in a way similar to the following:

    ```
    double x = /* call to a method that
          reads a floating-point number */;
    ```

    or

    ```
    double x = IO.readDouble();
                         // read user input
    ```

    Processing string input (e.g., with `StringTokenizer`) and converting strings to numeric values (e.g., with `Integer.parseInt`) is not in the subset.

12. Testing of output is restricted to `System.out.print` and `System.out.println`. As with user input, there are many possible ways for directing the output of a program, for example to `System.out`, to a file, or to a text area in a graphical user interface. The AP Java subset includes the ability to print output to `System.out`, because it makes it easy to formulate questions. Since most graphical environments allow printing of debug messages to `System.out` (with output being collected in a special window, e.g., the "Java console" in a browser), students are usually familiar with this method of producing output. Formatted output (e.g., with `NumberFormat`) is not in the subset.

13. The `main` method and command-line arguments are not in the subset. The AP Computer Science Development Committee does not prescribe any particular approach for program invocation. In free-response questions, students are not expected to invoke programs. In case studies, program invocation with `main` may occur, but the `main` method will be kept very simple.

14. Arrays: one-dimensional arrays and two-dimensional rectangular arrays are part of the AP Java subset. Both arrays of primitive types (e.g., `int[]`) and arrays of objects (e.g., `Student[]`) are in the subset. Initialization of named arrays (`int[] arr = { 1, 2, 3 };`) is part of the AP Java subset. Two-dimensional arrays will only be tested on the AB exam. Arrays with more than two dimensions (e.g., `rubik = new Color[3][3][3]`) are not in the subset. “Ragged” arrays (e.g., `new int[3][]`) are not in the subset. In particular, students do not need to know that an `int[3][3]` really is an “array of arrays” whose rows can be replaced with other `int[]` arrays. However, students are expected to know that `arr[0].length` is the number of columns in a rectangular two-dimensional array `arr`. Anonymous arrays (e.g., `new int[] { 1, 2, 3 }`) are not in the subset.

15. The control structures `if`, `if/else`, `while`, `for`, `return` are part of the AP Java subset. The `do/while`, `switch`, plain and labeled `break` and `continue` statements are not in the subset.

16. Method overloading (e.g., `MyClass.someMethod(String str)` and `MyClass.someMethod(int num)`) is part of the AP Java subset. Students should understand that the signature of a method depends on the number, types, and order of its parameters but does not include the return type of the method.

17. Classes: Students are expected to construct objects with the `new` operator, to supply construction parameters, and to invoke accessor and modifier methods. Students are expected to modify existing classes (by adding or modifying methods and instance variables). Students are expected to design their own classes.

18. Visibility: In the AP Java subset, all classes are `public.` All instance variables are `private.` Methods, constructors, and constants (`static final` variables) are either `public` or `private.` The AP Java subset does not use `protected` and package (default) visibility.

19. The AP Java subset uses `/* */`, and `//` comments. Javadoc comments are not part of the subset.

20. The `final` keyword is only used for `final` block scope constants and `static final` class scope constants. `final` parameters or instance variables, `final` methods and `final` classes are not in the subset.

21. The concept of `static` methods is a part of the subset. Students are required to understand when the use of `static` methods is appropriate. In the exam, `static` methods are always invoked through a class, never an object (i.e., `ClassName.method(),` not `obj.method()).`

22. `static final` variables are part of the subset, other `static` variables are not.

23. The `null` reference is part of the AP Java subset.

24. The use of `this` is restricted to passing the implicit parameter in its entirety to another method (e.g., `obj.method(this)`) and to descriptions such as "the implicit parameter `this`". Using `this.var` or `this.method(args)` is not in the subset. In particular, students are not required to know the idiom "`this.var = var`", where `var` is both the name of an instance variable and a parameter variable. Calling other constructors from a constructor with the `this(args)` notation is not in the subset.

25. The use of `super` to invoke a superclass constructor (`super(args)`), or to invoke a superclass method (i.e., `super.method(args)`) is part of the AP Java subset.

26. Students are expected to implement constructors that initialize all instance variables. Class constants are initialized with an initializer: `public static final MAX_SCORE = 5;`

    The rules for default initialization (with 0, `false` or `null`) are not in the subset. Initializing instance variables with an initializer is not in the subset. Initialization blocks are not in the subset.

27. Students are expected to extend classes and implement interfaces. Students are also expected to have a knowledge of inheritance that includes understanding the concepts of method overriding and polymorphism. Students are expected to implement their own subclasses.

28. Students are expected to read the definitions of interfaces and abstract classes and understand that the abstract methods need to be redefined in non-abstract classes. Students are expected to write interfaces or class declarations when given a general description of the interface or class. On the AB exam, students are expected to define their own interfaces and abstract classes.

29. Students are expected to understand the difference between object equality (`equals`) and identity (`==`). The implementation of `equals` and `hashCode` methods are not in the subset.

30. Cloning is not in the subset, because of the complexities of implementing the `clone` method correctly and the fact that `clone` is rarely required in Java programs.

31. The `finalize` method is not in the subset.

32. Students are expected to understand that conversion from a subclass reference to a superclass reference is legal and does not require a cast. Class casts (generally from `Object` to another class) are part of the AP Java subset, to enable the use of generic collections, for example:
    `Person p = (Person) people.get(i);`

    The `instanceof` operator is not in the subset. Array type compatibility and casts between array types are not in the subset.

33. Students are expected to have a basic understanding of packages and a reading knowledge of `import` statements of the form
    `import packageName.subpackageName.ClassName;`

    `import` statements with a trailing `*`, packages and methods for locating class files (e.g., through a class path) are not in the subset.

34. Nested and inner classes are not in the subset.

35. Threads are not in the subset.

36. Students are expected to understand the exceptions that occur when their programs contain errors (in particular, `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException`, `ClassCastException`, `IllegalArgumentException`). On the AB exam, students are expected to be able to throw the unchecked `IllegalStateException` and `NoSuchElementException` in their own methods (principally when implementing collection ADTs). Checked exceptions are not in the subset. In particular, the `try/catch/finally` statements and the `throws` modifier are not in the subset.

Summary Table

Tested in A, AB exam	Tested in AB exam only	Potentially relevant to CS1/CS2 course but not tested
`int`, `double`, `boolean`		`short`, `long`, `byte`, `char`, `float`
`+`, `-`, `*`, `/`, `%`, `++`, `--` `=`, `+=`, `-=`, `*=`, `/=`, `%=` `==`, `!=`, `<`, `<=`, `>`, `>=`		Using the values of `++`, `--` expressions in other expressions
`&&`, `||`, `!` and short-circuit evaluation		`<<`, `>>`, `>>>`, `&`, `~`, `|`, `^`, `?:`
`(int)`, `(double)`		Other numeric casts such as `(char)` or `(float)`
String concatenation		`StringBuffer`
Escape sequences `\"`, `\\`, `\n` inside strings		Other escape sequences (`\'`, `\t`, `\unnnn`)
`System.out.print`, `System.out.println`		`System.in`, Stream input/output, GUI input/output, parsing input, formatted output

Tested in A, AB exam	Tested in AB exam only	Potentially relevant to CS1/CS2 course but not tested
		`public static void main(String[] args)`
1-dimensional arrays	2-dimensional rectangular arrays	Arrays with 3 or more dimensions, ragged arrays
`if`, `if/else`, `while`, `for`, `return`		`do/while`, `switch`, `break`, `continue`
Modify existing classes, design classes		
`public` classes, `private` instance variables, `public` or `private` methods or constants		`protected` or package visibility
		`javadoc`
`static final` class variables		`final local` variables, `final` parameter variables, instance variables, methods or classes
`static` methods		`static` non-`final` variables
`null`, `this`, `super`, `super.method(args)`		`this.var`, `this.method(args)`, `this(args)`

Tested in A, AB exam	Tested in AB exam only	Potentially relevant to CS1/CS2 course but not tested
Constructors and initialization of `static` variables		Default initialization of instance variables, initialization blocks
Understand inheritance hierarchies. Design and implement subclasses. Modify subclass implementations and implementations of interfaces.		
Understand the concepts of abstract classes and interfaces. Design an interface.	Design and implement abstract classes	
Understand `equals`, `==`, and `!=` comparison of objects `Comparable.compareTo`		`clone`, implementation of `equals`
Conversion to supertypes and `(Subtype)` casts		`instanceof`
		Nested classes, inner classes
Package concept, `import packageName.className;`		`import packageName.*` defining packages, class path

Tested in A, AB exam	Tested in AB exam only	Potentially relevant to CS1/CS2 course but not tested
Exception concept, common exceptions	Throwing standard unchecked exceptions	Checked exceptions `try/catch/finally`, `throws`
`Comparable`, `String`, `Math`, `Random`, `Object`, `ArrayList`	`List`, `Set`, `Map`, `Iterator`, `ListIterator`, `LinkedList`, `HashSet`, `TreeSet`, `HashMap`, `TreeMap`	
Wrapper classes (`Integer`, `Double`)		
		Sorting methods in `Arrays` and `Collections`

Appendix B

Standard Java Library Methods Required for APCS A

`class java.lang.Object`
- `boolean equals(Object other)`
- `String toString()`

`interface java.lang.Comparable`
- `int compareTo(Object other)`

    // return value < 0 if `this` is less than `other`
    // return value = 0 if `this` is equal to `other`
    // return value > 0 if `this` is greater than `other`

`class java.lang.Integer`
`        implements java.lang.Comparable`
- `Integer(int value)`   // constructor
- `int intValue()`
- `boolean equals(Object other)`
- `String toString()`
- `int compareTo(Object other)`

    `// specified by java.lang.Comparable`

`class java.lang.Double`
`        implements java.lang.Comparable`
- `Double(double value)`   // constructor
- `double doubleValue()`
- `boolean equals(Object other)`
- `String toString()`
- `int compareTo(Object other)`

    `// specified by java.lang.Comparable`

`class java.lang.String`
`        implements java.lang.Comparable`
- `int compareTo(Object other)`

    `// specified by java.lang.Comparable`
- `boolean equals(Object other)`
- `int length()`

• `String substring(int from, int to)`
  `//` returns the substring beginning at `from`
  `//` and ending at `to-1`
• `String substring(int from)`
  `//` returns `substring(from, length())`
• `int indexOf(String s)`
  `//` returns the index of the first occurrence of `s;`
  `//` returns −1 if not found

`class java.lang.Math`
• `static int abs(int x)`
• `static double abs(double x)`
• `static double pow(double base, double exponent)`
• `static double sqrt(double x)`

`class java.util.Random`
• `int nextInt(int n)`
  `//` returns an integer in the range from 0 to `n-1` inclusive
• `double nextDouble()`

`class java.util.ArrayList`
• `int size()`
• `boolean add(Object x)`
• `Object get(int index)`
• `Object set(int index, Object x)`
  `//` replaces the element at index with `x`
  `//` returns the element formerly at the specified position
• `void add(int index, Object x)`
  `//` inserts `x` at position `index,` sliding elements
  `//` at position `index` and higher to the right
  `//` (adds 1 to their indices) and adjusts size
• `Object remove(int index)`
  `//` removes element from position `index`, sliding
  `//` elements at position index + 1 and higher to the
  `//` left (subtracts 1 from their indices) and adjusts size

Appendix C

Standard Java Library Methods Required for APCS AB

`class java.lang.Object`
- `boolean equals(Object other)`
- `String toString()`
- `int hashCode()`

`interface java.lang.Comparable`
- `int compareTo(Object other)`
  - // return value < 0 if `this` is less than `other`
  - // return value $= 0$ if `this` is equal to `other`
  - // return value > 0 if `this` is greater than `other`

`class java.lang.Integer`
`        implements java.lang.Comparable`
- `Integer(int value)` // constructor
- `int intValue()`
- `boolean equals(Object other)`
- `String toString()`
- `int compareTo(Object other)`
  // specified by `java.lang.Comparable`

`class java.lang.Double`
`        implements java.lang.Comparable`
- `Double(double value)` // constructor
- `double doubleValue()`
- `boolean equals(Object other)`
- `String toString()`
- `int compareTo(Object other)`
  // specified by `java.lang.Comparable`

class java.lang.String
        implements java.lang.Comparable

- int compareTo(Object other)
    // specified by java.lang.Comparable
- boolean equals(Object other)
- int length()
- String substring(int from, int to)
    // returns the substring beginning at from
    // and ending at to-1
- String substring(int from)
    // returns substring(from, length())
- int indexOf(String s)
    // returns the index of the first occurrence of s;
    // returns -1 if not found

class java.lang.Math

- static int abs(int x)
- static double abs(double x)
- static double pow(double base,
                      double exponent)
- static double sqrt(double x)

class java.util.Random

- int nextInt(int n)
    // returns an integer in the range from 0 to n-1 inclusive
- double nextDouble()

interface java.util.List

- boolean add(Object x)
- int size()
- Object get(int index)
- Object set(int index, Object x)
    // replaces the element at index with x
    // returns the element formerly at the specified position
- Iterator iterator()
- ListIterator listIterator()

`class java.util.ArrayList`
`        implements java.util.List`

- Methods in addition to the `List` methods:
- `void add(int index, Object x)`
  - `//` inserts `x` at position `index`, sliding elements
  - `//` at position `index` and higher to the right
  - `//` (adds 1 to their indices) and adjusts size
- `Object remove(int index)`
  - `//` removes element from position `index`, sliding
  - `//` elements at position `index` + 1 and higher to the
  - `//` left (subtracts 1 from their indices) and adjusts size

`class java.util.LinkedList`
`        implements java.util.List`

- Methods in addition to the `List` methods:
- `void addFirst(Object x)`
- `void addLast(Object x)`
- `Object getFirst()`
- `Object getLast()`
- `Object removeFirst()`
- `Object removeLast()`

`interface java.util.Set`

- `boolean add(Object x)`
- `boolean contains(Object x)`
- `boolean remove(Object x)`
- `int size()`
- `Iterator iterator()`

`class java.util.HashSet`
`        implements java.util.Set`
`class java.util.TreeSet`
`        implements java.util.Set`

`interface java.util.Map`
- `Object put(Object key, Object value)`
- `Object get(Object key)`
- `Object remove(Object key)`
- `boolean containsKey(Object key)`
- `int size()`
- `Set keySet()`

```
class java.util.HashMap
        implements java.util.Map
class java.util.TreeMap
        implements java.util.Map
```

`interface java.util.Iterator`
- `boolean hasNext()`
- `Object next()`
- `void remove()`

```
interface java.util.ListIterator
            extends java.util.Iterator
```
- Methods in addition to the `Iterator` methods
- `void add(Object x)`
- `void set(Object x)`

Appendix D

Implementation classes for linked list and tree nodes (APCS AB)

Unless otherwise noted, assume that a linked list implemented from the `ListNode` class does not have a dummy header node.

```
public class ListNode
{
  private Object value;
  private ListNode next;

  public ListNode(Object initValue,
                  ListNode initNext)
    { value = initValue; next = initNext; }

  public Object getValue() { return value; }

  public ListNode getNext() { return next; }

  public void setValue(Object theNewValue)
    { value = theNewValue; }

  public void setNext(ListNode theNewNext)
    { next = theNewNext; }
}
```

Unless otherwise noted, assume that a tree implemented from the `TreeNode` class does not have a dummy root node.

```
public class TreeNode
{
  private Object value;
  private TreeNode left;
  private TreeNode right;

  public TreeNode(Object initValue)
    { value = initValue;
      left = null; right = null; }

  public TreeNode(Object initValue,
                  TreeNode initLeft,
                  TreeNode initRight)
    { value = initValue;
      left = initLeft; right = initRight; }

  public Object getValue() { return value; }

  public TreeNode getLeft() { return left; }

  public TreeNode getRight() { return right; }

  public void setValue(Object theNewValue)
    { value = theNewValue; }

  public void setLeft(TreeNode theNewLeft)
    { left = theNewLeft; }

  public void setRight(TreeNode theNewRight)
    { right = theNewRight; }
}
```

Appendix E

Interfaces for stacks, queues, and priority queues (APCS AB)

`Interface for stacks`

```
(* See note at end of reference)

public interface Stack
{
  /**
   * postcondition:
   *   returns true if stack is empty;
   *   otherwise, returns false
   */
  boolean isEmpty();

  /**
   * precondition:
   *   stack is [e1, e2, ..., en] with n >= 0
   * postcondition:
   *   stack is [e1, e2, ..., en, x]
   */
  void push(Object x);

  /**
   * precondition:
   *   stack is [e1, e2, ..., en] with n >= 1
   * postcondition:
   *   stack is [e1, e2, ..., e(n-1)];
   *   returns en
   * exceptions:
   *   throws an unchecked exception if the
   *   stack is empty
   */
  Object pop();
  /**
   * precondition:
   *   stack is [e1, e2, ..., en] with n >= 1
   * postcondition:
   *   returns en
   * exceptions:
   *   throws an unchecked exception if the
   *   stack is empty
   */
  Object peekTop();
}
```

Interface for queues
(* See note at end of reference)

```
public interface Queue
{
  /**
   * postcondition:
   *   returns true if queue is empty;
   *   otherwise, returns false
   */
  boolean isEmpty();

  /**
   * precondition:
   *   queue is [e1, e2, ..., en] with n >= 0
   * postcondition:
   *   queue is [e1, e2, ..., en, x]
   */
  void enqueue(Object x);
  /**
   * precondition:
   *   queue is [e1, e2, . . ., en] with n >= 1
   * postcondition:
   *   queue is [e2, . . ., en]; returns e1
   * exceptions:
   *   throws an unchecked exception if the
   *   queue is empty
   */
  Object dequeue();

  /**
   * precondition:
   *   queue is [e1, e2, . . ., en] with n >= 1
   * postcondition:
   *   returns e1
   * exceptions:
   *   throws an unchecked exception if the
   *   queue is empty
   */
  Object peekFront();
}
```

Interface for priority queues
(* See note at end of reference)

```
public interface PriorityQueue
{
  /**
   * postcondition:
   *   returns true if the number of elements
   *   in the priority queue is 0;
   *   otherwise, returns false
   */
  boolean isEmpty();
  /**
   * postcondition:
   *   x has been added to the priority queue;
   *   the number of elements in the priority
   *   queue is increased by 1.
   */
  void add(Object x);

  /**
   * postcondition:
   *   The smallest item in the priority queue
   *   is removed and returned;
   *   the number of elements in the
   *   priority queue is decreased by 1.
   * exceptions:
   *   throws unchecked exception if
   *   priority queue is empty
   */
  Object removeMin();

  /**
   * postcondition:
   *   The smallest item in the priority queue
   *   is returned;
   *   the priority queue is unchanged
   * exceptions:
   *   throws unchecked exception if
   *   priority queue is empty
   */
  Object peekMin();
}
```

*** Note regarding use of stacks, queues, and priority queues**

When a stack, queue, or priority queue object needs to be instantiated, code such as the following is used:

```
Queue q = new ListQueue();
           // ListQueue implements Queue
```

AP® Program Essentials

The AP Reading

In June, the free-response sections of the exams, as well as the Studio Art portfolios, are scored by college faculty and secondary school AP teachers at the AP Reading. Thousands of readers participate, under the direction of a Chief Reader in each field. The experience offers both significant professional development and the opportunity to network with like-minded educators.

If you are an AP teacher or a college faculty member and would like to serve as a reader, you can visit AP Central for more information on how to apply. Alternatively, send an e-mail message to apreader@ets.org, or call Performance Scoring Services at 609 406-5383.

AP Grades

The readers' scores on the essay and problem-solving questions are combined with the results of the computer-scored multiple-choice questions, and the total raw scores are converted to AP's 5-point scale:

AP GRADE	QUALIFICATION
5	Extremely well qualified
4	Well qualified
3	Qualified
2	Possibly qualified
1	No recommendation

Grade Distributions

Many teachers want to compare their students' grades with the national percentiles. Grade distribution charts are available at AP Central, as is information on how the cut-off points for each AP grade are calculated. Grade distribution charts are also available on the AP student site at www.collegeboard.com/apstudents.

Earning College Credit and/or Placement

Credit, advanced placement, or both are awarded by the college or university, not the College Board or the AP Program. The best source of specific and up-to-date information about an individual institution's policy is its catalog or Web site.

Why Colleges Grant Credit and/or Placement for AP Grades

Colleges know that the AP grades of their incoming students represent a level of achievement equivalent to that of students who take the same course in the colleges' own classrooms. That equivalency is assured through several Advanced Placement Program processes:

- College faculty serve on the committees that develop the course descriptions and examinations in each AP subject.
- College faculty are responsible for standard setting and are involved in the evaluation of student responses at the AP Reading.
- AP courses and exams are updated regularly, based on both the results of curriculum surveys at up to 200 colleges and universities and the interactions of committee members with professional organizations in their discipline.
- College comparability studies are undertaken in which the performance of college students on AP Exams is compared with that of AP students to confirm that the AP grade scale of 1–5 is properly aligned with current college standards.

In addition, the College Board has commissioned studies that use a "bottom-line" approach to validating AP Exam grades by comparing the achievement of AP versus non-AP students in higher-level college courses. For example, in the 1998 Morgan and Ramist "21-College" study, AP students who were exempted from introductory courses and who completed a higher-level course in college were compared favorably, on the basis of their college grades, with students who completed the prerequisite first course in college, then took the second, higher-level course in the subject area. Such studies answer the question of greatest concern to colleges — are AP students who are exempted from introductory courses as well prepared to continue in a subject area as students who took their first course in college? To see the results of several college validity studies, go to AP Central. (The Morgan and Ramist study can be downloaded from the site in its entirety.)

Guidelines on Granting Credit and/or Placement for AP Grades

If you are an admissions administrator and need guidance on setting an AP policy for your college or university, you will find the *College and University Guide to the Advanced Placement Program* useful; see the back of this booklet for ordering information. Alternatively, contact your local College Board office, as noted on the inside back cover of this Course Description.

Finding Colleges That Accept AP Grades

In addition to contacting colleges directly for their AP policies, students and teachers can use College Search, an online resource maintained by the College Board through its Annual Survey of Colleges. College Search can be accessed via the College Board's Web site (www.collegeboard.com). It is worth remembering that policies are subject to change. Contact the college directly to get the most up-to-date information.

AP Awards

The AP Program offers a number of awards to recognize high school students who have demonstrated college-level achievement through AP courses and exams. Although there is no monetary award, in addition to an award certificate, student achievement is acknowledged on any grade report sent to colleges following the announcement of the awards. For detailed information on AP Awards, including qualification criteria, visit AP Central or contact the College Board's National Office. Students can find this information at www.collegeboard.com/apstudents.

AP Calendar

The *AP Program Guide* and the *Bulletin for AP Students and Parents* provide education professionals and students, respectively, with information on the various events associated with the AP year. Information on ordering and downloading these publications can be found at the back of this booklet.

Test Security

The entire AP Exam must be kept secure at all times. Forty-eight hours after the exam has been administered, the green and blue inserts containing the free-response questions (Section II) can be made available for teacher and student review.* **However, the multiple-choice section (Section I) MUST remain secure both before and after the exam administration.** No one other than students taking the exam can ever have access to or see the questions contained in Section 1 — this includes AP Coordinators and all teachers. The multiple-choice section must never be shared, copied in any manner, or reconstructed by teachers and students after the exam.

---

*The alternate form of the free-response section (used for late testing administration) is NOT released.

Selected multiple-choice questions are reused from year to year to provide an essential method of establishing high exam reliability, controlled levels of difficulty, and comparability with earlier exams. These goals can be attained only when the multiple-choice questions remain secure. This is why teachers cannot view the questions and students cannot share information about these questions with anyone following the exam administration.

To ensure that all students have an equal opportunity to demonstrate their abilities on the exam, AP Exams must be administered in a uniform manner. **It is extremely important to follow the administration schedule and all procedures outlined in detail in the most recent *AP Coordinator's Manual*.** Please note that Studio Art portfolios and their contents are not considered secure testing materials; see the *AP Coordinator's Manual* for further information. The manual also includes directions on how to deal with misconduct and other security problems. Any breach of security should be reported to Test Security immediately (call 800 353-8570, fax 609 406-9709, or e-mail tsreturns@ets.org).

Teacher Support

You can find the following Web resources at AP Central:

- Teachers' Resources (reviews of classroom resources).
- Institutes & Workshops (a searchable database of professional development opportunities).
- The most up-to-date and comprehensive information on AP courses, exams, and other Program resources.
- The opportunity to exchange teaching methods and materials with the international AP community using electronic discussion groups (EDGs).
- An electronic library of AP publications, including released exam questions, the *AP Coordinator's Manual*, Course Descriptions, and sample syllabi.
- Opportunities for professional involvement in the AP Program.
- Information about state and federal support for the AP Program.
- AP Program data, research, and statistics.
- FAQs about the AP Program.
- Current news and features about the AP Program, its courses and teachers.

AP teachers can also use a number of AP publications, CD-ROMs, and videos that supplement these Web resources. Please see the following pages for an overview and ordering information.

Pre-AP®

Pre-AP® is a suite of K–12 professional development resources and services to equip middle and high school teachers with the strategies and tools they need to engage their students in high-level learning, thereby ensuring that every middle and high school student has the depth and understanding of the skills, habits of mind, and concepts they need to succeed in college.

Pre-AP rests upon a profound hope and heartfelt esteem for teachers and students. Conceptually, Pre-AP is based on two important premises. The first is the expectation that all students can perform at rigorous academic levels. This expectation should be reflected in curriculum and instruction throughout the school such that all students are consistently being challenged to expand their knowledge and skills to the next level.

The second is the belief that we can prepare every student for higher intellectual engagement by starting the development of skills and acquisition of knowledge as early as possible. Addressed effectively, the middle and high school years can provide a powerful opportunity to help all students acquire the knowledge, concepts, and skills needed to engage in a higher level of learning.

Since Pre-AP teacher professional development supports explicitly the goal of college as an option for every student, it is important to have a recognized standard for college-level academic work. The Advanced Placement Program (AP) provides these standards for Pre-AP. Pre-AP teacher professional development resources reflect topics, concepts, and skills found in AP courses.

The College Board does not design, develop, or assess courses labeled "Pre-AP." Courses labeled "Pre-AP" that inappropriately restrict access to AP and other college-level work are inconsistent with the fundamental purpose of the Pre-AP initiatives of the College Board. We encourage schools, districts, and policymakers to utilize Pre-AP professional development in a manner that ensures equitable access to rigorous academic experiences for all students.

Pre-AP Professional Development

Pre-AP professional development is administered by Pre-AP Initiatives, a unit in K–12 Professional Development, and is available through workshops and conferences coordinated by the regional offices of the College Board. Pre-AP professional development is divided into two categories:

1. **Articulation of content and pedagogy across the middle and high school years —** The emphasis of professional development in this category is aligning curriculum and improving teacher communication. The intended outcome from articulation is a coordinated program of teaching skills and concepts over several years.
2. **Classroom strategies for middle and high school teachers —** Various approaches, techniques, and ideas are emphasized in professional development in the category.

For a complete list of Pre-AP Professional Development offerings, please contact your regional office or visit AP Central at apcentral.collegeboard.com.

AP Publications and Other Resources

A number of AP resources are available to help students, parents, AP Coordinators, and high school and college faculty learn more about the AP Program and its courses and exams. To identify resources that may be of particular use to you, refer to the following key.

AP Coordinators and Administrators	**A**
College Faculty	**C**
Students and Parents	**SP**
Teachers	**T**

Ordering Information

You have several options for ordering publications:

- **Online.** Visit the College Board store at store.collegeboard.com.
- **By mail.** Send a completed order form with your payment or credit card information to: Advanced Placement Program, Dept. E-06, P. O. Box 6670, Princeton, NJ 08541-6670. If you need a copy of the order form, you can download one from AP Central.

- **By fax.** Credit card orders can be faxed to AP Order Services at 609 771-7385.
- **By phone.** Call AP Order Services at 609 771-7243, Monday through Friday, 8:00 a.m. to 9:00 p.m. ET. Have your American Express, Discover, JCB, MasterCard, or VISA information ready. This phone number is for credit card orders only.

Payment must accompany all orders not on an institutional purchase order or credit card, and checks should be made payable to the College Board. The College Board pays UPS ground rate postage (or its equivalent) on all prepaid orders; delivery generally takes two to three weeks. Please do not use P.O. Box numbers. Postage will be charged on all orders requiring billing and/or requesting a faster method of delivery.

Publications may be returned for a full refund if they are returned within 30 days of invoice. Software and videos may be exchanged within 30 days if they are opened, or returned for a full refund if they are unopened. No collect or C.O.D. shipments are accepted. Unless otherwise specified, orders will be filled with the currently available edition; prices and discounts are subject to change without notice.

In compliance with Canadian law, all AP publications delivered to Canada incur the 7 percent GST. The GST registration number is 13141 4468 RT. Some Canadian schools are exempt from paying the GST. Appropriate proof of exemption must be provided when AP publications are ordered so that tax is not applied to the billing statement.

Print

Items marked with a computer mouse icon can be downloaded for free from AP Central.

Bulletin for AP Students and Parents **SP**

This bulletin provides a general description of the AP Program, including how to register for AP courses, and information on the policies and procedures related to taking the exams. It describes each AP Exam, lists the advantages of taking the exams, describes the grade reporting process, and includes the upcoming exam schedule. The *Bulletin* is available in both English and Spanish.

AP Program Guide **A**

This guide takes the AP Coordinator step-by-step through the school year — from organizing an AP program, through ordering and administering the AP Exams, payment, and grade reporting. It also includes information on teacher professional development, AP resources, and exam schedules. The *AP Program Guide* is sent automatically to all schools that register to participate in AP.

College and University Guide to the AP Program **C, A**

This guide is intended to help college and university faculty and administrators understand the benefits of having a coherent, equitable AP policy. Topics included are validity of AP grades; developing and maintaining scoring standards; ensuring equivalent achievement; state legislation supporting AP; and quantitative profiles of AP students by each AP subject.

Course Descriptions **SP, T, A, C**

Course Descriptions provide an outline of the AP course content, explain the kinds of skills students are expected to demonstrate in the corresponding introductory college-level course, and describe the AP Exam. They also provide sample multiple-choice questions with an answer key, as well as sample free-response questions. Note: The Course Description for AP Computer Science is available in electronic format only.

Pre-AP **A, T**

This brochure describes the Pre-AP concept and the professional development opportunities available to middle school and high school teachers.

Released Exams **T**

About every four to five years, on a rotating schedule, the AP Program releases a complete copy of each exam. In addition to providing the multiple-choice questions and answers, the publication describes the process of scoring the free-response questions and includes examples of students' actual responses, the scoring guidelines, and commentary that explains why the responses received the scores they did.

Teacher's Guides **T**

For those about to teach an AP course for the first time, or for experienced AP teachers who would like to get some fresh ideas for the classroom, the Teacher's Guide is an excellent resource. Each Teacher's Guide contains syllabi developed by high school teachers currently teaching the

AP course and college faculty who teach the equivalent course at colleges and universities. Along with detailed course outlines and innovative teaching tips, you'll also find extensive lists of suggested teaching resources.

AP Vertical Team Guides **T, A**

An AP Vertical Team (APVT) is made up of teachers from different grade levels who work together to develop and implement a sequential curriculum in a given discipline. The team's goal is to help students acquire the skills necessary for success in AP. To help teachers and administrators who are interested in establishing an APVT at their school, the College Board has published these guides: *A Guide for Advanced Placement English Vertical Teams*; *Advanced Placement Program Mathematics Vertical Teams Toolkit*; *AP Vertical Teams in Science*, *Social Studies*, *Foreign Language*, *Studio Art*, *and Music Theory: An Introduction*; *AP Vertical Teams Guide for Social Studies; AP Vertical Teams Guide for Fine Arts*, *Vol.1: Studio Art*; *AP Vertical Teams Guide for Fine Arts*, *Vol. 2: Music Theory*; and *AP Vertical Teams Guide for Fine Arts*, *Vol.1 and 2* (set).

Multimedia

APCD® (home version),
(multi-network site license) **SP, T**

These CD-ROMs are available for Calculus AB, English Language, English Literature, European History, Spanish Language, and U.S. History. They each include actual AP Exams, interactive tutorials, and other features, including exam descriptions, answers to frequently asked questions, study-skill suggestions, and test-taking strategies. There is also a listing of resources for further study and a planner to help students schedule and organize their study time.

The teacher version of each CD, which can be licensed for up to 50 workstations, enables you to monitor student progress and provide individual feedback. Included is a Teacher's Manual that gives full explanations along with suggestions for utilizing the APCD in the classroom.

College Board Offices

National Office
45 Columbus Avenue, New York, NY 10023-6992
212 713-8066
E-mail: ap@collegeboard.org

Middle States
Serving Delaware, District of Columbia, Maryland, New Jersey, New York, Pennsylvania, and Puerto Rico
Two Bala Plaza, Suite 900, Bala Cynwyd, PA 19004-1501
610 670-4400
E-mail: msro@collegeboard.org

Midwestern
Serving Illinois, Indiana, Iowa, Kansas, Michigan, Minnesota, Missouri, Nebraska, North Dakota, Ohio, South Dakota, West Virginia, and Wisconsin
1560 Sherman Avenue, Suite 1001, Evanston, IL 60201-4805
847 866-1700
E-mail: mro@collegeboard.org

New England
Serving Connecticut, Maine, Massachusetts, New Hampshire, Rhode Island, and Vermont
470 Totten Pond Road, Waltham, MA 02451-1982
781 890-9150
E-mail: nero@collegeboard.org

Southern
Serving Alabama, Florida, Georgia, Kentucky, Louisiana, Mississippi, North Carolina, South Carolina, Tennessee, and Virginia
3700 Crestwood Parkway, Suite 700, Duluth, GA 30096-5599
678 380-3300
E-mail: sro@collegeboard.org

Southwestern
Serving Arkansas, New Mexico, Oklahoma, and Texas
4330 South MoPac Expressway, Suite 200, Austin, TX 78735-6734
512 891-8400
E-mail: swro@collegeboard.org

Western
Serving Alaska, Arizona, California, Colorado, Hawaii, Idaho, Montana, Nevada, Oregon, Utah, Washington, and Wyoming
2099 Gateway Place, Suite 480, San Jose, CA 95110-1048
408 452-1400
E-mail: wro@collegeboard.org

Dallas Metroplex Office
Box 19666, 600 South West Street, Suite 108, Arlington, TX 76019
817 272-7200
E-mail: kwilson@collegeboard.org

Canada
1708 Dolphin Avenue, Suite 406, Kelowna, BC, Canada V1Y 9S4
250 861-9050; 800 667-4548 in Canada only
E-mail: gewonus@collegeboard.org

International
Serving all countries outside the United States and Canada
45 Columbus Avenue, New York, NY 10023-6992
212 713-8091
E-mail: apintl@collegeboard.org

2002-03 Development Committee and Chief Reader

apcentral.collegeboard.com

I.N. 997134