CHAPTER 1

Introductory Java Language Features

Fifty loops shalt thou make ...
—Exodus 26:5

Chapter Goals

- Packages and classes
- Types and identifiers
- Operators
- Input/output
- Storage of numbers
- Binary and hexadecimal numbers
- Control structures
- Errors and exceptions

The AP Computer Science course includes algorithm analysis, data structures, and the techniques and methods of modern programming, specifically, object-oriented programming. A high-level programming language is used to explore these concepts. Java is the language currently in use on the AP exam.

Java was developed by James Gosling and a team at Sun Microsystems in California; it continues to evolve. The AP exam covers a clearly defined subset of Java language features that are presented throughout this book, including some new features of Java 5.0 that were tested for the first time in May 2007. The College Board website, *http://www.collegeboard.com/student/testing/ap/subjects.html*, contains a complete listing of this subset.

Java provides basic control structures such as the `if-else` statement, `for` loop, for-each loop, and `while` loop, as well as fundamental built-in data types. But the power of the language lies in the manipulation of user-defined types called objects, many of which can interact in a single program.

PACKAGES AND CLASSES

A typical Java program has user-defined classes whose objects interact with those from Java class libraries. In Java, related classes are grouped into *packages*, many of which are provided with the compiler. You can put your own classes into a package—this facilitates their use in other programs.

The package `java.lang`, which contains many commonly used classes, is automatically provided to all Java programs. To use any other package in a program, an `import` statement must be used. To import all of the classes in a package called `packagename`, use the form

```
import packagename.*;
```

To import a single class called `ClassName` from the package, use

```
import packagename.ClassName;
```

Java has a hierarchy of packages and subpackages. Subpackages are selected using multiple dots:

```
import packagename.subpackagename.ClassName;
```

The `import` statement allows the programmer to use the objects and methods defined in the designated package. By convention Java package names are lowercase. The AP exam does not require knowledge of packages. You will not be expected to write any `import` statements.

A Java program must have at least one class, the one that contains the *main method*. The java files that comprise your program are called *source files*.

A *compiler* converts source code into machine-readable form called *bytecode*.

Here is a typical source file for a Java program.

```
/* Program FirstProg.java
   Start with a comment, giving the program name and a brief
   description of what the program does. */

import package1.*;
import package2.subpackage.ClassName;

public class FirstProg  //note that the file name is FirstProg.java
{
    public static type1 method1(parameter list)
    {
        < code for method 1 >
    }
    public static type2 method2(parameter list)
    {
        < code for method 2 >
    }
        ...

    public static void main(String[] args)
    {
        < your code >
    }
}
```

NOTE

1. All Java methods must be contained in a class, and all program statements must be placed inside a method.

2. Typically, the class that contains the `main` method does not contain many additional methods.
3. The words `class`, `public`, `static`, `void`, and `main` are *reserved words*, also called *keywords*.
4. The keyword `public` signals that the class or method is usable outside of the class, whereas `private` data members or methods (see Chapter 2) are not.
5. The keyword `static` is used for methods that will not access any objects of a class, such as the methods in the `FirstProg` class in the example on the previous page. This is typically true for all methods in a source file that contains no *instance variables* (see Chapter 2). Most methods in Java do operate on objects and are not static. The `main` method, however, must always be static.
6. The program shown on the previous page is a Java *application*. This is not to be confused with a Java *applet*, a program that runs inside a web browser or applet viewer. Applets are not part of the AP subset.

TYPES AND IDENTIFIERS

Identifiers

An *identifier* is a name for a variable, parameter, constant, user-defined method, or user-defined class. In Java an identifier is any sequence of letters, digits, and the underscore character. Identifiers may not begin with a digit. Identifiers are case-sensitive, which means that `age` and `Age` are different. Wherever possible identifiers should be concise and self-documenting. A variable called `area` is more illuminating than one called `a`.

By convention identifiers for variables and methods are lowercase. Uppercase letters are used to separate these into multiple words, for example `getName`, `findSurfaceArea`, `preTaxTotal`, and so on. Note that a class name starts with a capital letter. Reserved words are entirely lowercase and may not be used as identifiers.

Built-in Types

Every identifier in a Java program has a type associated with it. The *primitive* or *built-in* types that are included in the AP Java subset are

`int`	An integer. For example, `2`, `-26`, `3000`
`boolean`	A boolean. Just two values, `true` or `false`
`double`	A double precision floating-point number. For example, `2.718`, `-367189.41`, `1.6e4`

(Note that primitive type `char` is not included in the AP Java subset.)

Integer values are stored exactly. Because there's a fixed amount of memory set aside for their storage, however, integers are bounded. If you try to store a value whose magnitude is too big in an `int` variable, you'll get an *overflow error*. (Java gives you no warning. You just get a wrong result!)

An identifier, for example a *variable*, is introduced into a Java program with a *declaration* that specifies its type. A variable is often initialized in its declaration. Some examples follow:

```
int x;
double y,z;
boolean found;
int count = 1;                  //count initialized to 1
double p = 2.3, q = 4.1;        //p and q initialized to 2.3 and 4.1
```

One type can be cast to another compatible type if appropriate. For example,

```
int total, n;
double average;
    ...
average = (double) total/n;     //total cast to double to ensure
                                //real division is used
```

Alternatively,

```
average = total/(double) n;
```

Assigning an `int` to a `double` automatically casts the `int` to `double`. For example,

```
int num = 5;
double realNum = num;     //num is cast to double
```

Assigning a `double` to an `int` without a cast, however, causes a compile-time error. For example,

```
double x = 6.79;
int intNum = x;      //Error. Need an explicit cast to int
```

Note that casting a floating-point (real) number to an integer simply truncates the number. For example,

```
double cost = 10.95;
int numDollars = (int) cost;     //sets numDollars to 10
```

If your intent was to round `cost` to the nearest dollar, you needed to write

```
int numDollars = (int) (cost + 0.5);  //numDollars has value 11
```

To round a negative number to the nearest integer:

```
double negAmount = -4.8;
int roundNeg = (int) (negAmount - 0.5);   //roundNeg has value -5
```

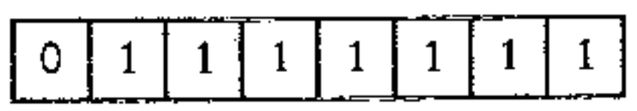
The strategy of adding or subtracting 0.5 before casting correctly rounds in all cases.

Storage of Numbers

INTEGERS

Integer values in Java are stored exactly, as a string of bits (binary digits). One of the bits stores the sign of the integer, 0 for positive, 1 for negative.

The Java built-in integral type, `byte`, uses one byte (eight bits) of storage.



The picture represents the largest positive integer that can be stored using type `byte`: $2^7 - 1$.

Type `int` in Java uses four bytes (32 bits). Taking one bit for a sign, the largest possible integer stored is $2^{31} - 1$. In general, an n-bit integer uses $n/8$ bytes of storage, and stores integers from -2^{n-1} to $2^{n-1} - 1$. (Note that the extra value on the negative side comes from not having to store -0.) There are two Java constants that you should know. `Integer.MAX_VALUE` holds the maximum value an `int` can hold, $2^{31} - 1$. `Integer.MIN_VALUE` holds the minimum value an `int` can hold, -2^{31}.

Built-in types in Java are `byte` (one byte), `short` (two bytes), `int` (four bytes), and `long` (eight bytes). Of these, only `int` is in the AP Java subset.

FLOATING-POINT NUMBERS

There are two built-in types in Java that store real numbers: `float`, which uses four bytes, and `double`, which uses eight bytes. A *floating-point number* is stored in two parts: a *mantissa*, which specifies the digits of the number, and an exponent. The JVM (Java Virtual Machine) represents the number using scientific notation:

$$\text{sign} * \text{mantissa} * 2^{\text{exponent}}$$

In this expression, 2 is the *base* or *radix* of the number. In type `double` eleven bits are allocated for the exponent, and (typically) 52 bits for the mantissa. One bit is allocated for the sign. This is a *double-precision* number. Type `float`, which is *single-precision*, is not in the AP Java subset.

When floating-point numbers are converted to binary, most cannot be represented exactly, leading to *round-off error*. These errors are compounded by arithmetic operations. For example,

$$0.1*26 \neq 0.1+0.1+\cdots+0.1 \quad \text{(26 terms)}$$

In Java, no exceptions are thrown for floating-point operations. There are two situations you should be aware of:

- When an operation is performed that gives an undefined result, Java expresses this result as `NaN`, "not a number." Examples of operations that produce `NaN` are: taking the square root of a negative number, and 0.0 divided by 0.0.
- An operation that gives an infinitely large or infinitely small number, like division by zero, produces a result of `Infinity` or `-Infinity` in Java.

Hexadecimal Numbers

A *hexadecimal number* or *hex number* uses base (radix) 16, and is represented with the symbols 0 – 9 and A – F (occasionally a – f), where A represents 10, and F represents 15. To denote a hex number in Java, the prefix "0x" or "0X" is used, for example, `0xC2A`. On the AP exam, the representation is likely to be with the subscript hex: $C2A_{hex}$. In expanded form, this number means

$$\begin{aligned} &(C)(16^2)+(2)(16^1)+(A)(16^0) \\ &=(12)(16^2)+(2)(16)+(10)(1) \\ &=3114, \text{ or } 3114_{dec} \end{aligned}$$

The advantages of hex numbers are their compactness, and the ease of conversion between hex and binary. Notice that any hex digit expands to four bits. For example,

$$5_{hex} = 0101_{bin} \quad \text{and} \quad F_{hex} = 1111_{bin}$$

Thus, $5F_{hex} = 01011111_{bin}$, which is 1011111_{bin}.

Similarly, to convert a binary number to hex, convert in groups of four from right to left. If necessary, pad with zeroes to complete the last group of four. For example,

$$\begin{aligned} 1011101_{bin} &= 0101 \quad 1101_{bin} \\ &= \ 5 \qquad D_{hex} \\ &= 5D_{hex} \end{aligned}$$

Final Variables

A *final variable* or *user-defined constant*, identified by the keyword `final`, is used to name a quantity whose value will not change. Here are some examples of `final` declarations:

```
final double TAX_RATE = 0.08;
final int CLASS_SIZE = 35;
```

NOTE

1. Constant identifiers are, by convention, capitalized.
2. A `final` variable can be declared without initializing it immediately. For example,

```
final double TAX_RATE;
if (<some condition>)
    TAX_RATE = 0.08;
else
    TAX_RATE = 0.0;
// TAX_RATE can be given a value just once: its value is final!
```

3. A common use for a constant is as an array bound. For example,

```
final int MAXSTUDENTS = 25;
int[] classList = new int[MAXSTUDENTS];
```

4. Using constants makes it easier to revise code. Just a single change in the `final` declaration need be made, rather than having to change every occurrence of a value.

OPERATORS

Arithmetic Operators

Operator	Meaning	Example
+	addition	`3 + x`
-	subtraction	`p - q`
*	multiplication	`6 * i`
/	division	`10 / 4  //returns 2, not 2.5!`
%	mod (remainder)	`11 % 8  //returns 3`

NOTE

1. These operators can be applied to types `int` and `double`, even if both types occur in the same expression. For an operation involving a `double` and an `int`, the `int` is promoted to `double`, and the result is a `double`.
2. The mod operator %, as in the expression `a % b`, gives the remainder when `a` is divided by `b`. Thus `10 % 3` evaluates to `1`, whereas `4.2 % 2.0` evaluates to `0.2`.
3. Integer division `a/b` where both `a` and `b` are of type `int` returns the integer quotient only (i.e., the answer is truncated). Thus, `22/6` gives `3`, and `3/4` gives `0`. If at least one of the operands is of type `double`, then the operation becomes regular floating-point division, and there is no truncation. You can control the kind of division that is carried out by explicitly casting (one or both of) the operands from `int` to `double` and vice versa. Thus

```
3.0 / 4          →  0.75
3 / 4.0          →  0.75
(int) 3.0 / 4    →  0
(double) 3 / 4   →  0.75
```

   You must, however, be careful:

```
(double) (3 / 4)   →  0.0
```

   since the integer division `3/4` is computed first, before casting to `double`.
4. The arithmetic operators follow the normal precedence rules (order of operations):

   (1) parentheses, from the inner ones out (highest precedence)
   (2) *, /, %
   (3) +, - (lowest precedence)

   Here operators on the same line have the same precedence, and, in the absence of parentheses, are invoked from left to right. Thus the expression `19 % 5 * 3 + 14 / 5` evaluates to `4 * 3 + 2 = 14`. Note that casting has precedence over all of these operators. Thus, in the expression `(double) 3/4`, `3` will be cast to `double` before the division is done.

Relational Operators

Operator	Meaning	Example
==	equal to	`if (x == 100)`
!=	not equal to	`if (age != 21)`
>	greater than	`if (salary > 30000)`
<	less than	`if (grade < 65)`
>=	greater than or equal to	`if (age >= 16)`
<=	less than or equal to	`if (height <= 6)`

NOTE

1. Relational operators are used in *boolean expressions* that evaluate to `true` or `false`.

```
boolean x = (a != b);        //initializes x to true if a != b,
                             // false otherwise
```

```
return p == q;                    //returns true if p equals q,
                                  // false otherwise
```

2. If the operands are an `int` and a `double`, the `int` is promoted to a `double` as for arithmetic operators.
3. Relational operators should generally be used only in the comparison of primitive types (i.e., `int`, `double`, or `boolean`). User-defined types are compared using the `equals` and `compareTo` methods (see pp. 136 and 165).
4. Be careful when comparing floating-point values! Since floating-point numbers cannot always be represented exactly in the computer memory, they should not be compared directly using relational operators.

Do not routinely use == to test for equality of floating-point numbers.

Comparing Floating-Point Numbers

Because of round-off errors in floating-point numbers, you can't rely on using the == or != operators to compare two `double` values for equality. They may differ in their last significant digit or two because of round-off error. Instead, you should test that the magnitude of the difference between the numbers is less than some number about the size of the machine precision. The machine precision is usually denoted ϵ and is typically about 10^{-16} for double precision (i.e., about 16 decimal digits). So you would like to test something like $|x - y| \leq \epsilon$. But this is no good if x and y are very large. For example, suppose $x = 1234567890.123456$ and $y = 1234567890.123457$. These numbers are essentially equal to machine precision, since they differ only in the 16th significant digit. But $|x - y| = 10^{-6}$, not 10^{-16}. So in general you should check the *relative* difference:

$$\frac{|x - y|}{\max(|x|, |y|)} \leq \epsilon$$

To avoid problems with dividing by zero, code this as

$$|x - y| \leq \epsilon \max(|x|, |y|)$$

An example of code that uses a correct comparison of real numbers can be found in the Shape class on p. 137.

Logical Operators

Operator	Meaning	Example
!	NOT	`if (!found)`
&&	AND	`if (x < 3 && y > 4)`
||	OR	`if (age < 2 || height < 4)`

NOTE

1. Logical operators are applied to boolean expressions to form *compound boolean expressions* that evaluate to `true` or `false`.
2. Values of `true` or `false` are assigned according to the truth tables for the logical operators.

&&	T	F
T	T	F
F	F	F

||	T	F
T	T	T
F	T	F

!	
T	F
F	T

For example, F `&&` T evaluates to F, while T `||` F evaluates to T.

3. *Short-circuit evaluation.* The subexpressions in a compound boolean expression are evaluated from left to right, and evaluation automatically stops as soon as the value of the entire expression is known. For example, consider a boolean OR expression of the form `A || B`, where `A` and `B` are some boolean expressions. If `A` is `true`, then the expression is `true` irrespective of the value of `B`. Similarly, if `A` is `false`, then `A && B` evaluates to `false` irrespective of the second operand. So in each case the second operand is not evaluated. For example,

```
if (numScores != 0 && scoreTotal/numScores > 90)
```

will not cause a run-time `ArithmeticException` (division-by-zero error) if the value of `numScores` is `0`. This is because `numScores != 0` will evaluate to `false`, causing the entire boolean expression to evaluate to `false` without having to evaluate the second expression containing the division.

Assignment Operators

Operator	Example	Meaning
=	`x = 2`	simple assignment
+=	`x += 4`	`x = x + 4`
-=	`y -= 6`	`y = y - 6`
*=	`p *= 5`	`p = p * 5`
/=	`n /= 10`	`n = n / 10`
%=	`n %= 10`	`n = n % 10`

NOTE

1. All these operators, with the exception of simple assignment, are called *compound assignment operators.*
2. *Chaining* of assignment statements is allowed, with evaluation from right to left.

```
int next, prev, sum;
next = prev = sum = 0;  //initializes sum to 0, then prev to 0
                        //then next to 0
```

Increment and Decrement Operators

Operator	Example	Meaning
++	i++ or ++i	i is incremented by 1
--	k-- or --k	k is decremented by 1

Note that i++ (postfix) and ++i (prefix) both have the net effect of incrementing i by 1, but they are not equivalent. For example, if i currently has the value 5, then System.out.println(i++) will print 5 and then increment i to 6, whereas System.out.println(++i) will first increment i to 6 and then print 6. It's easy to remember: if the ++ is first, you first increment. A similar distinction occurs between k-- and --k. (Note: You do not need to know these distinctions for the AP exam.)

Operator Precedence

highest precedence	→	(1)	!, ++, --
		(2)	*, /, %
		(3)	+, -
		(4)	<, >, <=, >=
		(5)	==, !=
		(6)	&&
		(7)	||
lowest precedence	→	(8)	=, +=, -=, *=, /=, %=

Here operators on the same line have equal precedence. The evaluation of the operators with equal precedence is from left to right, except for rows (1) and (8) where the order is right to left. It is easy to remember: The only "backward" order is for the unary operators (row 1) and for the various assignment operators (row 8).

Example

What will be output by the following statement?

```
System.out.println(5 + 3 < 6 - 1);
```

Since + and - have precedence over <, 5 + 3 and 6 - 1 will be evaluated before evaluating the boolean expression. Since the value of the expression is false, the statement will output false.

INPUT/OUTPUT

Input

Since there are so many ways to provide input to a program, user input is not a part of the AP Java subset. If reading input is a necessary part of a question on the AP exam, it will be indicated something like this:

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

or

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

NOTE

The Scanner class, new in Java 5.0, simplifies both console and file input. It will not, however, be tested on the AP exam.

Output

Testing of output will be restricted to `System.out.print` and `System.out.println`. Formatted output will not be tested.

`System.out` is an object in the `System` class that allows output to be displayed on the screen. The `println` method outputs an item and then goes to a new line. The `print` method outputs an item without going to a new line afterward. An item to be printed can be a string, or a number, or the value of a boolean expression (`true` or `false`). Here are some examples:

```
System.out.print("Hot");
System.out.println("dog");          } prints  Hotdog

System.out.println("Hot");
System.out.println("dog");          } prints  Hot
                                              dog

System.out.println(7 + 3);          } prints  10

System.out.println(7 == 2 + 5);     } prints  true

int x = 27;
System.out.println(x);              } prints  27
System.out.println("Value of x is " + x);
                                    prints  Value of x is 27
```

In the last example, the value of x, 27, is converted to the string `"27"`, which is then concatenated to the string `"Value of x is "`.

To print the "values" of user-defined objects, the `toString()` method is invoked (see p. 164).

Escape Sequences

An *escape sequence* is a backslash followed by a single character. It is used to print special characters. The three escape sequences that you should know for the AP exam are

Escape Sequence	Meaning
\n	newline
\"	double quote
\\	backslash

Here are some examples:

```
System.out.println("Welcome to\na new line");
```

prints

```
Welcome to
a new line
```

The statement

```
System.out.println("He is known as \"Hothead Harry\".");
```

prints

```
He is known as "Hothead Harry".
```

The statement

```
System.out.println("The file path is d:\\myFiles\\..");
```

prints

```
The file path is d:\myFiles\..
```

CONTROL STRUCTURES

Control structures are the mechanism by which you make the statements of a program run in a nonsequential order. There are two general types: decision making and iteration.

Decision-Making Control Structures

These include the `if`, `if...else`, and `switch` statements. They are all selection control structures that introduce a decision-making ability into a program. Based on the truth value of a boolean expression, the computer will decide which path to follow. The `switch` statement is not part of the AP Java subset.

THE `if` STATEMENT

```
if (boolean expression)
{
    statements
}
```

Here the *statements* will be executed only if the *boolean expression* is `true`. If it is `false`, control passes immediately to the first statement following the `if` statement.

THE `if...else` STATEMENT

```
if (boolean expression)
{
    statements
}
else
{
    statements
}
```

Here if the *boolean expression* is `true`, only the *statements* immediately following the test will be executed. If the *boolean expression* is `false`, only the *statements* following the `else` will be executed.

NESTED `if` STATEMENT

If the statement part of an `if` statement is itself an `if` statement, the result is a *nested* `if` *statement.*

Example 1

```
if (boolean expr1)
    if (boolean expr2)
        statement;
```

This is equivalent to

```
if (boolean expr1 && boolean expr2)
    statement;
```

Example 2

Beware the dangling `else`! Suppose you want to read in an integer and print it if it's positive and even. Will the following code do the job?

```
int n = IO.readInt();          //read user input
if (n > 0)
    if (n % 2 == 0)
        System.out.println(n);
else
    System.out.println(n + " is not positive");
```

A user enters 7 and is surprised to see the output

```
7 is not positive
```

The reason is that `else` always gets matched with the *nearest* unpaired `if`, not the first `if` as the indenting would suggest.

There are two ways to fix the preceding code. The first is to use {} delimiters to group the statements correctly.

```
int n = IO.readInt();          //read user input
if (n > 0)
{
    if (n % 2 == 0)
        System.out.println(n);
}
else
    System.out.println(n + " is not positive");
```

The second way of fixing the code is to rearrange the statements.

```
int n = IO.readInt();          //read user input
if (n <= 0)
    System.out.println(n + " is not positive");
else
    if (n % 2 == 0)
        System.out.println(n);
```

EXTENDED if STATEMENT

For example,

```
String grade = IO.readString();          //read user input
if (grade.equals("A"))
    System.out.println("Excellent!");
else if (grade.equals("B"))
    System.out.println("Good");
else if (grade.equals("C") || grade.equals("D"))
    System.out.println("Poor");
else if (grade.equals("F"))
    System.out.println("Egregious!");
else
    System.out.println("Invalid grade");
```

If any of A, B, C, D, or F are entered, an appropriate message will be written, and control will go to the statement immediately following the extended if statement. If any other string is entered, the final else is invoked, and the message Invalid grade will be written.

Iteration

Java has three different control structures that allow the computer to perform iterative tasks: the for loop, while loop, and do...while loop. The do...while loop is not in the AP Java subset.

THE for LOOP

The general form of the for loop is

```
for (initialization; termination condition; update statement)
{
    statements              //body of loop
}
```

The termination condition is tested at the top of the loop; the update statement is performed at the bottom.

Example 1

```
//outputs 1 2 3 4
for (i = 1; i < 5; i++)
    System.out.print(i + " ");
```

Here's how it works. The *loop variable* i is initialized to 1, and the termination condition i < 5 is evaluated. If it is true, the body of the loop is executed and then the loop variable i is incremented according to the update statement. As soon as the termination condition is false (i.e., i >= 5), control passes to the first statement following the loop.

Example 2

```
//outputs 20 19 18 17 16 15
for (k = 20; k >= 15; k--)
    System.out.print(k + " ");
```

Example 3

```
//outputs 2 4 6 8 10
for (j = 2; j <= 10; j += 2)
    System.out.print(j + " ");
```

NOTE

1. The loop variable should not have its value changed inside the loop body.
2. The initializing and update statements can use any valid constants, variables, or expressions.
3. The scope (see p. 92) of the loop variable can be restricted to the loop body by combining the loop variable declaration with the initialization. For example,

```
for (int i = 0; i < 3; i++)
{
    ...
}
```

4. The following loop is syntactically valid:

```
for (int i = 1; i <= 0; i++)
{
    ...
}
```

The loop body will not be executed at all, since the exiting condition is true before the first execution.

THE FOR-EACH LOOP

This is used to iterate over an array or collection. The general form of the loop is

```
for (SomeType element : collection)
{
    statements
}
```

(Read the top line as "For each `element` of type `SomeType` in `collection` ...")

Example

```
//Outputs all elements of arr, one per line.
for (int element : arr)
    System.out.println(element);
```

NOTE

1. The for-each loop cannot be used for replacing or removing elements as you traverse.
2. The loop hides the index variable that is used with arrays.

THE `while` LOOP

The general form of the `while` loop is

```
while (boolean test)
{
    statements          //loop body
}
```

The *boolean test* is performed at the beginning of the loop. If `true`, the loop body is executed. Otherwise, control passes to the first statement following the loop. After execution of the loop body, the test is performed again. If true, the loop is executed again, and so on.

Example 1

```
int i = 1, mult3 = 3;
while (mult3 < 20)
{
    System.out.print(mult3 + " ");
    i++;
    mult3 *= i;
}                          //outputs 3 6 18
```

NOTE

The body of a `while` loop must contain a statement that leads to termination.

1. It is possible for the body of a `while` loop never to be executed. This will happen if the test evaluates to `false` the first time.
2. Disaster will strike in the form of an infinite loop if the test can never be false. Don't forget to change the loop variable in the body of the loop in a way that leads to termination!

Example 2

```
int power2 = 1;
while (power2 != 20)
{
    System.out.println(power2);
    power2 *= 2;
}
```

Since `power2` will never exactly equal 20, the loop will grind merrily along eventually causing an integer overflow.

Example 3

```
/* Screen out bad data.
 * The loop won't allow execution to continue until a valid
 * integer is entered. */
System.out.println("Enter a positive integer from 1 to 100");
int num = IO.readInt();          //read user input
while (num < 1 || num > 100)
{
    System.out.println("Number must be from 1 to 100.");
    System.out.println("Please reenter");
    num = IO.readInt();
}
```

Example 4

```
/* Uses a sentinel to terminate data entered at the keyboard.
 * The sentinel is a value that cannot be part of the data.
 * It signals the end of the list. */
final int SENTINEL = -999;
System.out.println("Enter list of positive integers," +
    " end list with " + SENTINEL);
int value = IO.readInt();        //read user input
while (value != SENTINEL)
{
    process the value
    value = IO.readInt();        //read another value
}
```

NESTED LOOPS

You create a *nested loop* when a loop is a statement in the body of another loop.

Example 1

```
for (int k = 1; k <= 3; k++)
{
    for (int i = 1; i <= 4; i++)
        System.out.print("*");
    System.out.println();
}
```

Think:

```
for each of 3 rows
{
    print 4 stars
    go to next line
}
```

Output:

```
****
****
****
```

Example 2

This example has two loops nested in an outer loop.

```
for (int i = 1; i <= 6; i++)
{
    for (int j = 1; j <= i; j++)
        System.out.print("+");
    for (int j = 1; j <= 6 - i; j++)
        System.out.print("*");
    System.out.println();
}
```

Output:

```
++++++
++++++
++++++
++++++
++++++
++++++
++++++
```

ERRORS AND EXCEPTIONS

An *exception* is an error condition that occurs during the execution of a Java program. For example, if you divide an integer by zero, an `ArithmeticException` will be thrown. If you use a negative array index, an `ArrayIndexOutOfBoundsException` will be thrown.

An *unchecked exception* is one where you don't provide code to deal with the error. Such exceptions are automatically handled by Java's standard exception-handling methods, which terminate execution. You now need to fix your code!

A *checked exception* is one where you provide code to handle the exception, either a `try/catch/finally` statement, or an explicit `throw new ...Exception` clause. These exceptions are not necessarily caused by an error in the code. For example, an unexpected end-of-file could be due to a broken network connection. Checked exceptions are not part of the AP Java subset.

The following exceptions are in the AP Java subset:

Exception	Discussed on page
`ArithmeticException`	this page
`NullPointerException`	94
`ClassCastException`	132
`ArrayIndexOutOfBoundsException`	222
`IndexOutOfBoundsException`	233
`IllegalArgumentException`	next page, 353

See also `NoSuchElementException` (pp. 236, 237) and `IllegalStateException` (pp. 236, 237), which refer to iterators, an optional topic.

Java allows you to write code that throws a standard unchecked exception. Here are typical examples:

Example 1

```
if (numScores == 0)
    throw new ArithmeticException("Cannot divide by zero");
else
    findAverageScore();
```

Example 2

```
public void setRadius(int newRadius)
{
    if (newRadius < 0)
        throw new IllegalArgumentException
              ("Radius cannot be negative");
    else
        radius = newRadius;
}
```

NOTE

1. `throw` and `new` are both reserved words.
2. The error message is optional: The line in Example 1 could have read

   ```
   throw new ArithmeticException();
   ```

   The message, however, is useful, since it tells the person running the program what went wrong.
3. An `IllegalArgumentException` is thrown to indicate that a parameter does not satisfy a method's precondition.

Chapter Summary

Be sure that you understand the difference between primitive and user-defined types, and between the following types of operators: arithmetic, relational, logical, and assignment. Know which conditions lead to what types of errors.

You should be able to work with numbers—know how to compare them, and how to convert between decimal, binary, and hexadecimal numbers. Know how integers and floating-point numbers are stored in memory, and be aware of the conditions that can lead to round-off error.

You should know the `Integer` constants `Integer.MIN_VALUE` and `Integer.MAX_VALUE`.

Be familiar with each of the following control structures: conditional statements, `for` loops, `while` loops, and for-each loops.

Be aware of the AP exam expectations concerning input and output.

MULTIPLE-CHOICE QUESTIONS ON INTRODUCTORY JAVA LANGUAGE CONCEPTS

1. Which of the following pairs of declarations will cause an error message?

I
```
double x = 14.7;
int y = x;
```

II
```
double x = 14.7;
int y = (int) x;
```

III
```
int x = 14;
double y = x;
```

(A) None
(B) I only
(C) II only
(D) III only
(E) I and III only

2. What output will be produced by

```
System.out.print("\\* This is not\n a comment *\\");
```

(A) `* This is not a comment *`

(B) `\* This is not a comment *\`

(C)
```
* This is not
a comment *
```

(D)
```
\\* This is not
a comment *\\
```

(E)
```
\* This is not
a comment *\
```

3. Refer to the following code fragment:

```
double answer = 13 / 5;
System.out.println("13 / 5 = " + answer);
```

The output is

```
13 / 5 = 2.0
```

The programmer intends the output to be

```
13 / 5 = 2.6
```

Which of the following replacements for the first line of code will *not* fix the problem?

(A) `double answer = (double) 13 / 5;`
(B) `double answer = 13 / (double) 5;`
(C) `double answer = 13.0 / 5;`
(D) `double answer = 13 / 5.0;`
(E) `double answer = (double) (13 / 5);`

4. What value is stored in `result` if

```
int result = 13 - 3 * 6 / 4 % 3;
```

(A) −5
(B) 0
(C) 13
(D) −1
(E) 12

5. Suppose that addition and subtraction had higher precedence than multiplication and division. Then the expression

```
2 + 3 * 12 / 7 - 4 + 8
```

would evaluate to which of the following?

(A) 11
(B) 12
(C) 5
(D) 9
(E) −4

6. Let `x` be a variable of type `double` that is positive. A program contains the boolean expression `(Math.pow(x,0.5) == Math.sqrt(x))`. Even though $x^{1/2}$ is mathematically equivalent to $\sqrt{x}$, the above expression returns the value `false` in a student's program. Which of the following is the most likely reason?

(A) `Math.pow` returns an `int`, while `Math.sqrt` returns a `double`.
(B) `x` was imprecisely calculated in a previous program statement.
(C) The computer stores floating-point numbers with 32-bit words.
(D) There is round-off error in calculating the `pow` and `sqrt` functions.
(E) There is overflow error in calculating the `pow` function.

7. Consider the following code segment

```
if (n != 0 && x / n > 100)
    statement1;
else
    statement2;
```

If n is of type int and has a value of 0 when the segment is executed, what will happen?

(A) An ArithmeticException will be thrown.
(B) A syntax error will occur.
(C) *statement1*, but not *statement2*, will be executed.
(D) *statement2*, but not *statement1*, will be executed.
(E) Neither *statement1* nor *statement2* will be executed; control will pass to the first statement following the if statement.

8. What will the output be for the following poorly formatted program segment, if the input value for num is 22?

```
int num =  call to a method that reads an integer;
if (num > 0)
if (num % 5 == 0)
System.out.println(num);
else System.out.println(num + " is negative");
```

(A) 22
(B) 4
(C) 2 is negative
(D) 22 is negative
(E) Nothing will be output.

9. What values are stored in x and y after execution of the following program segment?

```
int x = 30, y = 40;
if (x >= 0)
{
    if (x <= 100)
    {
        y = x * 3;
        if (y < 50)
            x /= 10;
    }
    else
        y = x * 2;
}
else
    y = -x;
```

(A) x = 30 y = 90
(B) x = 30 y = -30
(C) x = 30 y = 60
(D) x = 3 y = -3
(E) x = 30 y = 40

10. Which of the following will evaluate to true only if boolean expressions `A`, `B`, and `C` are all false?
    (A) `!A && !(B && !C)`
    (B) `!A || !B || !C`
    (C) `!(A || B || C)`
    (D) `!(A && B && C)`
    (E) `!A || !(B || !C)`

11. Assume that `a` and `b` are integers. The boolean expression

```
!(a <= b) && (a * b > 0)
```

    will always evaluate to `true` given that
    (A) `a = b`
    (B) `a > b`
    (C) `a < b`
    (D) `a > b` and `b > 0`
    (E) `a > b` and `b < 0`

12. Given that `a`, `b`, and `c` are integers, consider the boolean expression

```
(a < b) || !((c == a * b) && (c < a))
```

    Which of the following will *guarantee* that the expression is `true`?
    (A) `c < a` is `false`.
    (B) `c < a` is `true`.
    (C) `a < b` is `false`.
    (D) `c == a * b` is `true`.
    (E) `c == a * b` is `true`, and `c < a` is `true`.

13. Given that `n` and `count` are both of type `int`, which statement is true about the following code segments?

```
I for (count = 1; count <= n; count++)
      System.out.println(count);
```

```
II count = 1;
   while (count <= n)
   {
       System.out.println(count);
       count++;
   }
```

    (A) I and II are exactly equivalent for all input values `n`.
    (B) I and II are exactly equivalent for all input values $n \geq 1$, but differ when $n \leq 0$.
    (C) I and II are exactly equivalent only when $n = 0$.
    (D) I and II are exactly equivalent only when `n` is even.
    (E) I and II are not equivalent for any input values of `n`.

14. The following fragment intends that a user will enter a list of positive integers at the keyboard and terminate the list with a sentinel:

```
int value = 0;
final int SENTINEL = -999;
while (value != SENTINEL)
{
    //code to process value
        ...
    value = IO.readInt();        //read user input
}
```

The fragment is not correct. Which is a true statement?

(A) The sentinel gets processed.
(B) The last nonsentinel value entered in the list fails to get processed.
(C) A poor choice of `SENTINEL` value causes the loop to terminate before all values have been processed.
(D) The code will always process a value that is not on the list.
(E) Entering the `SENTINEL` value as the first value causes a run-time error.

15. Suppose that base-2 (binary) numbers and base-16 (hexadecimal) numbers can be denoted with subscripts, as shown below:

$$2A_{hex} = 101010_{bin}$$

Which is equal to $3D_{hex}$?

(A) 111101_{bin}
(B) 101111_{bin}
(C) 10011_{bin}
(D) 110100_{bin}
(E) 101101_{bin}

16. A common use of hexadecimal numerals is to specify colors on web pages. Every color has a red, green, and blue component. In decimal notation, these are denoted with an ordered triple (x, y, z), where x, y, and z are the three components, each an `int` from 0 to 255. For example, a certain shade of red, whose red, green, and blue components are 238, 9, and 63, is represented as (238, 9, 63).

In hexadecimal, a color is represented in the format #RRGGBB, where RR, GG, and BB are hex values for the red, green, and blue. Using this notation, the color (238, 9, 63) would be coded as #EE093F.

Which of the following hex codes represents the color (14, 20, 255)?

(A) #1418FE
(B) #0E20FE
(C) #0E14FF
(D) #0FE5FE
(E) #0D14FF

17. In Java, a variable of type `int` is represented internally as a 32-bit signed integer. Suppose that one bit stores the sign, and the other 31 bits store the magnitude of the number in base 2. In this scheme, what is the largest value that can be stored as type `int`?
    (A) 2^{32}
    (B) $2^{32} - 1$
    (C) 2^{31}
    (D) $2^{31} - 1$
    (E) 2^{30}

18. Consider this code segment:

```
int x = 10, y = 0;
while (x > 5)
{
    y = 3;
    while (y < x)
    {
        y *= 2;
        if (y % x == 1)
            y += x;
    }
    x -= 3;
}
System.out.println(x + " " + y);
```

What will be output after execution of this code segment?
(A) `1  6`
(B) `7  12`
(C) `-3  12`
(D) `4  12`
(E) `-3  6`

Questions 19 and 20 refer to the following method, `checkNumber`, which checks the validity of its four-digit integer parameter.

```
//Precondition:  n is a 4-digit integer.
//Postcondition: Returns true if n is valid, false otherwise.
boolean checkNumber(int n)
{
    int d1,d2,d3,checkDigit,nRemaining,rem;
    //strip off digits
    checkDigit = n % 10;
    nRemaining = n / 10;
    d3 = nRemaining % 10;
    nRemaining /= 10;
    d2 = nRemaining % 10;
    nRemaining /= 10;
    d1 = nRemaining % 10;
    //check validity
    rem = (d1 + d2 + d3) % 7;
    return rem == checkDigit;
}
```

A program invokes method `checkNumber` with the statement

```
boolean valid = checkNumber(num);
```

19. Which of the following values of `num` will result in `valid` having a value of `true`?
    (A) `6143`
    (B) `6144`
    (C) `6145`
    (D) `6146`
    (E) `6147`

20. What is the purpose of the local variable `nRemaining`?
    (A) It is not possible to separate `n` into digits without the help of a temporary variable.
    (B) `nRemaining` prevents the parameter `num` from being altered.
    (C) `nRemaining` enhances the readability of the algorithm.
    (D) On exiting the method, the value of `nRemaining` may be reused.
    (E) `nRemaining` is needed as the left-hand side operand for integer division.

21. What output will be produced by this code segment? (Ignore spacing.)

```
for (int i = 5; i >= 1; i--)
{
    for (int j = i; j >= 1; j--)
        System.out.print(2 * j - 1);
    System.out.println();
}
```

(A)
```
9  7  5  3  1
9  7  5  3
9  7  5
9  7
9
```

(B)
```
9  7  5  3  1
7  5  3  1
5  3  1
3  1
1
```

(C)
```
9  7  5  3  1
7  5  3  1 -1
5  3  1 -1 -3
3  1 -1 -3 -5
1 -1 -3 -5 -7
```

(D)
```
1
1  3
1  3  5
1  3  5  7
1  3  5  7  9
```

(E)
```
1  3  5  7  9
1  3  5  7
1  3  5
1  3
1
```

22. Which of the following program fragments will produce this output? (Ignore spacing.)

```
2 - - - - -
- 4 - - - -
- - 6 - - -
- - - 8 - -
- - - - 10 -
- - - - - 12
```

I
```
for (int i = 1; i <= 6; i++)
  {
      for (int k = 1; k <= 6; k++)
          if (k == i)
              System.out.print(2 * k);
          else
              System.out.print("-");
      System.out.println();
  }
```

II
```
for (int i = 1; i <= 6; i++)
  {
      for (int k = 1; k <= i - 1; k++)
          System.out.print("-");
      System.out.print(2 * i);
      for (int k = 1; k <= 6 - i; k++)
          System.out.print("-");
      System.out.println();
  }
```

III
```
for (int i = 1; i <= 6; i++)
  {
      for (int k = 1; k <= i - 1; k++)
          System.out.print("-");
      System.out.print(2 * i);
      for (int k = i + 1; k <= 6; k++)
          System.out.print("-");
      System.out.println();
  }
```

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

23. Consider this program segment:

```
int newNum = 0, temp;
int num = k;              //k is some predefined integer value ≥ 0
while (num > 10)
{
    temp = num % 10;
    num /= 10;
    newNum = newNum * 10 + temp;
}
System.out.print(newNum);
```

Which is a true statement about the segment?

I If `100 ≤ num ≤ 1000` initially, the final value of `newNum` must be in the range `10 ≤ newNum ≤ 100`.
II There is no initial value of `num` that will cause an infinite `while` loop.
III If `num ≤ 10` initially, `newNum` will have a final value of `0`.

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

24. Consider the method reverse:

```
//Precondition:  n > 0.
//Postcondition: returns n with its digits reversed.
//Example: If n = 234, method reverse returns 432.
int reverse(int n)
{
    int rem, revNum = 0;

    /* code segment */

    return revNum;
}
```

Which of the following replacements for /* *code segment* */ would cause the method to work as intended?

I
```
for (int i = 0; i <= n; i++)
{
    rem = n % 10;
    revNum = revNum * 10 + rem;
    n /= 10;
}
```

II
```
while (n != 0)
{
    rem = n % 10;
    revNum = revNum * 10 + rem;
    n /= 10;
}
```

III
```
for (int i = n; i != 0; i /= 10)
{
    rem = i % 10;
    revNum = revNum * 10 + rem;
}
```

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