2

Introductory Java Language Features

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

Learning Objectives

In this chapter, you will learn:

- → Packages and classes
- → Types and identifiers
- → Operators
- → Input/output
- → 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. A complete listing of this subset can be found on the College Board website.

Java provides basic control structures such as the `if-else` statement, `for` loop, enhanced `for` 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. For example, the package `java.util` contains the collections classes. Note that 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.*;
```

Note that the package name is all lowercase letters. 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;
```

For example,

```
import java.util.ArrayList;
```

The `import` statement allows the programmer to use the objects and methods defined in the designated package. 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.
2. The words `class`, `public`, `static`, and `void` are *reserved words,* also called *keywords.* (This means they have specific uses in Java and may not be used as identifiers.)
3. The keyword `public` signals that the class or method is usable outside of the class, whereas `private` data members or methods (see Chapter 3) are not.
4. 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 above. This is typically true for all methods in a source file that contains no *instance variables* (see Chapter 3). Most methods in Java do operate on objects and are not static. The `main` method, however, must always be static.
5. The program shown above is a Java *application.*

6. There are three different types of comment delimiters in Java:
   - `/*...*/`, which is the one used in the program shown, to enclose a block of comments. The block can extend over one or more lines.
   - `//`, which generates a comment on one line.
   - `/**...*/`, which generates Javadoc comments. These are used to create API documentation of Java library software.

Javadoc Comments

The Javadoc comments `@param` and `@return` are no longer part of the AP Java 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.

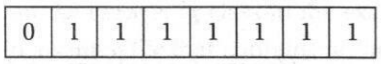
Storage of Numbers

The details of storage are not tested on the AP exam. They are, however, useful for understanding the differences between types `int` and `double`.

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 integer 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`, 11 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,

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

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 and Octal Numbers

Base 2, base 8, and base 16 are no longer part of the AP Java subset. Only base 10 will be used on the AP exam.

Final Variables

A *final variable* or *user-defined constant*, identified by the keyword `final`, is 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`). Strings are compared using the `equals` and `compareTo` methods (see p. 175).
4. Be careful when comparing floating-point values! Since floating-point numbers cannot always be represented exactly in the computer memory, a round-off error could be introduced, leading to an incorrect result when == is used to test for equality.

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

OPTIONAL TOPIC

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

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

Logical Operators

A *logical operator* (sometimes called a *boolean operator*) is one that returns a `boolean` result that is based on the `boolean` result(s) of one or two other `boolean` expressions. The three logical operators are shown in the table below.

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. (This is not tested on the AP exam.)

```
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);
```

Solution

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 = ...;     //read user input
```

Note

The `Scanner` class 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);
System.out.println("Value of x is " + x);
```
prints `27`

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. 171).

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 in 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 = ...;          //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 = ...;          //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 = ...;          //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 = ...;          //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. 106) 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.

Enhanced `for` loop (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 enhanced `for` loop should be used for accessing elements in the data structure, not 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
```

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

Note

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!
3. When the loop finishes executing, the value of the loop variable `mult3` is 72, causing the test `mult3 < 20` to fail. Notice that `i` is also updated and its final value is 4.

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 = ...;        //read user input
while (num < 1 || num > 100)
{
    System.out.println("Number must be from 1 to 100.");
    System.out.println("Please reenter");
    num = ...;
}
```

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 = ...;        //read user input
while (value != SENTINEL)
{
    process the value
    value = ...;        //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();
}
```

The table below shows what the algorithm is doing.

k	i	Steps
1	1 2 3 4	When k is 1, print 4 stars, go to next line
2	1 2 3 4	When k is 2, do it again
3	1 2 3 4	And again, when k is 3

Think:

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

Here is the final 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 that is automatically handled by Java's standard exception-handling methods, which terminate execution. It is thrown if an attempt is made to divide an integer by 0, or if an array index goes out of bounds, and so on. The exception tells you that 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 unchecked exceptions are in the AP Java subset:

Exception	Discussed on page
`ArithmeticException`	on the previous page
`NullPointerException`	110
`ArrayIndexOutOfBoundsException`	239
`IndexOutOfBoundsException`	250
`StringIndexOutOfBoundsException`	176
`ConcurrentModificationException`	254

Java allows you to write code that throws a standard unchecked exception. Here is a typical example.

Example

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

Note

1. `throw` and `new` are both reserved words. (The keywords `throw` and `throws` are not in the AP Java subset.)
2. The error message is optional: The line in this example could have read

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

   The message, however, is useful, since it tells the person running the program what went wrong.

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 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 enhanced `for` loops.

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

Learn the unchecked exceptions that are part of the AP Java subset.