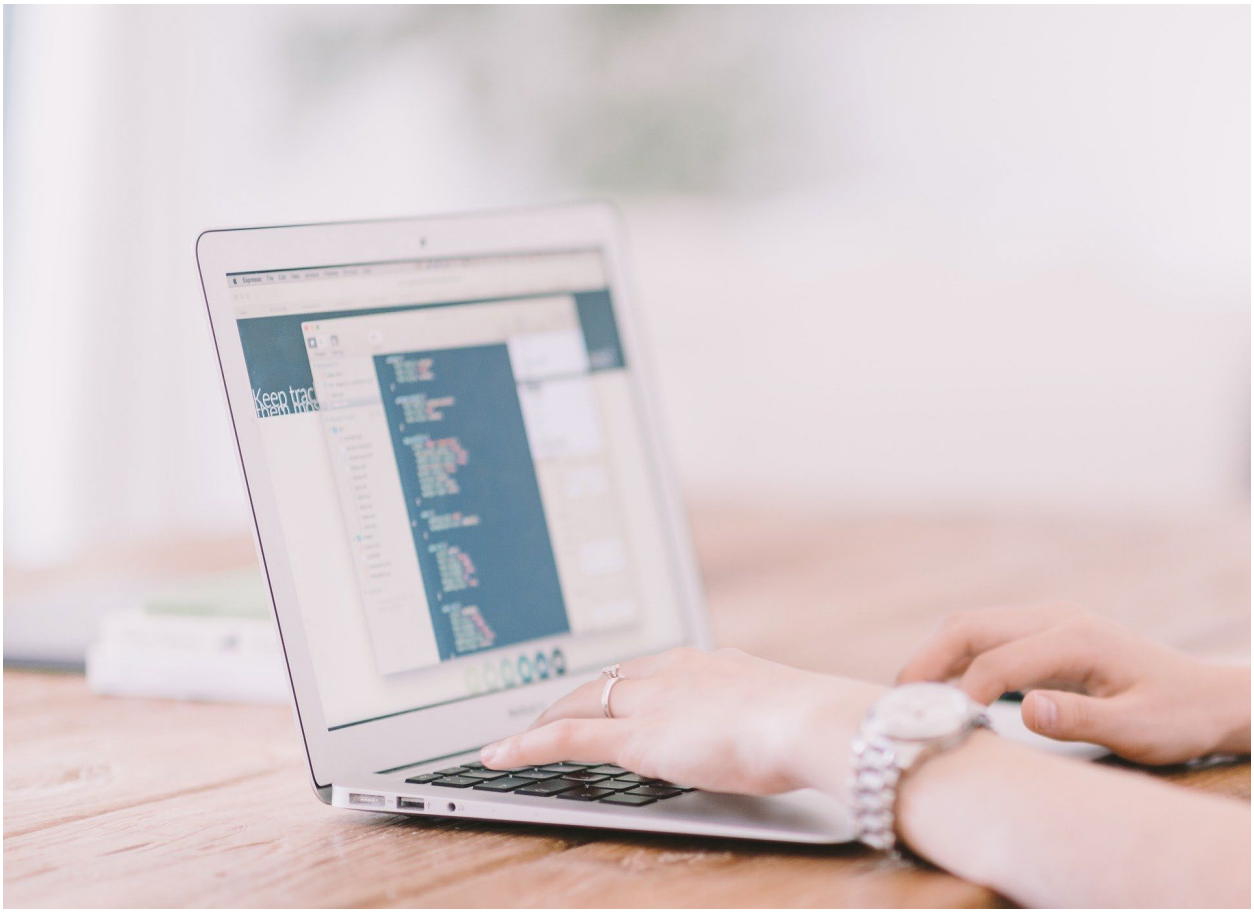
AP Computer Science A Study Guide



Key Exam Details

The AP® Computer Science A course is equivalent to a first-semester, college-level course in computer science. The 3-hour, end-of-course exam is comprised of 44 questions, including 40 multiple-choice questions (50% of the exam) and 4 free-response questions (50% of the exam).

The exam covers the following course content categories:

- Primitive Types: 2.5%–5% of test questions
- Using Objects: 5%–7.5% of test questions
- Boolean Expressions and if Statements: 15%–17.5% of test questions
- Iteration: 17.5%–22.5% of test questions
- Writing Classes: 5%–7.5% of test questions
- Array: 10%–15% of test questions
- `ArrayList`: 2.5%–7.5% of test questions
- 2D Array: 7.5%–10% of test questions
- Inheritance: 5%–10% of test questions
- Recursion: 5%–7.5% of test questions

This guide provides an overview of the main tested subjects, along with sample AP multiple-choice questions that are similar to the questions you will see on test day.

Primitive Types

Around 2.5–5% of the questions you'll see on the exam cover the topic of Primitive Types.

Printing and Comments

The `System.out.print` and `System.out.println` methods are used to send output for display on the console. The only difference between them is that the `println` method moves the cursor to a new line after displaying the given data, while the `print` method does not.

A *comment* is any text in a source code file that is marked to not be executed by the computer. In Java, single line comments are denoted by `//`, and multiline comments are demarcated by `/*` and `*/`, as in the following examples:

```
// this is a single line comment

/*
This is a multiline comment.
All of this will be ignored by the computer.
*/
```

Data Types

Every value in a program is of a certain type, and that type determines what operations can be performed on the value. Every type is categorized as being either a *primitive* type or a *reference* type. Though Java has eight primitive types, only the three shown in the table below are used in AP Computer Science A. All primitive data can be represented using *literals*, which are representations in code of exact values.

Type	Description	Examples of literals
`int`	integer numbers	`3, -14, 21860`
`double`	floating point numbers	`3.14, -1.0, 48.7662`
`boolean`	`true` and `false`	`true, false`

Arithmetic Expressions

The primitive numeric types, `int` and `double`, can be used in *arithmetic expressions*. An arithmetic expression includes numeric values combined with the arithmetic operators:

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	modulus

When more than one operator is present in an expression, they are evaluated according to precedence rules, where operators in the first group are evaluated before operators in the second group:

1) `* / %`
2) `+ -`

For operators within the same group, they are simply evaluated in the order in which they appear in the expression. Parentheses can always be used to override the default precedence rules. For example, the expression `4 + 3 * 2` evaluates to `10`, while `(4 + 3) * 2` evaluates to `14`.

When an arithmetic operation involves two `int` values, the result is an `int`. This is especially important to keep in mind with division, where the result will truncate any non-integer part. For example, `7 / 4` evaluates to `1`, not `1.75` as might be expected. If an operation involves at least one `double` value, the result will be a `double`. In particular, division will behave as expected mathematically.

Variable Declaration and Assignment

Besides using literal expressions, most programs will also use variables to represent data. A *variable* is a name that is associated with a piece of computer memory that stores a value. Once a variable has been declared and assigned a value, it can be used in any situation that the corresponding literal value can be used.

Since every value has a type, every variable has a type as well. Every variable that is used in a program must be declared as being of a certain type. A variable declaration statement consists of a type followed

by a name. For example, the statement `int age;` declares a variable called `age` that will be used to store an `int` value.

Once a variable has been declared, it can be used in an assignment statement. An assignment statement has a variable on the left side of an equal sign, and an expression on the right side. Note that the declaration of a variable can be combined with an assignment statement, so that the following are equivalent:

`int age;` `age = 18;`	`int age = 18;`

The value of a variable can be changed by simply using it in another assignment statement:

```
double fin = 3.2;
System.out.println(fin); // prints 3.2
fin = 4.5 - 5.1;
System.out.println(fin); // prints -0.6
```

If a variable is intended to refer to a value that will never change, it can be declared using the `final` keyword. It can be assigned a value as usual, but then will never be able to be changed again:

```
final int x = 5;
x = x - 2; // this line will cause a compiler error
```

Compound Operators

A common operation in many programs is to retrieve the value of a variable, update it using an arithmetic operation, and storing the result back in the same variable. For example, the statement `x = x * 5` will update the variable *x* so that its new value is five times its original value.

For every arithmetic operator, there is a corresponding compound operator that corresponds to exactly this type of operation.

Compound operator	Example statement	Equivalent to…
`+=`	`x += 3`	`x = x + 3`
`-=`	`x -= 1`	`x = x - 1`
`*=`	`x *= 2`	`x = x * 2`
`/=`	`x /= 10`	`x = x / 10`
`%=`	`x %= 10`	`x = x % 10`

Adding one and subtracting one from a variable are referred to as *increment* and *decrement* operations, respectively, and correspond to additional shortcut operators in Java. Increment and decrement, then, can each be done in three ways:

Increment	`x++`	`x += 1`	`x = x + 1`
Decrement	`x--`	`x -= 1`	`x = x - 1`

Casting

Values of a certain type can only be stored in a variable of that type. The following statements will cause a compiler error, since the right side is `6.3`, a `double` value, while the variable on the left is declared to be of type `int`:

```
int myVariable = 6.3;
```

The casting operators `(int)` and `(double)` can be used to create temporary values converted to another type. Casting a `double` to an `int` results in truncation. For example, if the `double` variable `points` has the value `12.8` the expression `(int)points` will evaluate to `12`, making the following statement legal:

```
int x = (int)points;
```

In some cases, `int` values will automatically be cast to `double` value. This makes it legal to store an `int` value in a `double` variable, as in the following example:

```
int x = 10;
double y = 2 * x + 3; // y will store the value 23.0
```

Similarly, when calling a method that declares a `double` parameter, it is legal to pass an integer value in as the actual parameter.

Free Response Tip

Be careful about storing accumulated values in an `int` variable, especially if the task requires you to find an average. Either store the accumulated value in a `double` variable, or be sure to cast to a `double` before dividing to find the average. Otherwise, the calculated average will be truncated, even if the result is stored in a `double` variable.

Suggested Reading

- Hortsmann. *Big Java: Early Objects*, 6th edition. Chapter 4.
- Gaddis & Muganda. *Starting Out with Java*, 4th edition. Chapter 2.
- Lewis & Loftus. *Java Software Solutions*, 9th edition. Chapter 2.
- Deitel & Deitel. *Java: How to Program, Early Objects*, 11th edition. Chapters 2 and 4.
- Liang. *Introduction to Java Programming, Brief Version*, 11th edition. Chapter 2.

Sample Primitive Types Questions

Consider the following code segment.

```
int x = 9;

int y = 2;

int z = 1;

System.out.println(x / y * 1.5 - z);
```

What is printed when the code segment is executed?

A. 5
B. 5.0
C. 5.75
D. 8
E. 9

Explanation:

The correct answer is choice B. In the arithmetic expression, division and multiplication have the highest and identical precedence and will be evaluated left-to-right. 9/2 evaluates to 4 (integer division). 4 multiplied by 1.5 evaluates to 6.0 (a double data type). 6.0 – 1 evaluates to 5.0.

What is the output of the following code segment?

```
double a = 3.6;

int b = (int)a + 2;

double c = b;

System.out.print(a + " " + b + " " + c);
```

A. 3.6 5 5.0
B. 3.6 6 6.0
C. 3.6 6 6
D. 3.6 5 5
E. 3.6 5.0 6

Explanation:
The correct answer is A. The first line of the code segment assigns the value 3.6 to the variable a. The variable a, typecast as an `int`, evaluates to 3—the expression 3 added to 2 evaluates to 5, which is assigned into b. 5 is assigned into variable c, but is automatically widened into the double data type, 5.0.

Consider the following code segment.

```
int a = 8;

System.out.print("*****");
```

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

System.out.println(a + 2)

System.out.println("*****");
```

What is printed when the code segment is executed?

A.

```
*****8
10
*****
```

B.

```
*****810*****
```

C.

```
*****
8
10
*****
```

D.

```
*****
8
10*****
```

E.

```
*****
810
*****
```

Explanation:
The correct answer is A. The difference between `System.out.print` and `System.out.println`, is that `System.out.println` advances the cursor to the next line (printing a newline) immediately after printing the specified argument. In this correct answer, first a row of five asterisks is printed without a newline, then the value of the variable a is printed with a newline, then the value of the expression a+2 is printed with a newline, and finally a row of five asterisks is printed with a newline.

Using Objects

On your AP exam, 5–7.5% of questions will cover the topic Using Objects.

As mentioned, all values in Java belong to either a primitive type or a reference type. The numeric primitive types were covered in the previous section. In this section, we will discuss reference types.

An *object* is a compound value that has attributes, or data, and methods that can access or manipulate the attributes. A *class* is a blueprint, or template, for the objects of a certain type. A class specifies what attributes and methods an object will have.

Constructing and Storing Objects

An object is created from a class by calling the class *constructor* along with the `new` keyword. The name of a constructor is the same as the name of the class it belongs to, and it is followed by a (possibly empty) list of values. These values are parameters, and they represent initial values that will be used to create the object.

The *signature* of a constructor consists of the name of the constructor along with the list of types of parameters that it expects. When calling the constructor, the parameter list provided must match the signature. A class may define multiple constructors as long as their signatures differ; in such a case, the constructor is said to be *overloaded*.

For example, the `Rectangle` class from the Java standard library contains, among others, the following two constructors:

```
Rectangle(int width, int height)
Rectangle(int x, int y, int width, int height)
```

The following would be valid constructor calls:

```
new Rectangle(5, 6) // calls the first constructor
new Rectangle(-1, 2, 3, 8) // calls the second constructor
```

However, the following would not be valid:

```
new Rectangle(3.2, 1) // invalid since the first parameter is double
new Rectangle(1, 2, 3) // invalid since it has three parameters
```

An object needs to be stored in a variable whose type is compatible with the class the object belongs to. In most cases, the type of the variable will exactly match the type of the object. An exception is discussed in the section on inheritance.

A complete statement to construct and store a rectangle object, then, looks like this:

```
Rectangle myRectangle = new Rectangle(5, 6);
```

A variable that refers to an object, as opposed to a primitive value, is called a *reference variable*. The name comes from the fact that the memory associated with it does not store the object itself, but rather a

reference to the object; that is, the location in memory where the object exists. The special value of `null` is reserved for reference variables that do not contain a reference to any actual object.

Calling Void Methods

Interaction with objects is done primarily by calling their methods, which define what the object can do, and what can be done with it. As with constructors, every method has a signature. The signature of a method consists of its name along with a (possibly empty) list of the types of parameters it defines. Methods can be overloaded. That is, multiple methods with the same name may exist in a class, as long as their signatures are different.

When a method is called, the execution of the program is interrupted, and control is transferred to the method. When the method is complete, execution continues at the method call. Some methods return a result when they are complete, in which case that value is available when execution continues.

Other methods, known as *void* methods, do not return a value, and therefore can only be called as standalone statements, rather than as part of an expression. A method is called by using the dot operator between the name of the object and the name of the method, followed by a list of parameters in parentheses.

For example, the `Rectangle` class defines a void method with signature `grow(int h, int v)`. If the variable `myRectangle` stores a reference to a `Rectangle` object, the following is a valid call to the `grow` method:

```
myRectangle.grow(4, 4);
```

Note that a method cannot be called on a `null` value, so in the previous example, if `myRectangle` was `null`, the statement shown would cause a `NullPointerException` to be thrown.

Calling Non-Void Methods

When a method is not void, it has a *return type*. The method returns a value, and the method call expression evaluates to this value. Since it has a value, it can be used as part of an expression in place of any value of the specified type.

For example, the method `getHeight()` in the `Rectangle` class returns a value of type `int`. Therefore, a call to the `getHeight` method can be used in place of an integer value in any expression. If the variable `myRectangle` refers to a `Rectangle` object, then the following is a valid statement:

```
int someValue = 2 * myRectangle.getHeight() + 1;
```

If, in addition to returning a value, a method has side effects, there may be instances when you do not care about the returned value. If you only care about the side effects of a method, it can be called as if it were a void method, even if it returns a value. This means that the following statement is legal, although it may not be useful in many situations:

```
myRectangle.getHeight();
```

Strings

A *string* is a sequence of characters. String literals are enclosed in double quotes, as in `"Hello"`, `"32"`, and `""`. Notice that in the second example, the value is a string, even though it contains numeral characters. The last of these examples is referred to as the empty string.

Strings represent an exception to the general rule of object construction. Since they are so common, the constructor does not have to be explicitly called with the `new` keyword as with all other objects. Rather, they can be constructed by simply using literal values. Strings can be combined, or concatenated, using the `+` operator. The following example shows the creation and concatenation of strings.

```
String str1 = "AP";
String str2 = "Exam";
String combined = str1 + " " + str2; // combined will store the string
"AP Exam"
```

When primitive values are concatenated with strings, they are automatically cast to strings. Therefore, the expression `"I am " + (14 + 4) + " years old"` evaluates to the string `"I am 18 years old"`.

An *escape sequence* is a series of characters beginning with a backslash that has special meaning in Java. The following table shows the three escape sequences used in AP Computer Science A.

Escape sequence	Meaning
`\"`	double quote character
`\\`	backslash character
`\n`	new line character

The characters in a string are classified by their position, or *index*. The indices start at 0, so that in the string `"Hello"`, the `"e"` character is at index 1. Any attempt to refer to a character at an invalid index will result in a `StringIndexOutOfBoundsException` being thrown.

The following table shows the methods that are included in the AP Computer Science A exam.

Method/Constructor	Description
`String(String str)`	constructs a new string that is identical to `str`
`int length()`	returns the number of characters in the string
`String substring(int from, int to)`	returns a new string consisting of the characters that range from index `from` to index `to - 1`
`String substring(int from)`	returns a new string consisting of the characters beginning at index `from` and continuing to the end of the string
`int indexOf(String str)`	returns the index of the first occurrence of `str` within the string, if any, and `-1` otherwise
`boolean equals(String other)`	returns `true` if the string is equal to `other`, and `false` otherwise
`int compareTo(String other)`	returns a negative value if the string comes before `other`, a positive value if the string comes after `other`, and 0 if the two are equal

The `charAt` method is not covered; retrieving a single character (as a string) at index `n` can be accomplished by calling `substring(n, n+1)`.

It is important to note that strings are immutable. They have no mutator methods. Methods such as `substring` return a new string, and do not modify the original.

Wrapper Classes

There are various circumstances in which it is more convenient to work with objects rather than primitive values (see the section on `ArrayList` for an example). Because of this, Java provides the wrapper classes `Integer` and `Double`. Each of these classes has a constructor that accepts a primitive value of the appropriate type, and a method that returns the primitive value. The `Integer` class also provides static fields that represent the maximum and minimum values that can be represented by an `int`.

Integer
`Integer(int value)` Constructs an `Integer` object with the specified value
`int intValue()` Returns the stored `int` value
`Integer.MAX_VALUE` The maximum value that can be represented by an `int`
`Integer.MIN_VALUE` The minimum value that can be represented by an `int`

Double
`Double(double value)` Constructs a `Double` object with the specified value
`double doubleValue()` Returns the stored `double` value

The Java compiler has features called autoboxing and unboxing that automatically convert between these wrapper classes and the corresponding primitive types. This makes it unnecessary to explicitly construct `Integer` or `Double` objects, and to call their `intValue` or `doubleValue` methods, respectively. In practice, then, when a method expects a `double`, a `Double` can be passed, and vice versa.

Static Methods and Math

A *static* method is a method that is called using the name of the class to which it belongs, rather than an object. The `Math` class is an example of a class that contains only static methods. The methods you are expected to know are in the following table.

Method	Description
`int abs(int x)`	Returns the absolute value of `x`
`double abs(double x)`	Returns the absolute value of `x`
`double pow(double b, double e)`	Returns b^e
`double sqrt(double x)`	Returns the square root of `x`
`double random()`	Returns a value in the interval $[0,1)$

Free Response Tip

When writing code in a free response question, think carefully about whether every method you call is static or not. If it is static, make sure it is preceded by a class name. If not, make sure it is preceded by an object name. The only time you can call a method without any dot operator is when you are calling a method within the same class.

Suggested Reading

- Horstmann. *Big Java: Early Objects*, 6th edition. Chapter 2.
- Gaddis & Muganda. *Starting Out with Java*, 4th edition. Chapters 5 and 6.
- Lewis & Loftus. *Java Software Solutions*, 9th edition. Chapter 3.
- Deitel & Deitel. *Java: How to Program, Early Objects*, 11th edition. Chapter 3.
- Liang. *Introduction to Java Programming, Brief Version*, 11th edition. Chapters 2 and 4.

Sample Using Objects Questions

Consider the following code segment.

```
public class Toy
{
    private String name;
    public Toy()
    {
        name = "Venus";
    }
    public Toy(String s)
        name = s;
    }
    public void printName()
    {
        System.out.println("This toy is named: " + name);
    }
}
```

Which of the following statements correctly creates an instance of a Toy object named `Mars`? Assume that each of the choices exist in a class other than Toy.

A.
```
Toy t = new Toy();
t.name = "Mars";
```

B.
```
Toy t = new Toy("Mars");
```

C.
```
Toy t;
t = "Mars";
```

D.
```
Toy t = "Mars";
```

E.
```
String s = "Mars";
Toy t = s;
```

Explanation:
The correct answer is B. Every object is created using the keyword `new` followed by a call to one of the class's constructors. Choice A is incorrect because `name` is a private field and cannot be referenced outside of the class `Toy`. Choices C, D, and E are incorrect because a constructor is never called using the keyword new to instantiate the class `Toy`.

Consider the following code segment.

```
String s1 = "Hello";

String s2 = "World";

System.out.println(s1 + ", " + s2);

String s3 = new String("Hi");

s3 += ",\nWorld";

System.out.println(s3);
```

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

A.
```
Hello+, +World
Hi,\nWorld
```

B.
```
Hello, World
Hi,
World
```

C.
```
Hello
World
,
Hi
,
World
```

D.
```
HelloWorld,
Hi,
World
```

E.
```
Hello, World
Hi,\nWorld
```

Explanation:
The correct answer is B. The + operator is used to concatenate Strings in Java. String objects can either be created by a literal String, or by the `new` operator. `\n` represents the newline character. The += operator concatenates onto the end of the specified String.

Which of the following is a code segment that would yield a compiler error?

A. `double v = Math.pow((double)4, 3.0);`
B. `double w = Math.pow(4.0,0.0);`
C. `int x = Math.pow(4.0,3.0);`
D. `double y = Math.pow(4,3);`
E. `double z = Math.pow(4.0,3.0);`

Explanation:
The correct answer is C. The `Math.pow` method returns a `double`, which cannot be automatically narrowed into the `int` datatype. An `int` typecast is necessary. The code segment in choice A will compile successfully; The `double` typecast is not necessary, but in this case, explicitly converts 4 to 4.0. The code segment in choice B will compile successfully; a base raised to the exponent 0.0 will evaluate to 1.0. The code segment in choice D will compile successfully; the `Math.pow` method takes two `double` parameters. The `int` arguments are implicitly converted into `double` values. The code segment in choice E will compile successfully; the `Math.pow` method takes two `double` parameters and returns a `double` value.

Boolean Expressions and if Statements

About 15–17.5% of the questions on your AP exam will cover the topic of Boolean Expressions and if Statements.

Boolean Expressions

A *Boolean* value is either `true` or `false`. Although these literals can be used, the more common way to create Boolean values is using the relational operators with primitive values. The six relational operators are <, <=, >, >=, ==, and `!=`. Any expression that evaluates to a Boolean value is called a *Boolean expression*.

More complex Boolean expressions can be formed using the logical operators `&&` (and), `||` (or), and `!` (not).

`&&`	`A && B` is true only when `A` and `B` are both true
`||`	`A || B` is true when at least one of `A` or `B` is true
!	`!A` is true only when `A` is false

Two Boolean expressions are equivalent if they evaluate to the same truth value for all possible values of their variables. De Morgan's Laws provide a common method for transforming Boolean expressions into equivalent ones. The laws state that `!(A && B)` is equivalent to `!A || !B`, and that `!(A || B)` is equivalent to `!A && !B`.

Comparing Objects

Recall that reference variables store references to objects, rather than the objects themselves. Because of this, when objects are compared using == and `!=`, it is only the references that are being compared, not the contents of the actual objects. That is, these operators only check if two references are aliases of each other. Additionally, == and `!=` can be used to check whether a reference variable is `null`.

Comparing objects themselves for equality can only be achieved if the class provides an `equals` method, as we saw exists for `String`.

Free Response Tip

Only use == and `!=` if you are either comparing primitive values or checking to see if a reference variable is `null`. If an object has an equals method, and you want to check if it is not equal to another object, you can use the following idiom: `!obj1.equals(obj2)`.

`if` and `if-else` Statements

Control flow statements are used when a program needs to make decisions based on its current state. An `if` statement allows the program to either execute or skip a section of code based on whether a Boolean expression is true or false.

The syntax for an `if` statement is shown as follows:

```
if (expression) {
      // one or more statements
}
```

If `expression` evaluates to `true`, the statements in the block (between the `{}`) are executed. Otherwise, the statements in the body are skipped.

Optionally, an `else` clause can be added:

```
if (expression) {
      // one or more statements
} else {
      // one or more statements
}
```

If `expression` evaluates to `true`, the statements in the `if` block are executed. Otherwise, the statements in the `else` block are executed.

For both the `if` and `else` blocks, if they consist of only a single statement, the curly braces can optionally be omitted.

`if-else-if` Statements

To check for multiple possibilities, an `if` can be followed by one or more `else if` clauses.

```
if (expression1) {
      // statements1
} else if (expression2) {
      // statements2
} else if (expression3) {
      // statements3
} else {
      // statements4
}
```

In this code, there are four possible execution paths. Execution begins at the top. If `expression1` is true, `statements1` is executed. Otherwise, if `expression2` is true, `statements2` is executed. If neither of the first two expressions are true, but `expression3` is, then `statements3` is executed. Finally, if none of the expressions are true, the `else` block is executed.

Note that there can be an arbitrary number of `else if` clauses, and that the `else` block at the end is optional. If no `else` block is provided, and none of the expressions are true, then nothing will be executed.

Free Response Tip

When writing `if-else-if` statements, keep in mind that each expression is only evaluated if none of the previous ones have evaluated as `true` yet; at most one of the statement blocks will be executed. This contrasts with consecutive `if` statements, where all of the Boolean expressions will be checked, and many of the statement blocks can potentially be run.

Suggested Reading

- Horstmann. *Big Java: Early Objects,* 6th edition. Chapter 4.
- Gaddis & Muganda. *Starting Out with Java,* 4th edition. Chapter 3.
- Lewis & Loftus. *Java Software Solutions*, 9th edition. Chapter 5.
- Deitel & Deitel. *Java: How to Program, Early Objects,* 11th edition. Chapter 5.
- Liang. *Introduction to Java Programming, Brief Version*, 11th edition. Chapter 3.

Sample Boolean Expressions and if Statements Questions

Consider the following Boolean expression.

```
(temperature >= 20 && temperature <= 80) && (humidity <= .5)
```

Which of the following Boolean expressions are equivalent to the expression above?

I. `(humidity <= .5) && (20 <= temperature <= 80)`

II. `(! (humidity > .5)) && (!(temperature < 20 || temperature > 80))`

III. `(20 <= temperature && temperature <= 80) && (humidity <= .5)`

A. II only
B. III only
C. II and III only
D. I and III only
E. I, II, and III

Explanation:
The correct answer is C. The Boolean expression to consider in this question returns `True` if `temperature` is between `20` and `80` inclusive, and humidity is less than or equal to `.5`. Following De Morgan's Laws, Boolean expression II is equivalent by checking if humidity is not greater than .5 and also not outside of the range `20-80`. `20 <= temperature` is equivalent to `temperature >= 20` in Boolean expression III.

Consider the following code segment:

```
int score = 5;
if (score >= 2)
{
   score = 1;
   System.out.print("A");
}
else
{
   System.out.print("B");
}
if (score <= 3)
{
```

```
   score = 9;
   System.out.print("C");
}
else
{
   System.out.print("D");
}
System.out.print(score);
```

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

A. A1
B. BD5
C. BD9
D. AC9
E. ABCD9

Explanation:
The correct answer is D. There are two if-else blocks. In the first if-else block, the Boolean expression evaluates to true (5>=2), and the if block is executed (prints A and assigns the value of 1 to the variable score) and the else is skipped. In the second if-else block, the Boolean expression evaluates to true (1<=3), and the if block is executed (prints C and assigns the value of 9 to the variable score) and the else is skipped. The final value of score is 9. B and D are not printed because the Boolean expression in both if-else blocks evaluates to true.

The following table maps a temperature in Fahrenheit to a qualitative description.

Temperature in Fahrenheit	Description
90 or above	Hot
70 – 89	Warm
45 – 69	Mild
44 or below	Chilly

Which of the following code segments will print the correct description for a given integer temperature?

I.

```
if (temperature >= 90)
{
    System.out.println("Hot");
}
else if (temperature >= 70)
```

```
{
    System.out.println("Warm");
}
else if (temperature >= 45)
{
    System.out.println("Mild");
}
else
{
    System.out.println("Chilly");
}
```

II.

```
if (temperature <= 44)
{
    System.out.println("Chilly");
}
else if (temperature <= 69)
{
    System.out.println("Mild");
}
else if (temperature <= 89)
{
    System.out.println("Warm");
}
else
{
```

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

III.

```
if (temperature <= 44)
{
    System.out.println("Chilly");
}
else if (45 <= temperature && temperature < 70)
{
    System.out.println("Mild");
}
else if (70 <= temperature && temperature < 90)
{
    System.out.println("Warm");
}
else if (90 <= temperature)
{
    System.out.println("Hot");
}
```

A. I only
B. I and II only
C. II only
D. II and III only
E. I, II, and III

Explanation:
The correct answer is E. In a multiway condition statement, the first section of code is executed based on whichever condition is first true. For any possible integer value, the same string will print in all three code segments.

Iteration

On your AP exam, 17.5–22.5% of questions will cover the topic of Iteration.

`while` Loops

The `while` statement allows a code block to repeat as long as a condition is true.

```
while (expression) {
  // one or more statements
}
```

In the previous code, the block will be executed if `expression` is true. After execution, the condition is checked again, and if true, the block is executed again. This continues until the condition becomes false, at which point the rest of the program continues. Note that if `expression` is false the first time it is encountered, the body of the loop is never run.

It is important to make sure that the condition eventually becomes false, or the loop will be executed infinitely. For example, consider the following code:

```
int x = 12;
while (x > 0) {
  System.out.println("A");
}
```

This is an infinite loop, since the condition will always be true: `x` starts with a value of `12`, and there is nothing within the body of the loop that changes the value. Therefore, it will always be greater than 0. To fix the problem, consider this change:

```
int x = 12;
while (x > 0) {
  System.out.println("A");
  x--;
}
```

With the addition of the `x--` statement, the value of `x` is reduced by 1 each time through the loop. Eventually, it will become 0, the condition will no longer be true, and the loop will stop.

`for` Loops

A `for` loop uses a variable to count the iterations of a loop. It has the following structure:

```
for (initialization; expression; increment) {
  // statements
}
```

When the loop is first encountered, the *initialization* statement is executed. Then *expression* is evaluated. If it is true, the loop body is executed. After execution, *increment* is executed, and *expression* is evaluated again. This continues until *expression* is false, at which point the loop is done.

In practice, the initialization statement usually consists of a variable declaration and assignment, and the increment modifies that variable by adding or subtracting a fixed value, as in the following examples:

```
for (int i = 0; i < 5; i++) { … }
```

```
for (int count = 100; count >= 0; count -= 2) { … }
```

There are several standard algorithms that use `for` loops with which you should be familiar. These include:

- Identify the individual digits within an integer using a while loop, modulus, and division.
- Count the number of times that a criterion is met within a range of values.

There are many other such algorithms that involve arrays, which will be covered later.

Free Response Tip

When writing code, there are no absolute rules for deciding when to use a `while` loop vs a `for` loop. Either kind of loop can be used in any situation. However, `while` loops are generally better for situations in which you don't know in advance how many times the code needs to be executed, and need to check a condition to know whether or not to continue, whereas `for` loops are easier to use when you can explicitly count the repetitions.

String Algorithms

Loops are often used in the context of processing strings. Combined with the `indexOf` and `substring` methods, loops are essential for examination of different parts of strings. There are two particular patterns that frequently occur.

```
for (int i = 0; i < myStr.length() - k + 1; i++) {
  String subs = myStr.substring(i, i + k);
  // do something with subs
}
```

In this code, the variable `i` is used to keep track of the index in a string. In each iteration, a substring of length `k` is retrieved starting at index `i`. The substring can then be used as desired. For example, it could be concatenated onto another string, or checked for equality with some target string. Note carefully the value used in the loop condition: `i < str.length() - k + 1`. This is important for ensuring that the `substring` method does not result in a `StringIndexOutOfBoundsException`.

The other common pattern is using a `while` loop together with the `indexOf` method:

```
int pos = myStr.indexOf(target);
while (pos > 0) {
  // target string was found
  pos = myStr.indexOf(target, pos + 1);
}
```

Here, a string is repeatedly searched for some target. When it is no longer found, the `indexOf` method will return `-1`, which will cause the loop to terminate.

Nested Loops

When a loop is used in the body of another loop, it is referred to as a *nested* loop. Nested loops are often used in more complex string and array algorithms and are very common with 2D arrays. They are also commonly used for printing tabular data and patterns, as in this example:

```
for (int x = 5; x >= 1; x--) {
  for (int y = 0; y < x; y++) {
    System.out.print(y + " ");
  }
  System.out.println();
}
```

The result of executing this code will be the following:

```
01234
0123
012
01
0
```

Suggested Reading

- Horstmann. *Big Java: Early Objects*, 6th edition. Chapter 6.
- Gaddis & Muganda. *Starting Out with Java*, 4th edition. Chapter 4.
- Lewis & Loftus. *Java Software Solutions*, 9th edition. Chapter 5.
- Deitel & Deitel. *Java: How to Program, Early Objects*, 11th edition. Chapter 5.
- Liang. *Introduction to Java Programming, Brief Version*, 11th edition. Chapter 5.

Sample Iteration Questions

Which of the following code segments correctly completes the line marked **`/* missing code */`** in the following method named `reverse`? The method takes a `String` argument and returns a new string with the characters in the argument string reversed.

```
public static String reverse(String s)
{
    String toReturn = "";
    for ( /* missing code */ )
    {
        toReturn += s.charAt(i);
    }
    return toReturn;
}
```

A. `int i = s.length(); i > 0; i--`
B. `int i = 0; i < s.length(); i++`
C. `char i : s`
D. `int i = s.length() - 1; i >= 0; i--`
E. `int i = 0; i < s.length() - 1; i++`

Explanation:
The correct answer is D. Because the desired behavior of this method is to return a new string that is a reversal of the characters in the argument, one algorithm is that the for loop should iterate from the last character in the string to the first. The index of the last character is one less than the length of the string. The update of the for loop decrements to count backwards. Choice A is incorrect because this loop incorrectly throws an ArrayOutOfBounds exception during the first iteration of the loop. Because array indices begin at 0, the index of the last character is one less than the length of the string. Choice B is incorrect because this loop incorrectly iterates from the first character in the string to the last character. Choice C is incorrect because this enhanced for loop incorrectly extracts every character from a string instead of its position value. Choice E is incorrect because this loop incorrectly iterates from the first character in the string to the character before the last.

Consider the following code segment.

```
int i, j;
for (i = 1; i <= 3; i++)
{
    System.out.print(i + " ");
    j = i;
```

```
    while (j > 0)
    {
        System.out.print(j + " ");
        j--;
    }
}
```

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

A. 1 1 1 2 2 1 1 2 3 3 2 1
B. 1 1 0 2 2 1 0 3 3 2 1 0
C. 1 1 2 2 1 3 3 2 1
D. 1 1 2 2 3 3
E. 1 1 0 2 1 3 2 1

Explanation:
The correct answer is C. When a loop is placed inside of another loop, the inner loop must complete all of its iterations before the outer loop can continue. In this code segment, the outer loop iterates and prints 1 2 3. After 1 is printed, the inner loop prints 1; after 2 is printed, the inner loop prints 2 1; after 3 is printed, the inner loop prints 3 2 1.

Consider the following code segment, which includes line numbers.

```
1 int i;
2 for (i = 1; i <= 100; i += 2)
3 {
4   if (i >= 10 && i < 100)
5   {
6     System.out.print(i + " ");
7   }
8 }
```

Explain how the result of the following program code changes, given that Line 2 is changed to:

```
for (i = 100; i >= 1; i -= 2)
```

A. The initial program prints all two-digit odd numbers in increasing order; the modified program prints all even numbers from 100 to 2 in decreasing order.
B. The initial program prints all two-digit even numbers in increasing order; the modified program prints all two-digit odd numbers in decreasing order.

C. The initial program prints all two-digit even numbers in increasing order; the modified program prints all two-digit even numbers in decreasing order.
D. The initial program prints all two-digit odd numbers in increasing order; the modified program prints all two-digit even numbers in decreasing order.
E. The initial program prints all odd numbers from 1 to 99 in increasing order; the modified program prints all even numbers from 100 to 2 in decreasing order.

Explanation:

The correct answer is D. Line 4 of the code segment enforces the condition that only two-digit integers are printed in Line 6. The initial program's for loop iterates over every odd integer from 1 to 100; the modified program's for loop iterates over every even integer from 100 to 1.

Writing Classes

About 5–7.5% of the questions on your AP exam will cover the topic of Writing Classes.

Class Structure and Visibility

The basic structure of a class is shown below.

```
public class MyClass {

  // instance variables

  // constructor

  // methods
}
```

The instance variables are the data, or attributes, associated with an object. C*onstructors* are the means by which the objects are constructed, and *methods* are the behaviors that the objects have available to them.

Every declaration in a class, whether it is an instance variable, constructor, or method, has a *visibility modifier* attached to it. The visibility modifier can be `public` or `private`. When something is declared as `public`, it can be directly accessed from outside of the class. Declaring something as `private`, on the other hand, restricts access to code inside the class itself. For the purposes of the AP Computer Science A exam, the following guidelines apply:

- All instance variables should be `private`
- All constructors should be `public`
- Methods may be `public` or `private`

Encapsulation refers to the idea that an object should keep the details of its internal workings hidden. Declaring instance variables as `private`, and only allowing access to them via carefully designed `public` accessor and mutator methods, ensures that this principle is followed.

Other than the addition of the visibility modifier, the declaration of an instance variable follows the same syntax as that of a local variable:

```
private String myName;

private boolean isOpen;
```

Free Response Tip

Make sure you *never* declare a `public` instance variable. Although doing so will not result in any sort of error, it is considered very poor programming practice, and will often be penalized in free response questions even if your code works flawlessly.

Constructors

The purpose of a constructor is to set up an object with some initial state, and generally consists of assigning values to the instance variables. Constructors can have one or more parameters. As mentioned previously, constructors can be overloaded. That is, a class can have multiple constructors, as long as they all have different signatures. A constructor that does not have any parameters is referred to as a default constructor.

If a class has no constructor, or if a constructor does not explicitly set an instance variable, the variable will automatically be given a default value. The default value for numeric types is `0`, for Boolean values it is `false`, and for reference types it is `null`.

Documentation

In addition to single-line and multi-line comments, Java has a third comment syntax, which generates Javadoc documentation. These comments are enclosed by `/**` and `*/`, and are used to document the description, purpose, and conditions associated with a class, instance variable, constructor, or method.

A *precondition* is a condition that must be true immediately prior to a method being called. If it is not true, the method should not be expected to work as described and may cause an error to occur. In other words, it is the responsibility of the programmer calling the method to ensure that the precondition is satisfied before calling the method; the condition will not actually be checked within the method.

A *postcondition*, on the other hand, describes something that is guaranteed to be true immediately after the execution of the method. It often describes the behavior of the method by answering two questions:

- What does this method return?
- What will the state of the object be when this method is complete?

Free Response Tip

Pay very close attention to the preconditions and postconditions given in the narratives for free response questions. They are usually very detailed and exactly describe the method that writing is expected to do. Preconditions will save you from having to write code to check conditions, and postconditions will help you ensure that your code does what it is meant to do.

Writing Methods

A *method* is a block of code that exists within a class and has access to the instance variables. The syntax for writing a method is as shown:

```
visibility returnType methodName(parameters) {
  // method body
}
```

`visibility` can be either `public` or `private`. Most methods are `public`, but there may be situations in which it makes sense to keep the accessibility of a method limited to its class. The return type specifies what type of value, if any, the method will return. If the method will not return any value, the keyword `void` is used in place of the return type.

A method is called an *accessor* if it retrieves and returns, but does not modify, data associated with the class. This may be the value of an instance variable, or a computed value derived from several instance variables and parameters. To return a value from a method, the statement `return expression;` is used. It is important to note that a `return` statement will immediately terminate the execution of the method. Any code that follows it will never be reached, and if it is within a loop there will be no further iterations.

Free Response Tip

Consider whether the method you are writing is `void` or not. If it is `void`, make sure you do not include a return statement anywhere in your code. On the other hand, if it is not void, you must make sure that *every possible* execution path includes a return statement.

A *mutator* method changes the state of the object in question by modifying one or more of its instance variables. Recall that the principle of encapsulation requires that an object keeps its instance variables private. Mutator methods are how an object can provide a publicly accessible means of allowing limited modification, while maintaining control of the implementation details.

A common method implemented in many classes is the `toString` method. This method returns a string and does not have any parameters. It is intended to return a string description of the object in question, usually including some or all of the values of its instance variables. Of note, whenever an object reference is passed to the `System.out.print` or `System.out.println` method, the `toString` method of the object is called automatically, and the returned string is printed to the console.

If an object is passed as a parameter to a method, the method can only access its private variables if it is of the same type as the enclosing class. Otherwise, it is limited to using publicly accessible methods from the parameter object.

The parameter as declared within a method header is referred to as a *formal parameter*, and the value passed in when the method is called is called an *actual parameter*. Consider the following example of a method and a call to it:

```
public void doSomething(int x) {
  System.out.println(x);
}

doSomething(3);
```

The formal parameter here is `x`, while the actual parameter is `3`.

In Java, parameters are always passed *by value*. This means that if a variable is specified as an actual parameter, the method receives a copy of the variable, not the variable itself. If the parameter is a primitive value, this means that the actual parameter can never be modified from within the method. If the parameter is a reference type, however, the formal parameter ends up being an alias of the actual parameter. In this case, the underlying object may be able to be changed within the method. However, as a general rule, it is considered good practice to avoid mutating an object passed to a method as a parameter, unless the postconditions of the method require that it be done.

Scope and Access

The *scope* of a variable refers to the code within which it is accessible. There are several important principles related to scope that you need to know:

- A *local* variable is one that is declared within the body of a constructor or method. These variables are not declared as either public or private, and they are only accessible within their enclosing blocks.
- Instance variables declared in a class are accessible throughout the entire class.
- If an instance variable and a local variable have the same name, the local variable is said to *shadow* the instance variable. Within the scope of the local variable, the variable name will refer to it, and not to the instance variable.
- Parameters behave similarly to local variables, in that they are only accessible in the constructor or method within which they are declared.

Within a constructor or non-static method, there is always an additional variable available, called `this`, which behaves similarly to an instance variable. It is a reference to the current object, and it can be used to pass the object as a parameter to a method.

Static Variables and Methods

Static variables and methods are associated with a class, rather than with instances of the class, and are declared by including the static keyword. For example:

```
private static int count;
public static double getValue() { … }
```

Static methods can be either public or private, but they only have access to static variables that cannot read or change any instance variables. In particular, they do not have access to the `this` keyword, as discussed in the previous section.

Suggested Reading

- Horstmann. *Big Java: Early Objects,* 6th edition. Chapter 3.
- Gaddis & Muganda. *Starting Out with Java*, 4th edition. Chapter 8.
- Lewis & Loftus. *Java Software Solutions*, 9th edition. Chapter 4.
- Deitel & Deitel. *Java: How to Program, Early Objects*, 11th edition. Chapter 8.
- Liang. *Introduction to Java Programming, Brief Version,* 11th edition. Chapters 9 and 10.

Sample Writing Classes Questions

Consider the following instance variables and method.

```
private int[] numbers;
private int value;


/**
 * Precondition: numbers contains int values in no particular order
 */
public void mystery()
{
    for (int i = 0; i <= numbers.length - 1; i++)
    {
        if (numbers[i] < value)
        {
            numbers[i] = 0;
        }
    }
}
```

Which choice best describes the postcondition of the method `mystery`?

A. All elements in the array will be set to 0.
B. Elements in the array will have the same value as they had before the method.
C. Elements in the array are set to zero if they exceed the variable `value`.
D. All elements in the array will be set to 0, except for the last element.
E. Elements in the array are set to zero if the variable `value` exceeded them.

Explanation:
The correct answer is E. The if statement checks if the value of an array element is less than the variable `value`. If it is, that element will be set to 0. Choice A is incorrect because the if statement may only be true of some elements in the array. Choice B is incorrect because elements in the array will be set to 0 if their value is less than the variable `value`; it cannot be guaranteed that the array will be unchanged. Choice C is incorrect because the if statement checks if the value of an array element is less than the variable `value`, not greater than it. Choice D is incorrect because the for loop iterates over every element in the array.

Consider the following code segment.

```
class Classroom
{
    static int numAllStudents;
    static int numClasses;
    int numStudents;

    public Classroom(int n)
    {
        numStudents = n;
        numAllStudents += numStudents;
        numClasses++;
    }

    /* missing code */
}
```

Which of the following methods can be used to replace **`/* missing code */`** to correctly print out the average number of students in each classroom?

I.
```
public double avg()
{
    return (double)this.numAllStudents / this.numClasses;
}
```

II.

```
public double avg()

{

    return (double)numAllStudents / numClasses;

}
```

III.

```
public static double avg()

{

    return (double)numAllStudents / numClasses;

}
```

A. I only
B. II only
C. III only
D. I and II only
E. II and III only

Explanation:
The correct answer is E. Method II is an example of an instance method. Instance methods can access the static variables of a class. Method III is an example of a static method, which also can access the static variables of a class. The `this` keyword refers to the current object and its instance variables. `numAllStudents` and `numClasses` are static variables and cannot be referenced from the instance keyword `this`.

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

```
public class Rect
{
    private int x;   // x-coordinate of top-left point of rect

    private int y;   // y-coordinate of top-left point of rect

    private int width;   // width of rect

    private int height;  // height of rect


    public Rect()

    {

        x = 0;

        y = 0;
```

```
        width = 10;
        height = 15;
    }

    public Rect(int a, int b, int ww, int hh)
    {
        x = a;
        y = b;
        width = ww;
        height = hh;
    }
}
```

Which of the following methods correctly implements a method named `isGreater` that takes a `Rect` argument and returns true if its area is greater than the area of the `Rect` parameter, and false otherwise?

A.

```
public boolean isGreater(int ww, int hh)
{
    if (width * height > ww * hh)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

B.

```
public boolean isGreater(Rect other)
{
    return Rect.width * Rect.height > other.width * other.height;
}
```

C.

```
public boolean isGreater(int w, int h, int ww, int hh)
{
    if (w * h > ww * hh)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

D.

```
public boolean isGreater(Rect other)
{
    return width * height > other.width * other.height;
}
```

E.

```
public boolean isGreater(int ww, int hh)
{
    return this.width * this.height > ww * hh;
}
```

Explanation:
The correct answer is D. Methods can access the private data of a parameter that is a reference to an object that is the same type as the method's enclosing class. Choice A is incorrect because this method has two int parameters, not one Rect parameter. Choice B is incorrect because Rect.width and Rect.height are not legal references. Width and height are instance variables, not static variables, and cannot be referenced with the Rect classname. Choice C is incorrect because this method has four paramenters, not one Rect parameter. Choice E is incorrect because this method has two int parameters, not one Rect parameter.

Arrays

About 10–15% of the questions on your AP exam will cover the topic of Arrays. An *array* is an object that allows a single variable to refer to multiple values of a particular type.

Creating Arrays

An array type is created by appending `[]` to any other type. For example, `int[]` is the type for an array of integers, and `Rectangle[]` is the type for an array of `Rectangle` objects. Instead of using explicit constructor calls, arrays are created using the `new` keyword followed by an expression that specifies the type and size of the array.

```
int[] nums = new int[5];

Rectangle[] myRectangles = new Rectangle[12];
```

The first line will create an array with enough space for 5 integers, and the second line will create an array with enough space for 12 `Rectangle` objects. Once an array is created, its size can never be changed.

When all the values that are to be stored in an array are known, an *initializer list* can be used to create the array. In this case, the size of the array is omitted, since it is automatically inferred from the list provided. For example, the following line of code will create an array of length 5 that stores the integers shown:

```
int[] pi = new int[] {3, 1, 4, 1, 5};
```

When an array is created without an initializer list, all its elements are automatically initialized with default values. Numeric types are initialized to `0` (or `0.0`), Booleans to `false`, and reference types to `null`.

Array Access and Traversal

Every element in an array has an index. The smallest valid index in every array is 0, and the largest is one less than the length of the array. For example, if an array is initialized with the list `{ 42, 18, 33, 16, 7, 60 }`, it might be represented like this:

0	1	2	3	4	5
42	18	33	16	7	60

Indices are enclosed in square brackets following the name of the array and are used to access and modify elements in an array. Use of an index outside of the valid range will cause an `ArrayIndexOutOfBoundsException` to be thrown.

Traversing an array refers to systematically accessing all elements within it. This is usually accomplished using a `for` loop, although a `while` loop can also be used. The `length` attribute of an array provides the length of the array, so a typical `for` loop looks like this:

```
for (int i = 0; i < myArr.length; i++) {
  // myArr[i] will refer to the element at index i in myArr
}
```

Free Response Tip

Be very careful not to go out of bounds when traversing an array. Consider the length of the array and whether you are using `<` or `<=` in your loop condition. Also keep in mind whether you are accessing only a single index within the loop (such as `myArr[i]`), or multiple indices (`myArr[i]` and `myArr[i+1]`, for example).

Enhanced `for` Loops

Another form of a `for` loop, referred to as an *enhanced* `for` loop, or for-each loop, can be useful for traversing arrays. The syntax for an enhanced `for` loop is simpler than that of a regular `for` loop:

```
for (type name : list) {
  // loop body
}
```

In this code, `type` is the data type of the elements of the array, `name` is a variable that will refer to each element, and `list` is the name of the array itself. In each iteration of the loop, an element of the array is assigned to the variable `name`. Instead of referring to `myArr[i]`, the code in the loop body can simply refer to `name`.

There are two very important limitations to remember about enhanced `for` loops:

- The loop body does not have access to the index. This means that it can only refer to a single element within the array, and that it cannot be used if the index itself is needed.
- The variable `name` in the previous code is a local variable. If it is modified, the underlying element in the array does not change. Therefore, enhanced `for` loops can never be used to modify an array.

Standard Array Algorithms

There are several standard algorithms that traverse arrays and process the elements within:

- Find a maximum or minimum in an array
- Compute the sum or mean of the values in an array
- Determine whether some or all of the elements in an array have a certain property
- Count the number of elements in an array satisfying a certain condition
- Access all pairs of consecutive elements

- Shift or rotate all elements in an array to the right or left

You should be familiar with all of these algorithms and be able to implement them on arrays of many different types of values and objects.

Free Response Tip

Be aware of the asymmetric conditions involving "at least one element" and "all elements" problems. If you are checking to see if at least one element satisfies a certain condition, you can return `true` from a method as soon as you find a single match, but have to allow the loop to complete all its iterations before you can return `false`. This situation is reversed when you are checking to see if all elements satisfy a condition. In that case, you can return `false` as soon as you find a single element that does not satisfy the condition but must wait until the loop completes before you can return `true`.

Suggested Reading

- Horstmann. *Big Java: Early Objects*, 6th edition. Chapter 7.
- Gaddis & Muganda. *Starting Out with Java*, 4th edition. Chapter 7.
- Lewis & Loftus. *Java Software Solutions*, 9th edition. Chapter 8.
- Deitel & Deitel. *Java: How to Program, Early Objects*, 11th edition. Chapter 7.
- Liang. *Introduction to Java Programming, Brief Version*, 11th edition. Chapter 7.

Sample Arrays Questions

Consider the following code segment.

```
int[] ary = {2,3,5,7,11,13};
int twoDigitCount = 0;
for (int item : ary)
{
    if (item >= 10)
    {
        twoDigitCount++;
        item = item - 10;
    }
}
```

What are the contents of the array `ary` after this code segment executes?

A. 2 3 5 7 1 3
B. 2 3 5 7 11 13
C. The loop never terminates.
D. 1 2 3 4 5 6
E. 0 1 2 3 4 5

Explanation:
The correct answer is B. Assigning a new value to the enhanced for loop variable does not change the value stored in the array. The last two values in the array stay the same. There is no counting variable as there is in the traditional for loop.

The method `swap` takes three arguments: an `int[]` array and two `integer` index values. The method shall swap the two values at the specified indices.

```
/** @param b an int value such that b is a valid index in array a
 *  @param c an int value such that c is a valid index in array a
*/
public static void swap(int[] a, int b, int c)
{
    /* missing code */
}
```

Which of the following choices correctly completes the code segment marked **`/* missing code */`**.?

A.
```
int temp = a[b];
a[b] = a[c];
```

```
a[c] = temp;
```

B.
```
a[b] = a[c];
a[c] = a[b];
```

C.
```
int temp = a[b];
a[c] = a[b];
a[b] = temp;
```

D.
```
int temp = a[c];
a[c] = temp;
a[c] = a[b];
```

E.
```
a[c] = a[b];
int temp = a[b];
a[c] = temp;
```

Explanation:
The correct answer is A. A temporary variable is needed to hold the value of the array at index `b`. The value of index `c` is then copied into the array at index `b`. Finally, the temporary value is copied into the array at index `c`, completing the swap. Choice B is incorrect because a temporary variable is not used and the original value of the array at index `b` is overwritten. Choice C is incorrect because the second statement in the swap algorithm is reversed; the original value of the array at index c is never swapped. Choices D and E are incorrect because a temporary variable is not used correctly; the original value of the array at index `c` is never copied to the correct location.

```
private int[] numbers;


/** Precondition: numbers contains int values
 *  in no particular order.
*/
public void mystery(int num)
{
    int i = 0;
    while (i < num.length)
    {
        if (i % 2 == 1)
            num[i] += 10;
```

```
        else
            num[i] -= 10;
        i++;
    }
}
```

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

```
mystery(numbers);
```

A. The contents of numbers are unchanged.
B. The Boolean expression has a compiler error. `length` is a method and should have parenthesis, like: `num.length()`
C. The contents of `numbers` are replaced by all zeros.
D. All elements except the last one, are modified in the following way: elements at an odd index have 10 added to them, and elements at an even index are subtracted by 10.
E. All elements are modified in the following way: elements at an odd index have 10 added to them, and elements at an even index are subtracted by 10.

Explanation:
The correct answer is E. The array loops from index 0 to the final index in the array. Odd index values are identified by the modulus operation and the check of whether they are not evenly divisible by 2. Choice A is incorrect because the value of the array argument is a reference, so changes to `num` have the effect of also changing `numbers`. Choice B is incorrect because. `num.length` is a field reference for arrays, not a method; no set of parenthesis is appropriate. Choice C is incorrect because the content of each element in the array is modified by either +10 or -10, not 0. Choice D is incorrect because `num.length` returns the count of the number of elements in the array, which is one greater than the last index value.

ArrayList

About 2.5–7.5% of the questions on your exam will fall under the `ArrayList` category.

Like an array, an `ArrayList` is an object that stores multiple elements of the same type. Unlike an array, however, its size is not set at creation and can change dynamically.

Creating and Storing `ArrayList` Objects

An `ArrayList` has type `ArrayList<E>`, where `E` is the reference type of object it will store. To create an `ArrayList`, call its default constructor:

```
ArrayList<E> myList = new ArrayList<E>();
```

This will create and store an empty `ArrayList` intended to store objects of type `E`.

`ArrayList` Methods

As with arrays, the elements in an `ArrayList` are indexed starting at 0. `ArrayList` includes many methods for storing, accessing, and modifying the elements stored within the list. The table below shows the most important ones.

Method	Description
`int size(int x)`	Returns the number of elements in the list
`boolean add(E obj)`	Adds `obj` to the end of the list and returns `true`
`void add(int index, E obj)`	Adds obj to the list at position `index`, moving all elements previously at positions `index` and higher to the right
`E get(int index)`	Returns the element at position `index`
`E set(int index, E obj)`	Replaces the element at position `index` with `obj`, and returns the removed element
`E remove(int index)`	Removes and returns the element at position `index`, moving all elements at position `index + 1` and higher to the left

Note the use of the type `E` as a parameter type and return type in many of these methods. Keep in mind that this is the type that was used in the declaration and creation of the `ArrayList`. For example, consider the following `ArrayList`:

```
ArrayList<String> studentNames = new ArrayList<String>();
```

In this example, the `set` method of `studentNames` has parameters of type `int` and `String`, and returns a `String`.

`ArrayList` Algorithms

The standard algorithms used to traverse and process arrays apply to `ArrayList` objects as well. In particular, both standard and enhanced `for` loops can be used for traversal. There are, however, a few special considerations and modifications to keep in mind:

- The length of an `ArrayList` is obtained from its `size()` method.
- Accessing the element at position `i` uses the `.get(i)` method call.
- Since the size of an `ArrayList` can change by adding and removing elements, the algorithms previously discussed can be extended to perform these actions.
- The size of an `ArrayList` cannot be changed within the body of an enhanced `for` loop; this will result in an error.

Free Response Tip

When removing an element from an `ArrayList` in the context of a for loop, be very careful to adjust for the removal, or an element will be skipped. If you are using a typical "counting up" loop, make sure to decrement the loop variable after each removal. Alternatively, change the loop to iterate backwards through the list; this will ensure that removals do not affect the remaining iterations.

Searching and Sorting

Searching and sorting lists of data are classical computer science problems, and as such, many algorithms have been developed to address them.

Sequential search, or *linear search*, is a standard and universal algorithm. It can be used on any type of list. It proceeds by sequentially checking each item in a list to see if it matches the target item. As soon as a match is found, the search can terminate. If the end of the list is reached with no match having been found, the result is negative.

Selection sort and *insertion sort* are algorithms that use nested loops to sort lists. Exact comparisons and computations of their running times are beyond the scope of AP Computer Science A, but informal analyses can be conducted by examining their loops and counting the number of iterations they execute with sample data.

More sophisticated algorithms for both searching and sorting will be discussed later in the context of recursion.

Suggested Reading

- Horstmann. *Big Java: Early Objects*, 6th edition. Chapter 7.
- Gaddis & Muganda. *Starting Out with Java*, 4th edition. Chapter 7.
- Lewis & Loftus. *Java Software Solutions*, 9th edition. Chapter 5.
- Deitel & Deitel. *Java: How to Program, Early Objects*, 11th edition. Chapter 7.
- Liang. *Introduction to Java Programming, Brief Version*, 11th edition. Chapter 7.

Sample `ArrayList` Questions

Consider the following code segment.

```
ArrayList<String> numbers = new ArrayList<String>();

numbers.add("zero");

numbers.add("one");

numbers.add("2");

numbers.add(1, "three");

numbers.set(2, "four");

numbers.add("five");

numbers.remove(0);

numbers.remove(0);
```

Which of the following best describes the contents of `numbers` after the code segment has been executed?

A. {"zero", "one", "three", "four", "five"}
B. {"four", "2", "five"}
C. {"three", "four", "one", "five"}
D. {"2", "three", "four", "five"}
E. {"zero", "three", "one", "five"}

Explanation:

The correct answer is B. First, `zero` and `one` are appended onto an empty ArrayList. The size of the list (2) is then appended to the ArrayList as a string ("2"). Then `three` is inserted in between `zero` and `one`. `four` replaced the value of the second element (`one`). five is appended to the end of the ArrayList. Finally, the first two elements are popped off.

Which of the following statements most accurately contrasts an `ArrayList` with an array?

A. An `ArrayList` object is mutable and may contain primitive values or object references whereas an array contains object references.
B. An `ArrayList` object is mutable and contains object references whereas an array contains either primitive values or object references.
C. An `ArrayList` object is immutable and contains object references whereas an array may contain either primitive values or object references.
D. An `ArrayList` object is mutable and may contain primitive values or object references whereas an array contains primitive values.
E. An `ArrayList` object is immutable and may contain primitive values or object references whereas an array may contain either primitive values or object references.

Explanation:
The correct answer is B. An `ArrayList` object is mutable and contains object references whereas an array contains either primitive values or object references.

A key concept in the insertion sort algorithm is the ability to correctly insert an element into a sorted list. Which of the following `insert` methods correctly inserts the string `s`, given by the first argument, into the sorted list `words`, given by the second argument?

I.

```
/**
 * Precondition: words is sorted lexicographically in increasing order
 */
public static void insert(String s, ArrayList<String> words)
{
    int i = 0;
    boolean done = false;
    while (i < words.size() && !done)
    {
        if (s.compareTo(words.get(i)) < 0)
        {
            words.add(i, s);
            done = true;
        }
        i++;
    }
    if(!done)
    {
        words.add(s);
    }
}
```

II.

```
/**
 * Precondition: words is sorted lexicographically in increasing order
 */
public static void insert(String s, ArrayList<String> words)
{
    int i = 0;
    boolean done = false;
    for (int i = 0; i < words.size() || !done; i++)
    {
        if (s.compareTo(words.get(i)) <= 0)
        {
            words.add(i, s);
            done = true;
        }
    }
}
```

III.

```
/**
 * Precondition: words is sorted lexicographically in increasing order
 */
public static void insert(String s, ArrayList<String> words)
{
    int i = 0;
    while (i < words.size())
    {
        if (s.compareTo(words.get(i)) >= 0)
```

```
            {
                words.add(s);
            }
            i++;
        }
    }
```

A. I
B. II
C. III
D. I and II
E. I and III

Explanation:
The correct answer is A. This is the only method that correctly inserts the string into the list. The algorithm works by using a loop to iterate over each item in the list until it finds the first item in the list that lexicographically precedes the string `s`. The string is inserted into the list in front of this item. Method II's for loop test is faulty, and will continue to iterate through the list, even after the string `s` is inserted into the list. The string `s` may be incorrectly inserted into the list multiple times. The if statement in method III would be appropriate for inserting the string `s` into a lexicographically sorted list in decreasing order.

2D Arrays

About 7.5–10% of the questions on your exam will cover 2D Arrays.

A *two-dimensional (2D) array* is an array of arrays, often thought of as a rectangular array of values, with rows and columns. For the purposes of AP Computer Science A, all 2D arrays are rectangular, so that each row has the same length.

Creating and Accessing 2D Arrays

The notation for creating and storying a 2D array is simply an extension of the notation for a 1D array. In the statement below, `type` is the type of value being stored, `rows` is the number of rows, and `cols` is the number of columns.

```
type[][] name = new type[rows][cols];
```

A 2D array can be initialized with items that are each initializer lists for a 1D array, as in the following example:

```
int[][] values = new int[][] {{1, 2, 3}, {4, 5, 6}};
```

If `arr` is a 2D array, the expression `arr[r]` refers to the element at position `r` in `arr`. But this element is itself a 1D array, so it can be further indexed with the expression `arr[r][c]`.

The two subscripts should be thought of as the row and column positions in the rectangular array. The row and column indices both start at `0`. The following example creates the array shown:

	0	1	2
0	1	2	3
1	4	5	6

Traversing 2D Arrays

Traversal of a 2D array is generally accomplished using nested loops. In a *row-major* traversal, the outer loop iterates through each row, while the inner loop iterates through each column in the row. In a *column-major* traversal, the order of the loops is reversed.

The number of rows in the 2D array `arr` is `arr.length`. The number columns can be accessed using `arr[x].length`, where `x` is any valid row index.

Free Response Tip

If you use an enhanced for loop as the outer loop in a 2D array traversal, keep in mind that the elements being accessed are themselves arrays. For example, if the 2D array `arr` consists of `int` values, the traversal might look like this:

```
for (int[] row: arr) {
      for (int x: row) {
      // traversal logic
     }
}
```

All the standard 1D array algorithms can be applied to 2D arrays. For example, the following method will find the average of all integers in a 2D array.

```
// precondition: values contains at least one row and one column
public static double average(int[][] values) {
  double total = 0;
  int count = 0;
  for (int r = 0; r < values.length; r++) {
    for (int c = 0; c < values[r].length; c++) {
      total += values[r][c];
      count++;
    }
  }
}
```

Suggested Reading

- Horstmann. *Big Java: Early Objects*, 6th edition. Chapter 7.
- Gaddis & Muganda. *Starting Out with Java*, 4th edition. Chapter 7.
- Lewis & Loftus. *Java Software Solutions*, 9th edition. Chapter 8.
- Deitel & Deitel. *Java: How to Program, Early Objects*, 11th edition. Chapter 7.
- Liang. *Introduction to Java Programming, Brief Version*, 11th edition. Chapter 8.

Sample 2D Arrays Questions

Consider the following code segment, which contains line numbers. The loop is intended to return the sum of both diagonals in a 2d array. A precondition is that the number of rows and columns in the 2d array are equal. However, the program does not produce correct output.

```
/** Preconditions:
 *      array.length > 0
 *      array.length == array[0].length
*/
 1 public int sumDiagonals(int[][] array)
 2 {
 3   int sum = 0;
 4   for (int i = 1; i < array.length; i++)
 5   {
 6        for (int j = 1; j < array[0].length; j++)
 7        {
 8            if (i == j || j == array[0].length-1-i)
 9            {
10                sum += array[i][j];
11            }
12        }
13    }
14    return sum;
15 }
```

Which of the below solutions will fix the program?

A. Change the name of the parameter throughout the method to `ary`.
B. In lines 4 and 6, initialize the variables `i` and `j` to 0 instead of 1.
C. In lines 4 and 6, change the test of each loop to `array[0].length` and `array.length`, respectively.
D. In line 8, change the first Boolean expression to `i != j`.
E. Before the method returns, add a second nested for loop to account for the diagonal that spans from the top-right to the bottom-left corner.

Explanation:
The correct answer is B. Both for loops should be initialized at 0, so that the iteration also includes the first row and first column. Choice A is incorrect because the identifier `array` is legal in Java. Changing the name would not affect the output of the program. Choice C is incorrect because the change would have no effect on the output of the program. Choice D is incorrect because the current Boolean expression `i == j` is correctly testing whether an element lies on the diagonal that spans from the top-left to the bottom-right. Choice E is incorrect because no second loop is necessary. The disjunction of Boolean expressions on Line 8 tests whether an element lies on either diagonal.

A two-dimensional array of integers is used to model a *w x h* visualization of single digits where the visualization has a width `w` and height `h`. If the array is named `viz`, and its width is `600` and its height is `400`, which of the below lines of code correctly places the single digit `5` approximately in the center of the visualization?

A. `viz[400,600] = 5`
B. `viz[600][400] = 5`
C. `viz[300][200] = 5`
D. `viz[300,200] = 5`
E. `viz[200][300] = 5`

Explanation:
The correct answer is E. In a two-dimensional array, the row is indexed first, followed by the column. The center of the array is at row 200, column 300. Choice A incorrectly indexes a two-dimensional array. Choice B is incorrect because this problem is asking for a 5 at the center of the visualization, and this choice is indexing row 600, column 400. Choice C is incorrect because in a two-dimensional array, the row is indexed first, followed by the column; in this choice, the column is indexed first. Choice D incorrectly indexes a two-dimensional array.

Consider the following `Util` class that contains a `mystery` method:

```
public class Util
{
    public static boolean mystery(int[][] a)
    {
        final int NUM = 0;

        for (int[] row : a)
        {
            for (int i = 0; i < row.length; i++)
            {
                if (a[i] != NUM)
```

```
                {
                    return false;
                }
            }
        }
        return true;
    }
}
```

Which of the following choices best describes the method `mystery`?

A. The method transforms every element in the 2d array into 0, and returns true on success, false otherwise.
B. The method returns true if there is an element in the 2d array that is nonzero.
C. The method counts the number of rows in the 2d array that are composed entirely of zeros.
D. The method counts the number of zeros in the 2d array.
E. The method returns true if every element in the 2d array is zero, false otherwise.

Explanation:
The correct answer is E. The traversal of the 2d array uses an outer for loop to access the rows of the 2d array, and an inner loop to access each element of the row. The method does not alter any value in the 2d array; returns true if all elements in the 2d array are zero; traverses each row in the 2d array but does not count how many rows are composed entirely of zeros; and immediately returns upon finding the first nonzero element.

Inheritance

About 5–10% of questions on your AP Computer Science A exam will cover the topic of inheritance.

In many situations, multiple classes share various characteristics and behaviors. There are various ways to represent these shared attributes so that code can be shared and reused between the classes. One of these ways is referred to as inheritance.

The Subclass Relationship

When a class is declared, it can `extend` another class:

```
public class B extends A { … }
```

This declaration creates an *inheritance* relationship between `A` and `B`. `B` is said to be a *subclass* of `A`. Equivalently, `A` is a *superclass* of `B`. A subclass can only have a single superclass, but a superclass can have many subclasses.

With inheritance, the attributes and behaviors of the superclass are automatically shared with the subclass. In the previous example, if `A` has a method called `doSomething`, and `x` is a variable of type `B`, then the call `x.doSomething()` is valid and will work as expected.

Free Response Tip

When writing a subclass, do not declare any instance variables that store the same information, or have the same name as instance variables that exist in the superclass. In addition, be sure that you only duplicate methods from the superclass if it needs to be overridden for changed or added functionality.

Subclass Constructors and `super`

Constructors are not inherited from a superclass. In general, subclass constructors must be explicitly written in the subclass. Within a subclass constructor, the `super` keyword can be used to call a superclass constructor, as in the following example:

```
public class Pet {
  private String name;

  public Pet() {
    name = "";
  }

  public Pet(String petName) {
    name = petName;
  }
}

public class Dog extends Pet {
  public Dog(dogName, dogBreed) {
    private String breed;
    super(dogName);
    breed = dogBreed;
  }
}
```

In this example, the `Dog` constructor calls the Pet constructor and passes `dogName` to it so that it can be stored in the `name` instance variable.

The call to a superclass constructor must be the first line in a subclass constructor. If the superclass has a default (no parameter) constructor, the call to `super` is optional; the default constructor will be called automatically if it is omitted.

It is very important to remember that although a subclass inherits all the attributes and behaviors of its superclass, it cannot access private variables or methods from the superclass.

Overriding Methods

When a subclass and its superclass both contain a method with the same signature, the method in the subclass is said to *override* the method in the superclass. Any call to the method from an object of the subclass type will result in the subclass method being called. Within the subclass, the `super` keyword can be used with dot notation to refer explicitly to a superclass method.

To illustrate this, consider the following classes and variable declarations.

```
public class Fruit {
  public String getInfo() {
    return "Juicy";
  }
}

public class Peach extends Fruit {
  public String getInfo() {
    return "Soft";
  }
}

public class Apple extends Fruit {
  public String getInfo() {
    return "Crunchy and " + super.getInfo();
  }
}

Peach p = new Peach();
Apple a = new Apple();
```

The call `p.getInfo()` will return `"Soft"`, while `a.getInfo()` will return `"Crunchy and Juicy"`.

Polymorphism

When two classes have an inheritance relationship, an *is-a* relationship is created between the classes. One of the most important consequences of this is that an object of the subclass type can be assigned to a reference of the superclass type. For example, consider the following declarations:

```
public class Vehicle { … }
public class Car extends Vehicle { … }
```

This creates the following relationship: A `Car` is-a `Vehicle`. Because of this relationship, the following assignments are all valid:

```
Vehicle myVehicle = new Vehicle();
Car myCar = new Car();
Vehicle myOtherCar = new Car();
```

The first two are straightforward, while in the third statement, a `Car` object is assigned to a `Vehicle` reference variable. This is valid since `Car extends Vehicle`, and so a `Car` is-a `Vehicle`. Note that the opposite direction would not be valid; you could not store a `Vehicle` object in a `Car` reference, since the is-a relationship does not exist in that direction.

Being able to do this is very useful in a variety of situations. Consider, for example, an object of type `ArrayList<Vehicle>`. Then objects of both `Vehicle` and `Car` types can be added to this list. For

another example, think about a method that declares a formal parameter of type `Vehicle`. Both a `Vehicle` and a `Car` could be passed as actual parameters to this method.

When a superclass is itself a subclass of another class, an inheritance hierarchy is formed. In particular, if `A` and `B` are classes such that an `A` is-a `B`, and a B is-a `C`, then an `A` is-a `C` as well. In other words, the is-a relationship is transitive.

When an object is stored in a reference of a different type, there are two important facts concerning the way method calls are treated at compile- and run-times.

- The methods declared in, or inherited by, the reference type determine the validity of a method call as decided by the compiler.
- The method in the object type determines what code is actually executed at run-time.

To demonstrate, consider the following declarations:

```
public class Person {

  private String myName;

  public Person(String name) {
    myName = name;
  }
  public String getTitle() {
    return "Person";
  }
  public String getName() {
    return name;
  }
}

public class Student {
  public Student(String name) {
    super(name);
  }

  public String getTitle() {
    return "Student";
  }

  public double getGPA() {
    return 3.8;
  }
}

Person stu = new Student("Cameron");
System.out.println(stu.getName()); // Line 1
System.out.println(stu.getTitle()); // Line 2
System.out.println(stu.getGPA()); // Line 3
```

First consider Line 1. The method call `stu.getName()` is valid, since the reference type of `stu` is `Person`, and `Person` declares a `getName` method. The object itself is a `Student`, which inherits the `getName` method, so when it is executed the inherited code is used.

Now consider Line 2. As before, the method call `stu.getTitle()` is valid since the reference type of `stu` is `Person`, and `Person` declares a `getTitle` method. Note that at this stage the presence of the overridden `getTitle` method in `Student` is irrelevant. It is only at run-time, when the code is being executed, that the object is examined, and the overridden method found and run.

Finally, consider Line 3. The reference type of `stu` is `Person`, and `Person` does not declare or inherit a `getGPA` method, so this method call will not compile. It does not matter that the object happens to actually be of type `Student`.

If a class does not explicitly extend another class, it automatically extends `Object`, a class that is included in the Java library. Therefore, `Object` is the ultimate superclass of every other Java class. This means that any type of object can be assigned to an `Object` reference.

The `Object` class has the following two methods, which are inherited by all other classes:

Method	Description
`boolean equals(Object other)`	Returns `true` if `other` is an alias of `this`, and `false` otherwise.
`String toString()`	Returns a string representation of `this`, including its type and location in memory.

These implementations are usually not useful, so many classes override the implementations of the `equals` and `toString` methods to provide appropriate and type-specific functionality.

Suggested Reading

- Horstmann. *Big Java: Early Objects*, 6th edition. Chapter 9.
- Gaddis & Muganda. *Starting Out with Java*, 4th edition. Chapter 10.
- Lewis & Loftus. *Java Software Solutions*, 9th edition. Chapter 9.
- Deitel & Deitel. *Java: How to Program, Early Objects*, 11th edition. Chapter 9.
- Liang. *Introduction to Java Programming, Brief Version*, 11th edition. Chapter 11.

Sample Inheritance Questions

Consider the following class definitions.

```
public class Idea1
{
    public void total(int n)
    {
        System.out.print(n+3);
    }
}

public class Idea2 extends Idea1
{
    public void total(int n)
    {
        n += 2;
        System.out.print(n);
    }
}
```

The following code segment appears in a class other than `Idea1` and `Idea2`.

```
Idea1 t = new Idea2();
t.total(5);
```

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

A. 7
B. 8
C. 5
D. 10
E. 15

Explanation:
The correct answer is A. Because there is no `super` call, only the `total` method in class `Idea2` is executed. Choice B is incorrect because the `total` method in `Idea1` is overridden by the `total` method in `Idea2`; the `total` method in `Idea1` is not called. Choice C is incorrect because the `total` method in

`Idea2` contains a += operator, which adds 2 to the passed in value of 5. Choices D and E are incorrect because the `total` method in `Idea1` is overridden by the `total` method in `Idea2`; the `total` method in `Class1` is not called.

Consider the following classes, which model a vehicle and car.

```
public class Vehicle
{

    private int xloc = 0;
    private int yloc = 0;

    public void goForward()
    {
        System.out.println("Moving forward");
    }

}

public class Car extends Vehicle
{

    public void goForward()
    {
        System.out.println("Wheels are moving");

        xloc += 3;
    }
}
```

Which of the following statements and method calls will create an instantiated object that produces the following output using polymorphism?

```
Wheels are moving
```

```
Wheels are moving
Location: 6,0
```

A.

```
Vehicle v1 = new Car();
v1.goForward();
v1.goForward();
System.out.println("Location:" + v1.xloc + "," + v1.yloc);
```

B.

```
Car c1 = new Car();
c1.goForward();
c1.goForward();
System.out.println("Location:" + c1.xloc + "," + c1.yloc);
```

C.

```
Car v1 = new Vehicle();
v1.goForward();
v1.goForward();
System.out.println("Location:" + v1.xloc + "," + v1.yloc);
```

D. A and C
E. A, B, and C

Explanation:
The correct answer is A. An instantiated object for Vehicle is correctly instantiated. The method `goForward` at runtime will execute the definition provided in the Vehicle class. Choice B is incorrect because the instantiated object is only a reference to the Car class, not Vehicle. Choice C is incorrect because the instantiation is written backwards. A reference variable typically has the type of a superclass and is instantiated with a subclass.

Consider the following `BankAccount` class, which is now extended by the class `SuperSavingsAccount`.

```
class BankAccount
{
```

```
    private double balance = 5.00;

    /**
     * @param balance: starting balance of the account
     */
    public BankAccount(double balance)
    {
        /* implementation not shown */
    }

    public void deposit(double d)
    {
        balance += d;
    }

    public void withdraw(double w)
    {
        if (balance >= w)
        {
            balance -= w;
        }
    }

    public double getBalance()
    {
        return balance;
    }
}

class SuperSavingsAccount extends BankAccount
```

```
{
    private double interestRate;


    /**
     * Constructor.
     * @param qualify - the value true sets the account's interest rat
e at 20%, otherwise is set to 5%
     */
    public SuperSavingsAccount(boolean qualify)
    {
        if (qualify)
        {
            interestRate = .2;
        }
        else
        {
            interestRate = .05;
        }
    }
}
```

What is the value of the account's starting balance if the following statement is used to create an instance of a `SuperSavingsAccount` object?

```
BankAccount ba = new SuperSavingsAccount(true);
```

A. The code segment has a compiler error. There is no starting balance because the constructor to `BankAccount` is never called.
B. $0.00
C. true
D. 20%
E. $5.00

Explanation:
The correct answer is A. The constructor in the superclass `BankAccount` must be explicitly called in the subclass `SuperSavingsAccount` because `BankAccount` does not have a no-argument constructor.

Afterwards, the starting balance will be set to the value of the argument passed to the `BankAccount` constructor. When a subclass's constructor does not explicitly call a superclass's constructor using `super`, Java inserts a call to the superclass's no-argument constructor. However, there is no no-argument constructor in `BankAccount`, so the code segment has a compiler error.

Recursion

Finally, *5–7.5%* of the questions on your AP exam will cover the topic of Recursion.

Recursive Methods

A method is called *recursive* if it calls itself. On the AP Computer Science A exam, you will be expected to understand and trace the logic of recursive methods, but you will not have to write any yourself.

A *base case* in a recursive method is an execution path that does not result in the method calling itself. Every recursive method must have at least one base case, or it will eventually cause an error when the number of times it has been called exceeds a threshold (which differs by system configuration).

Every recursive call has its own set of parameters and local variables. When a task is done recursively, the parameters are used to track the progress of the task. This is analogous to the way a loop variable keeps track of the progress through a `for` loop. In fact, every recursive method can theoretically be replicated using loops instead of recursion, although doing so may not be straightforward.

Tracing Recursive Method Calls

When tracing the execution of recursive methods, it is important to keep track of the parameters of each call. This is best demonstrated with an example:

```
public int calc(int x) {
  if (x == 0) {
    return 0;
  }
  else if (x < 0) {
    return -2 + calc(x + 1);
  }
  else {
    return 2 + calc(x - 1);
  }
}
```

This method has two recursive calls, and a single base case (when `x == 0`).

Consider the method call `calc(2)`. The parameter `x` is `2`, so we will refer to this as the x=2 method. The `else` path is taken, so the expression `2 + calc(1)` needs to be calculated. Therefore, `calc` is called with a parameter of `1`. In this x=1 method, the `else` path is taken again, so `calc` is called with a parameter of `0`. In this method the expression in the initial `if` statement is true, so `0` is immediately returned. Control now returns to the x=1 call, which can now calculate and return `2 + 0`, or `2`. Finally, control returns to the initial x=2 method. The expression `2 + calc(1)` evaluates to `2 + 2`, or `4`, so this is the final value returned by the initial call to `calc(2)`.

Recursive techniques can be used to traverse strings and arrays. Commonly, a parameter is used to represent a position in the string (or array), which is incremented in successive recursive calls. In this way,

the entire string (or array) can be reached. The base case is when this position has reached or exceeded the valid indices in the string (or array). For example, the following method will determine whether or not an array of integers contains the given value:

```
public boolean contains(int[] arr, int index, int target) {
  if (index >= arr.length) {
    return false;
  }
  else if (arr[index] == target) {
    return true;
  }
  else {
    return contains(arr, index + 1, target);
  }
}
```

The initial call to the method needs to specify index `0` as a starting point. For example, to search the array `myArray` for the value `5`, you would call `contains(myArray, 0, 5)`.

Another technique useful for strings is to pass substrings to the recursive call. This eliminates the need for a parameter representing a position. For example, the following method will reverse a string:

```
public String reverse(String str) {
  if (str.length() <= 1) {
    // a string of length 0 or 1 is the same as its reversal
    return 1;
  }
  else {
    // reverse everything except the first character, and concatenate
    // it to the end of the result
    return reverse(str.substring(1)) + str.substring(0, 1);
  }
}
```

Recursive Searching and Sorting

If an array or `ArrayList` is sorted, it can be searched using the *binary search* algorithm. This is a search algorithm that is, in most cases, significantly more efficient than sequential/linear search. It is worth emphasizing once again, however, that it can only be applied when a list is already in sorted order.

Binary search works by keeping track of the beginning and ending positions of the section of the list that remains to be searched. By taking the average of these two positions, and then examining the item at the position calculated, half of the list can be eliminated in the next iteration. This continues until the item has been found, or until there is no part of the list remaining, in which case the conclusion is that the item does not exist in the list.

This algorithm can be implemented either iteratively or recursively. In the iterative implementation, local variables keep track of the starting and ending positions, and a `while` loop controls the iteration of the

logic. In the recursive implementation, the start and end positions are kept track of via a pair of recursive call parameters.

There are also various recursive sorting algorithms, some of which are much more efficient than the insertion sort and selection sort algorithms mentioned earlier. One of the most common is *merge sort*. This algorithm relies on *merging*, which is the process of taking two lists that are already in sorted order and combining them into a single larger sorted list.

Suggested Reading

- Horstmann. *Big Java: Early Objects*, 6th edition. Chapter 13.
- Gaddis & Muganda. *Starting Out with Java*, 4th edition. Chapter 15.
- Lewis & Loftus. *Java Software Solutions*, 9th edition. Chapter 12.
- Deitel & Deitel. *Java: How to Program, Early Objects*, 11th edition. Chapter 18.
- Liang. *Introduction to Java Programming, Brief Version*, 11th edition. Chapter 18.

Sample Recursion Questions

Consider the following recursive method.

```
public static int mystery(int n)
{
    if (n <= 1)
    {
        return n;
    }
    else
    {
        return mystery(n/2) + mystery(n/3);
    }
}
```

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

A. 8
B. 13
C. 7
D. 4
E. 5

Explanation:

The correct answer is D. The call `mystery(10)` returns the expression `mystery(5)+mystery(3)`, which in turn returns the expression `(mystery(2)+mystery(1))+mystery(3)`, which is expanded to `((mystery(1)+mystery(0))+mystery(1))+(mystery(1)+mystery(1))`, which evaluates to, via the base case of the recursion, `1+0+1+1+1` = 4.

Consider a method `balancedParens` that is intended to use recursion to return whether a String is composed of zero or more left parentheses, immediately followed by an equal number of right parentheses. For example, the Strings "", "()", and "((((()))))" are balanced, while the Strings "(", "()()", and "((())" are not balanced.

```
public static boolean balancedParens(String s)
{
```

```
    if (s.length() == 0)
    {
        return /* missing code */ ;
    }
    else if (s.length() == 1)
    {
        return /* missing code */ ;
    }
    else if (s.charAt(0) != '(' || s.charAt(s.length()-1) != ')')
    {
        return /* missing code */ ;
    }
    else
    {
        return balancedParens(s.substring(1,s.length()-1));
    }
}
```

What are the Boolean values that should replace `/* missing code */` so that the `balancedParens` method works correctly?

A. false false false
B. false false true
C. true false false
D. true true true
E. true false true

Explanation:
The correct answer is C. The first if statement should return true, because the empty string is balanced by definition. The second if statement should return false, because a string of one character cannot be balanced. The third if statement should return false because a balanced string must begin with (and end with).

Consider the following recursive implementation of a binary search algorithm.

```
/**
 * Precondition:
 */
public static int binarySearch(int v, int[] numbers, int start, int en
d)
{
    int mid = (start + end) / 2;

    if (end < start)
    {
         return -1;
    }

    if (v > numbers[mid])
    {
        return binarySearch(v, numbers, start, mid-1);
    }

    if (v < numbers[mid])
    {
        return binarySearch(v, numbers, mid+1, end);
    }

    if (v == numbers[mid])
    {
        return mid;
    }

    return -1;
}
```

Which of the following choices best describes the precondition of the method? To search for the value `17` in some array `arr`, the initial call to the method will be: `binarySearch(17, arr, 0, arr.length-1)`.

A. The array `numbers` is unsorted.
B. The array `numbers` is already sorted in increasing order.
C. The array `numbers` is already sorted in decreasing order.
D. The method will crash if the passed in array `numbers` is empty.
E. The array `numbers` contains only positive integers.

Explanation:

The correct answer is C. This method looks at the value in the middle of the sorted array. If the value being searched for is greater than this value, the method recursively searches the left portion of the sorted array, implying that the array must be in decreasing order. A binary search algorithm assumes that the array being searched is already in sorted order. If the passed in array is empty, the value of the parameter end will be less than `start`, and the method will immediately return -1 rather than crash. Finally, this binary search implementation can work with both positive and negative integers. The value of -1 is being returned by the method to indicate if the value being searched for is not found.