Program Design and Analysis

CHAPTER 5

Weeks of coding can save you hours of planning.
—Anonymous

Chapter Goals

- Program development, including design and testing
- Object-oriented program design
- Relationships between classes
- Program analysis
- Big-O notation
- Loop invariants

Students of introductory computer science typically see themselves as programmers. They no sooner have a new programming project in their heads than they're at the computer, typing madly to get some code up and running. (Is this you?)

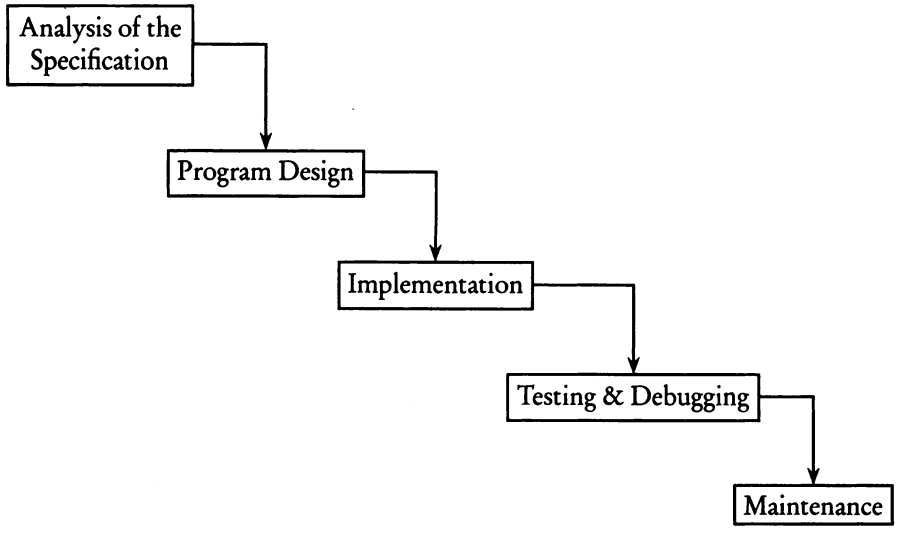
To succeed as a programmer, however, you have to combine the practical skills of a software engineer with the analytical mindset of a computer scientist. A software engineer oversees the life cycle of software development: initiation of the project, analysis of the specification, and design of the program, as well as implementation, testing, and maintenance of the final product. A computer scientist (among other things!) analyzes the implementation, correctness, and efficiency of algorithms. All these topics are tested on the APCS exam.

THE SOFTWARE DEVELOPMENT LIFE CYCLE

The Waterfall Model

The waterfall model of software development came about in the 1960s in order to bring structure and efficiency into the process of creating large programs.

Each step in the process flows into the next: The picture resembles a waterfall.



Program Specification

The *specification* is a written description of the project. Typically it is based on a customer's requirements. The first step in writing a program is to analyze the specification, make sure you understand it, and clarify with the customer anything that is unclear.

Program Design

Even for a small-scale program a good design can save programming time and enhance the reliability of the final program. The design is a fairly detailed plan for solving the problem outlined in the specification. It should include all objects that will be used in the solution, the data structures that will implement them, plus a detailed list of the tasks to be performed by the program.

A good design provides a fairly detailed overall plan at a glance, without including the minutiae of Java code.

Program Implementation

Program implementation is the coding phase. Design and implementation are discussed in more detail on p. 263.

Testing and Debugging

TEST DATA

Not every possible input value can be tested, so a programmer should be diligent in selecting a representative set of *test data*. Typical values in each part of a domain of the program should be selected, as well as endpoint values and out-of-range values. If only positive input is required, your test data should include a negative value just to check that your program handles it appropriately.

Example

A program must be written to insert a value into its correct position in this sorted list:

```
2  5  9
```

Test data should include

- A value less than 2
- A value between 2 and 5
- A value between 5 and 9
- A value greater than 9
- 2, 5, and 9
- A negative value

TYPES OF ERRORS (BUGS)

- A *compile-time error* occurs during compilation of the program. The compiler is unable to translate the program into bytecode and prints an appropriate error message. A *syntax error* is a compile-time error caused by violating the rules of the programming language. Examples include omitting semicolons or braces, using undeclared identifiers, using keywords inappropriately, having parameters that don't match in type and number, and invoking a method for an object whose class definition doesn't contain that method.
- A *run-time error* occurs during execution of the program. The Java run-time environment *throws an exception*, which means that it stops execution and prints an error message. Typical causes of run-time errors include attempting to divide an integer by zero, using an array index that is out of bounds, attempting to open a file that cannot be found, and so on. An error that causes a program to run forever ("infinite loop") can also be regarded as a run-time error. (See also Errors and Exceptions, p. 132.)
- An *intent* or *logic error* is one that fails to carry out the specification of the program. The program compiles and runs but does not do the job. These are sometimes the hardest types of errors to fix.

ROBUSTNESS

Always assume that any user of your program is not as smart as you are. You must therefore aim to write a *robust* program, namely one that

- Won't give inaccurate answers for some input data.
- Won't crash if the input data are invalid.
- Won't allow execution to proceed if invalid data are entered.

Examples of bad input data include out-of-range numbers, characters instead of numerical data, and a response of "maybe" when "yes" or "no" was asked for.

Note that bad input data that invalidates a computation won't be detected by Java. Your program should include code that catches the error, allows the error to be fixed, and allows program execution to resume.

Program Maintenance

Program maintenance involves upgrading the code as circumstances change. New features may be added. New programmers may come on board. To make their task easier, the original program must have clear and precise documentation.

OBJECT-ORIENTED PROGRAM DESIGN

Object-oriented programming has been the dominant programming methodology since the mid 1990s. It uses an approach that blurs the lines of the waterfall model. Analysis of the problem, development of the design, and pieces of the implementation all overlap and influence one another.

Here are the steps in object-oriented design:

- Identify classes to be written.
- Identify behaviors (i.e., methods) for each class.
- Determine the relationships between classes.
- Write the interface (public method headers) for each class.
- Implement the methods.

Identifying Classes

Identify the objects in the program by picking out the nouns in the program specification. Ignore pronouns and nouns that refer to the user. Select those nouns that seem suitable as classes, the "big-picture" nouns that describe the major objects in the application. Some of the other nouns may end up as attributes of the classes.

Identifying Behaviors

Find all verbs in the program description that help lead to the solution of the programming task. These are likely behaviors that will probably become the methods of the classes.

Encapsulation

Now decide which methods belong in which classes. Recall that the process of bundling a group of methods and data fields into a class is called *encapsulation*.

You will also need to decide which data fields each class will need and which data structures should store them. For example, if an object represents a list of items, consider an array or `ArrayList` as the data structure.

Determining Relationships Between Classes

INHERITANCE RELATIONSHIPS

Look for classes with common behaviors. This will help identify *inheritance relationships*. Recall the *is-a* relationship—if `object1` *is-a* `object2`, then `object2` is a candidate for a superclass.

COMPOSITION RELATIONSHIPS

Composition relationships are defined by the *has-a* relationship. For example, a `Nurse` *has-a* `Uniform`. Typically, if two classes have a composition relationship, one of them contains an instance variable whose type is the other class.

Note that a wrapper class always implements a *has-a* relationship with any objects that it wraps.

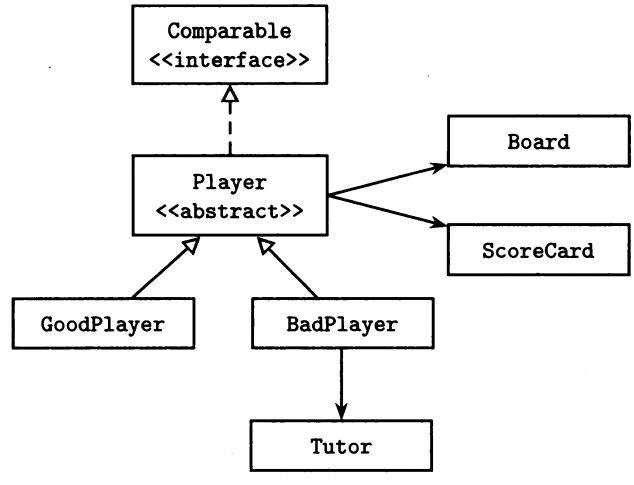
UML Diagrams

An excellent way to keep track of the relationships between classes and show the inheritance hierarchy in your programs is with a UML (Unified Modeling Language) diagram. This is a standard graphical scheme used by object-oriented programmers. Although it is not part of the AP subset, on the AP exam you may be expected to interpret simple UML diagrams and inheritance hierarchies.

Here is a simplified version of the UML rules:

- Represent classes with rectangles.
- Use angle brackets with the word "abstract" or "interface" to indicate either an abstract class or interface.
- Show the *is-a* relationship between classes with an open up-arrow.
- Show the *is-a* relationship that involves an interface with an open, dotted up-arrow.
- Show the *has-a* relationship with a down arrow (indicates composition).

Example



From this diagram you can see at a glance that `GoodPlayer` and `BadPlayer` are subclasses of an abstract class `Player`, and that each `Player` implements the `Comparable` interface. Every `Player` has a `Board` and a `ScoreCard`, while only the `BadPlayer` has a `Tutor`.

Implementing Classes

BOTTOM-UP DEVELOPMENT

For each method in a class, list all of the other classes needed to implement that particular method. These classes are called *collaborators*. A class that has no collaborators is *independent*.

To implement the classes, often an incremental, *bottom-up* approach is used. This means that independent classes are fully implemented and tested before being incorporated into the overall project. These unrelated classes can be implemented by different programmers.

Note that a class can be tested using a dummy `Tester` class that will be discarded when the methods of the class are working. Constructors, then methods, should be added, and tested, one at a time. A *driver class* that contains a `main` method can be used to test the program as you go. The purpose of the driver is to test the class fully before incorporating it as an object in a new class.

When each of the independent classes is working, classes that depend on just one other class are implemented and tested, and so on. This may lead to a working, bare bones version of the project. New features and enhancements can be added later.

Design flaws can be corrected at each stage of development. Remember, a design is never set in stone: It simply guides the implementation.

TOP-DOWN DEVELOPMENT

In a top-down design, the programmer starts with an overview of the program, selecting the highest-level controlling object and the tasks needed. During development of the program, subsidiary classes may be added to simplify existing classes.

Implementing Methods

PROCEDURAL ABSTRACTION

A good programmer avoids chunks of repeated code wherever possible. To this end, if several methods in a class require the same task, like a search or a swap, you should use *helper methods*. The `reduce` method in the `Rational` class on p. 172 is an example of such a method. Also, wherever possible you should enhance the readability of your code by using helper methods to break long methods into smaller tasks. The use of helper methods within a class is known as *procedural abstraction* and is an example of top-down development within a class. This process of breaking a long method into a sequence of smaller tasks is sometimes called *stepwise refinement*.

INFORMATION HIDING

Instance variables and helper methods are generally declared as `private`, which prevents client classes from accessing them. This strategy is called *information hiding*.

STUB METHOD

Sometimes it makes more sense in the development of a class to test a calling method before testing a method it invokes. A *stub* is a dummy method that stands in for a method until the actual method has been written and tested. A stub typically has an output statement to show that it was called in the correct place, or it may return some reasonable values if necessary.

ALGORITHM

An *algorithm* is a precise step-by-step procedure that solves a problem or achieves a goal. Don't write any code for an algorithm in a method until the steps are completely clear to you.

Example 1

A program must test the validity of a four-digit code number that a person will enter to be able to use a photocopy machine. The number is valid if the fourth digit equals the remainder when the sum of the first three digits is divided by seven.

Classes in the program may include an `IDNumber`, the four-digit code; `Display`, which would handle input and output; and `IDMain`, the driver for the program. The data structure used to implement an `IDNumber` could be an instance variable of type `int`, or an instance variable of type `String`, or four instance variables of type `int`—one per digit, and so on.

A top-down design for the program that tests the validity of the number is reflected in the steps of the `main` method of `IDMain`:

Create `Display`
Read in `IDNumber`
Check validity
Print message

Each method in this design is tested before the next method is added to `main`. If the display will be handled in a GUI (graphical user interface), stepwise refinement of the design might look like this:

Create `Display`
    Construct a `Display`
    Create window panels
    Set up text fields
    Add panels and fields to window

Read in `IDNumber`
    Prompt and Read

Check validity of `IDNumber`
    Check input
        Check characters
        Check range
    Separate into digits
    Check validity property

Print message
    Write number
    State if valid

NOTE

1. The `IDNumber` class, which contains the four-digit code, is responsible for the following operations:
    Split value into separate objects
    Check condition for validity

The `Display` class, which contains objects to read and display, must also contain an `IDNumber` object. It is responsible for the following operations:

Set up display
Read in code number
Display validity message

Creating these two classes with their data fields and operations (methods) is an example of encapsulation.

2. The `Display` method `readCodeNumber` needs private helper methods to check the input: `checkCharacters` and `checkRange`. This is an example of procedural abstraction (use of helper methods) and information hiding (making them private).
3. Initially the programmer had just an `IDNumber` class and a driver class. The `Display` class was added as a refinement, when it was realized that handling the input and message display was separate from checking the validity of the `IDNumber`. This is an example of top-down development (adding an auxiliary class to clarify the code).
4. The `IDNumber` class contains no data fields that are objects. It is therefore an independent class. The `Display` class, which contains an `IDNumber` data member, has a composition relationship with `IDNumber` (`Display` *has-a* `IDNumber`).
5. When testing the final program, the programmer should be sure to include each of the following as a user-entered code number: a valid four-digit number, an invalid four-digit number, an n-digit number, where $n \neq 4$, and a "number" that contains a nondigit character. A robust program should be able to deal with all these cases.

Example 2

A program must create a teacher's grade book. The program should maintain a class list of students for any number of classes in the teacher's schedule. A menu should be provided that allows the teacher to

- Create a new class of students.
- Enter a set of scores for any class.
- Correct any data that's been entered.
- Display the record of any student.
- Calculate the final average and grade for all students in a class.
- Print a class list, with or without grades.
- Add a student, delete a student, or transfer a student to another class.
- Save all the data in a file.

IDENTIFYING CLASSES

Use nouns in the specification to identify possible classes.

Use the nouns in the specification as a starting point for identifying classes in the program. The nouns are: program, teacher, grade book, class list, class, student, schedule, menu, set of scores, data, record, average, grade, and file.

Eliminate each of the following:

program	(Always eliminate "program" when used in this context.)
teacher	(Eliminate, because he or she is the user.)
schedule	(This will be reflected in the name of the external file for each class, e.g., `apcs_period3.dat`.)
data, record	(These are synonymous with student name, scores, grades, etc., and will be covered by these features.)
class	(This is synonymous with class list.)

The following seem to be excellent candidates for classes: `GradeBook`, `ClassList`, `Student`, and `FileHandler`. Other possibilities are `Menu`, `ScoreList`, and a `GUI_Display`.

RELATIONSHIPS BETWEEN CLASSES

There are no inheritance relationships. There are many composition relationships between objects, however. The `GradeBook` *has-a* `Menu`, the `ClassList` *has-a* `Student` (several, in fact!), a `Student` *has-a* `name`, `average`, `grade`, `list_of_scores`, etc. The programmer must decide whether to code these attributes as classes or data fields.

IDENTIFYING BEHAVIORS

Use verbs in the specification to identify possible methods.

Use the verbs in the specification to identify required operations in the program. The verbs are: maintain <list>, provide <menu>, allow <user>, create <list>, enter <scores>, correct <data>, display <record>, calculate <average>, calculate <grade>, print <list>, add <student>, delete <student>, transfer <student>, and save <data>.

You must make some design decisions about which class is responsible for which behavior. For example, will a `ClassList` display the record of a single `Student`, or will a `Student` display his or her own record? Who will enter scores—the `GradeBook`, a `ClassList`, or a `Student`? There's no right or wrong answer. You may start it one way and re-evaluate later on.

DECISIONS

Here are some preliminary decisions. The `GradeBook` will `provideMenu`. The menu selection will send execution to the relevant object.

The `ClassList` will maintain an updated list of each class. It will have these public methods: `addStudent`, `deleteStudent`, `transferStudent`, `createNewClass`, `printClassList`, `printScores`, and `updateList`. A good candidate for a helper method in this class is `search` for a given student.

Each `Student` will have complete personal and grade information. Public methods will include `setName`, `getName`, `enterScore`, `correctData`, `findAverage`, `getAverage`, `getGrade`, and `displayRecord`.

Saving and retrieving information is crucial to this program. The `FileHandler` will take care of `openFileForReading`, `openFileForWriting`, `closeFiles`, `loadClass`, and `saveClass`. The `FileHandler` class should be written and tested right at the beginning, using a small dummy class list.

`ScoreList` and `Student` are easy classes to implement. When these are working, the programmer can go on to `ClassList`. This is an example of bottom-up development.

Vocabulary Summary

Know these terms for the AP exam:

Vocabulary	Meaning
software development	Writing a program
object-oriented program	Uses interacting objects
program specification	Description of a task
program design	A written plan, an overview of the solution
program implementation	The code
test data	Input to test the program
program maintenance	Keeping the program working and up to date
top-down development	Implement main classes first, subsidiary classes later
independent class	Doesn't use other classes of the program in its code
bottom-up development	Implement lowest level, independent classes first
driver class	Used to test other classes; contains `main` method
inheritance relationship	*is-a* relationship between classes
composition relationship	*has-a* relationship between classes
inheritance hierarchy	Inheritance relationship shown in a tree-like diagram
UML diagram	Graphical representation of relationship between classes
data structure	Java construct for storing a data field (e.g., array)
encapsulation	Combining data fields and methods in a class
information hiding	Using `private` to restrict access
stepwise refinement	Breaking methods into smaller methods
procedural abstraction	Using helper methods
algorithm	Step-by-step process that solves a problem
stub method	Dummy method called by another method being tested
debugging	Fixing errors
robust program	Screens out bad input
compile-time error	Usually a syntax error; prevents program from compiling
syntax error	Bad language usage (e.g., missing brace)
run-time error	Occurs during execution (e.g., `int` division by 0)
exception	Run-time error thrown by Java method
logic error	Program runs but does the wrong thing

PROGRAM ANALYSIS

Program Correctness

Testing that a program works does not prove that the program is correct. After all, you can hardly expect to test programs for every conceivable set of input data. Computer scientists have developed mathematical techniques to prove correctness in certain cases, but these are beyond the scope of the APCS course. Nevertheless, you are expected to be able to make assertions about the state of a program at various points during its execution.

Assertions

An *assertion* is a precise statement about a program at any given point. The idea is that if an assertion is proved to be true, then the program is working correctly at that point.

An informal step on the way to writing correct algorithms is to be able to make three kinds of assertions about your code.

PRECONDITION

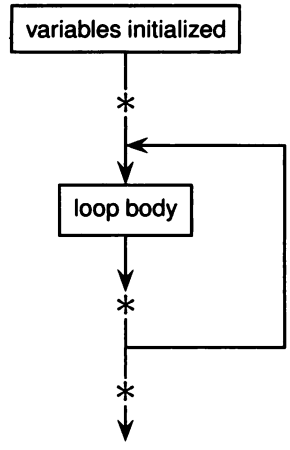
The *precondition* for any piece of code, whether it is a method, loop, or block, is a statement of what is true immediately before execution of that code.

POSTCONDITION

The *postcondition* for a piece of code is a statement of what is true immediately after execution of that code.

LOOP INVARIANT

A *loop invariant* applies only to a loop. It is a precise statement, in terms of the loop variables, of what is true before and after each iteration of the loop. It includes an assertion about the range of the loop variable. Informally, it describes how much of the loop's task has been completed at each stage.



The asterisks show the points at which the loop invariant must be true:

- After initialization
- After each iteration
- After the final exit

Example

```
// method to generate n! //
//Precondition:  n >= 0.
//Postcondition: n! has been returned.
public static int factorial(int n)
{
    int product = 1;
    int i = 0;
    while (i < n)
    {
        i++;
        product *= i;
    }
    return product;
}
```

After initialization	`i = 0,`	`product = 1,` i.e., `0!`
After first pass	`i = 1,`	`product = 1,` i.e., `1!`
After second pass	`i = 2,`	`product = 2,` i.e., `2!`
...		
After kth pass	`i = k,`	`product = k!`

The loop invariant for the `while` loop is

AB *(continued)*

product = i!, 0 ≤ i ≤ n

Here is an alternative method body for this method. (Assume the same method header, comment, and pre- and postconditions.)

```
{
    int product = 1;
    for (int i = 1; i <= n; i++)
        product *= i;
    return product;
}
```

The loop invariant for the `for` loop is

product = (i-1)!, 1 ≤ i ≤ n+1

Here `(i-1)!` (rather than `i!`) is correct because `i` is incremented at the *end* of each iteration of the loop. Also, `n+1` is needed in the second part of the loop invariant because `i` has a value of `n+1` after the final exit from the loop. Remember, the invariant must also be true after the final exit.

Efficiency

An efficient algorithm is one that is economical in the use of

- CPU time. This refers to the number of machine operations required to carry out the algorithm (arithmetic operations, comparisons, data movements, etc.).
- Memory. This refers to the number and complexity of the variables used.

Some factors that affect run-time efficiency include unnecessary tests, excessive movement of data elements, and redundant computations, especially in loops.

Always aim for early detection of output conditions: Your sorting algorithm should halt when the list is sorted; your search should stop if the key element has been found.

In discussing efficiency of an algorithm, we refer to the *best case*, *worst case*, and *average case*. The best case is a configuration of the data that causes the algorithm to run in the least possible amount of time. The worst case is a configuration that leads to the greatest possible run time. Typical configurations (i.e., not specially chosen data) give the average case. It is possible that best, worst, and average cases don't differ much in their run times.

For example, suppose that a list of distinct random numbers must be searched for a given key value. The algorithm used is a sequential search starting at the beginning of the list. In the best case, the key will be found in the first position examined. In the worst case, it will be in the last position or not in the list at all. On average, the key will be somewhere in the middle of the list.

Big-O Notation

AB ONLY

Big-O notation provides a quantitative way of describing the run time or space efficiency of an algorithm. This method is independent of both the programming language and the computer used.

Let n be the number of elements to be processed. For a given algorithm, express the number of comparisons, exchanges, data movements, and primitive operations as a function of n, $T(n)$. (Primitive operations involve simple built-in types and take one unit of time, for example, adding two `int`s, multiplying two `double`s, assigning an `int`,

AB *(continued)*

and performing simple tests.) The type of function that you get for $T(n)$ determines the "order" of the algorithm. For example, if $T(n)$ is a linear function of n, we say the algorithm is $O(n)$ ("order n"). The idea is that for large values of n, the run time will be proportional to n. Here is a list of the most common cases.

Function Type for *T(n)*	Big-O Description
constant	$O(1)$
logarithmic	$O(\log n)$
linear	$O(n)$
quadratic	$O(n^2)$
cubic	$O(n^3)$
exponential	$O(2^n)$

Example 1

An algorithm that searches an unordered list of n elements for the largest value could need n comparisons and n reassignments to a variable `max`. Thus, $T(n) \approx 2n$, which is linear, so the search algorithm is $O(n)$.

Example 2

An algorithm that prints out the last five elements of a long list stored as an array takes the same amount of time irrespective of the length of the list. Thus, $T(n) = 5$, a constant, and the algorithm is $O(1)$.

Example 3

Algorithm 1 executes with $T(n) = 3n^2 - 5n + 10$ and Algorithm 2 has $T(n) = \frac{1}{2}n^2 - 50n + 100$. Both of these are quadratic, and the algorithms are therefore $O(n^2)$. Constants, low-order terms, and coefficients of the highest order term are ignored in assessing big-O run times.

NOTE

Use average-case behavior to determine the big-O run time of an algorithm.

1. Big-O notation is only meaningful for large n. When n is large, there is some value n above which an $O(n^2)$ algorithm will always take longer than an $O(n)$ algorithm, or an $O(n)$ algorithm will take longer than an $O(\log n)$ algorithm, and so on.
2. The following table shows approximately how many computer operations could be expected given n and the big-O description of the algorithm. For example, an $O(n^2)$ algorithm performed on 100 elements would require on the order of $100^2 = 10^4$ computer operations, whereas an $O(\log_2 n)$ algorithm would require approximately seven operations.

n	*O(log₂ n)*	*O(n)*	*O(n²)*	*O(2ⁿ)*
16	4	16	256	2^{16}
100	7	100	10^4	2^{100}
1000	10	1000	10^6	2^{1000}

3. Notice that one can solve only very small problems with an algorithm that has exponential behavior. At the other extreme, a logarithmic algorithm is very efficient.

Chapter Summary

There's a lot of vocabulary that you are expected to know in this chapter. Learn the words!

Never make assumptions about a program specification, and always write a design before starting to write code. Even if you don't do this for your own programs, these are the answers you will be expected to give on the AP exam. You are certain to get questions about program design. Know the procedures and terminology involved in developing an object-oriented program.

Be sure you understand what is meant by best case, worst case, and average case for an algorithm. There will be many questions about efficiency on the AP exam. Level AB students must know the big-O run time for all standard algorithms.

By now you should know what a precondition and postcondition are. Level AB students only, practice some loop invariants.