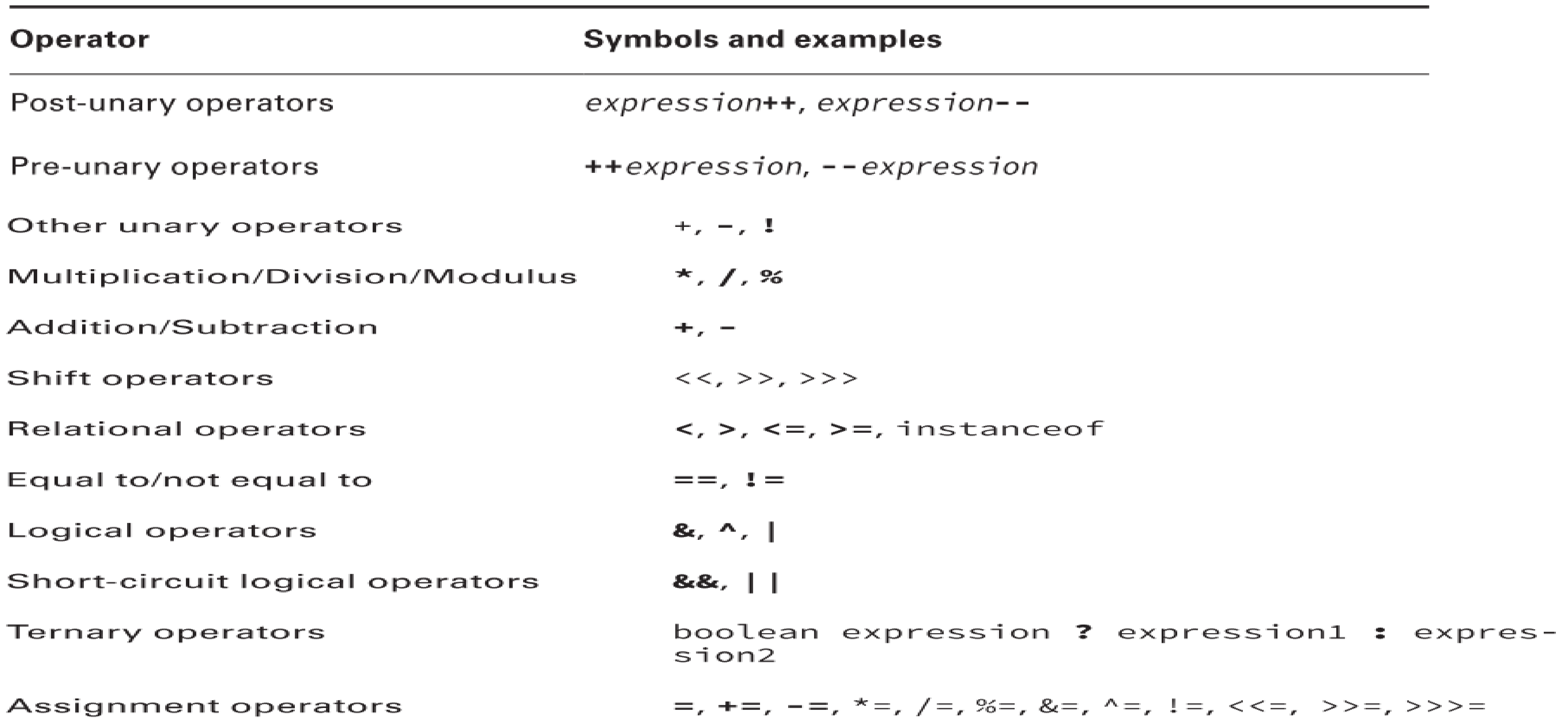
Chapter 2:Operators and Statements

• Chapter 2: Operators and Statements

Understanding Java Operators

Unless overridden with parentheses, Java operators follow order of operation, listed in the next Table, by decreasing order of operator precedence. If two operators have the same level of precedence, then Java guarantees left-to-right evaluation. You need to know only those operators in bold for the OCA exam.



Operator	Symbols and examples
Post-unary operators	*expression*++, *expression*--
Pre-unary operators	++*expression*, --*expression*
Other unary operators	+, -, !
Multiplication/Division/Modulus	*, /, %
Addition/Subtraction	+, -
Shift operators	<<, >>, >>>
Relational operators	<, >, <=, >=, instanceof
Equal to/not equal to	==, !=
Logical operators	&, ^, |
Short-circuit logical operators	&&, ||
Ternary operators	boolean expression ? expression1 : expression2
Assignment operators	=, +=, -=, *=, /=, %=, &=, ^=, !=, <<=, >>=, >>>=

• Chapter 2: Operators and Statements

Working with Binary Arithmetic Operators

- **Arithmetic Operators**

Arithmetic operators are often encountered in early mathematics and include addition (+), subtraction (-), multiplication (*), division (/), and modulus (%). They also include the unary operators, ++ and --.
Exp : int x = 2 * 5 + 3 * 4 - 8;

- **Numeric Promotion**

Now that you understand the basics of arithmetic operators, it is vital we talk about primitive numeric promotion, as Java may do things that seem unusual to you at first. If you recall in Chapter 1, “Java Building Blocks,” where we listed the primitive numeric types, each primitive has a bit-length. You don’t need to know the exact size of these types for the exam, but you should know which are bigger than others.

Chapter 2: Operators and Statements

Working with Binary Arithmetic Operators

- **Numeric Promotion (cont.)**

  You should memorize certain rules Java will follow when applying operators to data types:

Numeric Promotion Rules

1. If two values have different data types, Java will automatically promote one of the values to the larger of the two data types.

2. If one of the values is integral and the other is floating-point, Java will automatically promote the integral value to the floating-point value's data type.

3. Smaller data types, namely byte, short, and char, are first promoted to int any time they're used with a Java binary arithmetic operator, even if neither of the operands is int.

4. After all promotion has occurred and the operands have the same data type, the resulting value will have the same data type as its promoted operands.

• Chapter 2: Operators and Statements

Working with Binary Arithmetic Operators

- **Numeric Promotion (suite)**

What is the data type of x * y?

```
int x = 1;
long y = 33;
```

What is the data type of x + y?

```
double x = 39.21;
float y = 2.1;
```

What is the data type of x / y?

```
short x = 10;
short y = 3;
```

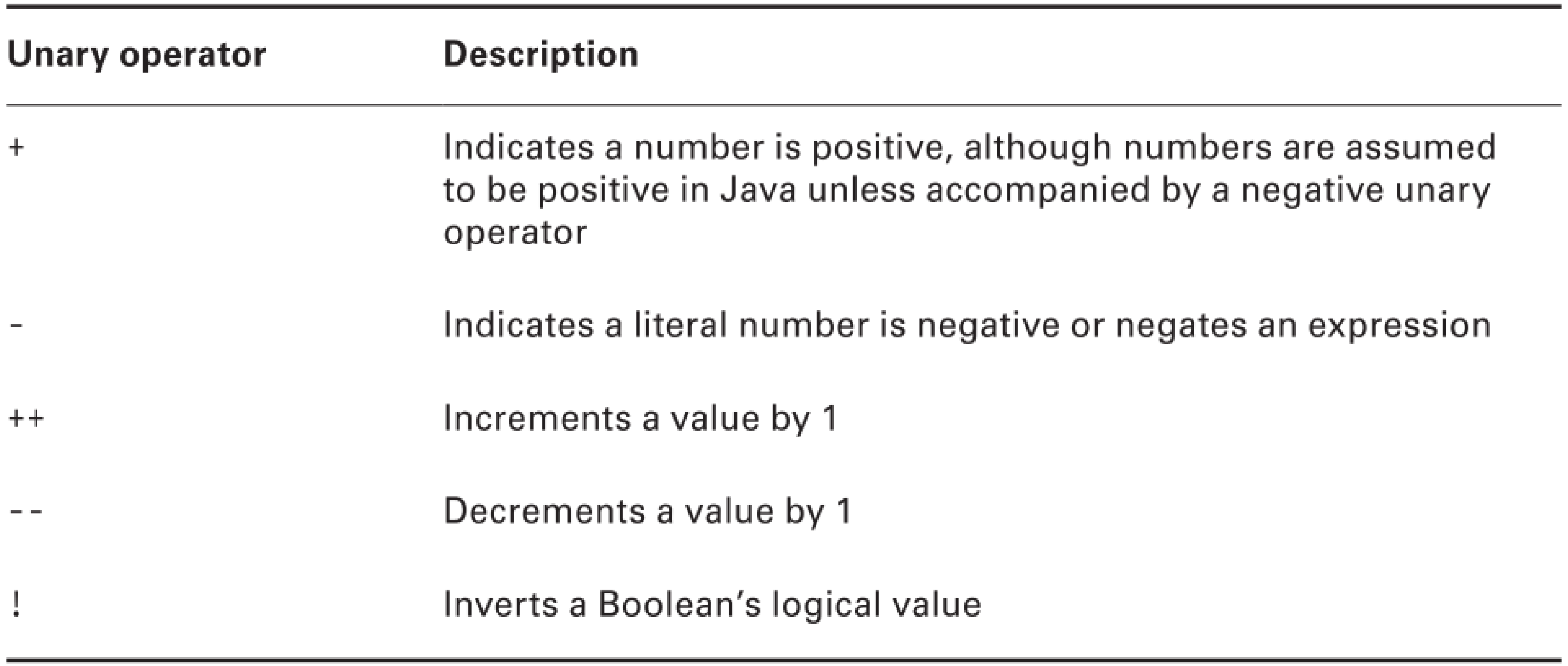
What is the data type of x * y / z?

```
short x = 14;
float y = 13;
double z = 30;
```

• Chapter 2: Operators and Statements

Working with Unary Operators

By definition, a *unary* operator is one that requires exactly one operand, or variable, to function. As shown in The Table , they often perform simple tasks, such as increasing a numeric variable by one, or negating a boolean value.



Unary operator	Description
+	Indicates a number is positive, although numbers are assumed to be positive in Java unless accompanied by a negative unary operator
-	Indicates a literal number is negative or negates an expression
++	Increments a value by 1
--	Decrements a value by 1
!	Inverts a Boolean's logical value

• Chapter 2: Operators and Statements

Working with Unary Operators

- **Logical Complement and Negation Operators :**

The *logical complement operator,* !, flips the value of a boolean expression.

Likewise, the *negation operator*, -, reverses the sign of a numeric expression.

```
boolean x = false;
System.out.println(x);  // false
x = !x;
System.out.println(x);  // true
```

```
double x = 1.21;
System.out.println(x);  // 1.21
x = -x;
System.out.println(x);  // -1.21
x = -x;
System.out.println(x);  // 1.21
```

```
int x = !5;  // DOES NOT COMPILE
boolean y = -true;  // DOES NOT COMPILE
boolean z = !0;  // DOES NOT COMPILE
```

Chapter 2: Operators and Statements

Working with Unary Operators

- **Increment and Decrement Operators :**

- Increment and decrement operators, ++ and --, respectively, can be applied to numeric operands and have the higher order or precedence, as compared to binary operators. In other words, they often get applied first to an expression.

- Increment and decrement operators require special care because the order they are applied to their associated operand can make a difference in how an expression is processed.

```
int counter = 0;
System.out.println(counter);  // Outputs 0
System.out.println(++counter);  // Outputs 1
System.out.println(counter); // Outputs 1
System.out.println(counter--);  // Outputs 1
System.out.println(counter);  // Outputs 0
```

```
int x = 3;
int y = ++x * 5 / x-- + --x;
System.out.println("x is " + x);
System.out.println("y is " + y);
```

• Chapter 2: Operators and Statements

Using Additional Binary Operators

- **Assignment Operators :**
  An assignment operator is a binary operator that modifies, or assigns, the variable on the left-hand side of the operator, with the result of the value on the right-hand side of the equation. The simplest assignment operator is the = assignment, which you have seen already:
  - int x = 1;

```
int x = 1.0;   // DOES NOT COMPILE
short y = 1921222;   // DOES NOT COMPILE
int z = 9f;   // DOES NOT COMPILE
long t = 192301398193810323;   // DOES NOT COMPILE
```

- **Casting Primitive Values**

```
short x = 10;
short y = 3;
short z = x * y;
```

```
short x = 10;
short y = 3;
short z = (short)(x * y);
```

• Chapter 2: Operators and Statements

Using Additional Binary Operators

- **Compound Assignment Operators:**
  Besides the simple assignment operator, =, there are also numerous compound assignment operators. Only two of the compound operators are required for the exam, += and -=.

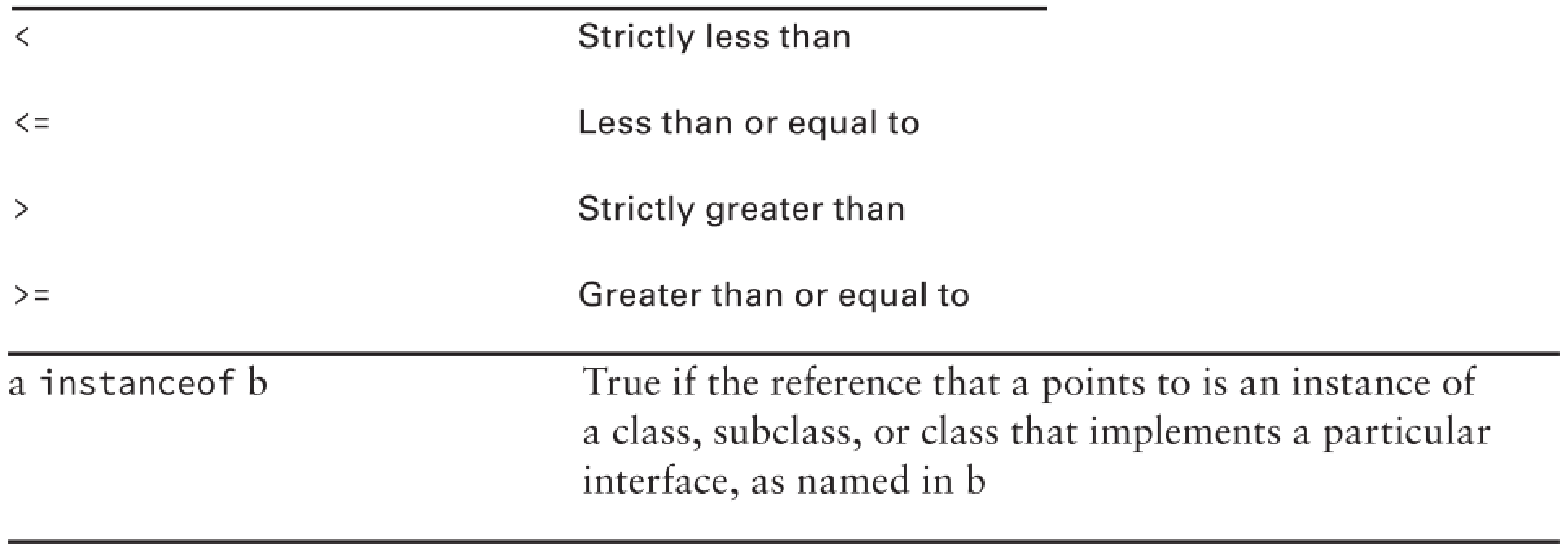
Compound operators are useful for more than just shorthand—they can also save us from having to explicitly cast a value.

```
long x = 10;
int y = 5;
y = y * x;   // DOES NOT COMPILE
```

```
long x = 10;
int y = 5;
y *= x;
```

• Chapter 2: Operators and Statements

Relational Operators



`<`	Strictly less than
`<=`	Less than or equal to
`>`	Strictly greater than
`>=`	Greater than or equal to
`a instanceof b`	True if the reference that a points to is an instance of a class, subclass, or class that implements a particular interface, as named in b

• Chapter 2: Operators and Statements

Logical Operators

x & y
(AND)

	y = true	y = false
x = true	true	false
x = false	false	false

x | y
(INCLUSIVE OR)

	y = true	y = false
x = true	true	true
x = false	true	false

x ^ y
(EXCLUSIVE OR)

	y = true	y = false
x = true	false	true
x = false	true	false

Here are some tips to help remember this table:
- ■ AND is only true if both operands are true.
- ■ Inclusive OR is only false if both operands are false.
- ■ Exclusive OR is only true if the operands are different.

Finally, we present the conditional operators, && and ||, which are often referred to as short-circuit operators. The *short-circuit operators* are nearly identical to the logical operators, & and |, respectively, except that the right-hand side of the expression may never be evaluated if the final result can be determined by the left-hand side of the expression.

```
boolean x = true || (y < 4);
```

```
int x = 6;
boolean y = (x >= 6) || (++x <= 7);
System.out.println(x);
```

Chapter 2: Operators and Statements

Equality Operators

Let’s start with the basics, the *equals* operator == and *not equals* operator !=. Like the relational operators, they compare two operands and return a boolean value about whether the expressions or values are equal, or not equal, respectively.

The equality operators are used in one of three scenarios:

1. Comparing two numeric primitive types. If the numeric values are of different data types, the values are automatically promoted as previously described. For example, 5 == 5.00 returns true since the left side is promoted to a double.
2. Comparing two boolean values.
3. Comparing two objects, including null and String values.

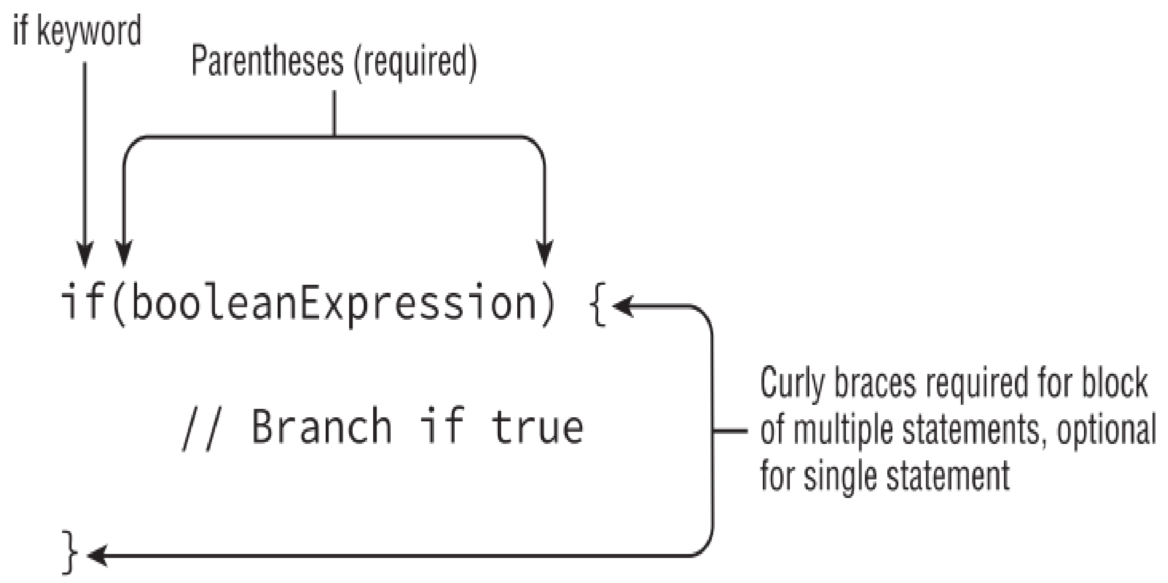
```
boolean x = true == 3;  // DOES NOT COMPILE
boolean y = false != "Giraffe";  // DOES NOT COMPILE
boolean z = 3 == "Kangaroo";  // DOES NOT COMPILE
```

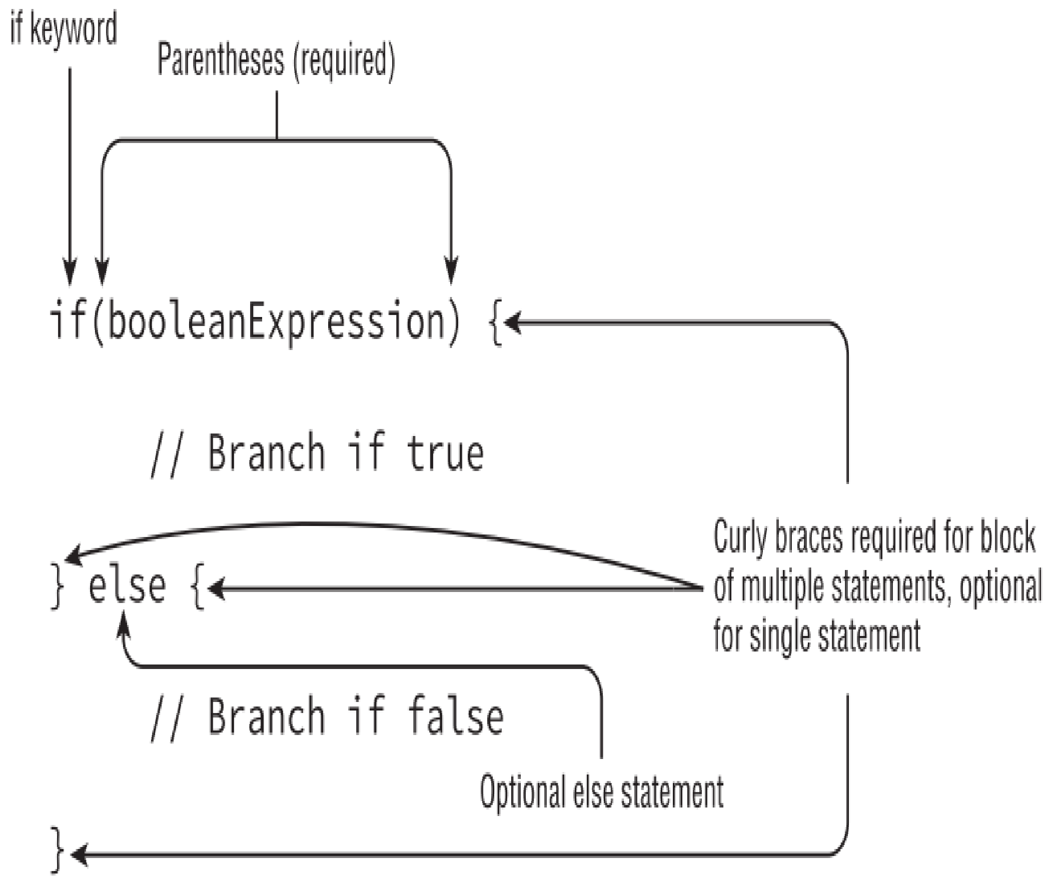
```
boolean y = false;
boolean x = (y = true);
System.out.println(x);  // Outputs true
```

```
File x = new File("myFile.txt");
File y = new File("myFile.txt");
File z = x;
System.out.println(x == y);  // Outputs false
System.out.println(x == z);  // Outputs true
```

- # Chapter 2: Operators and Statements

Understanding Java Statements

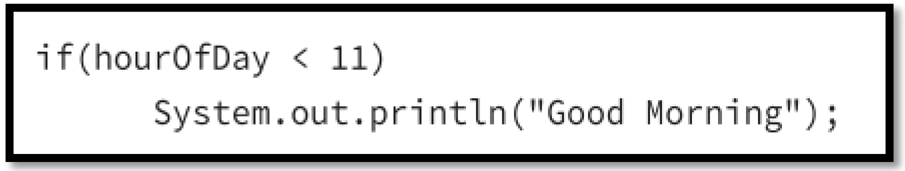

if keyword
Parentheses (required)
if(booleanExpression) {
// Branch if true
}
Curly braces required for block of multiple statements, optional for single statement



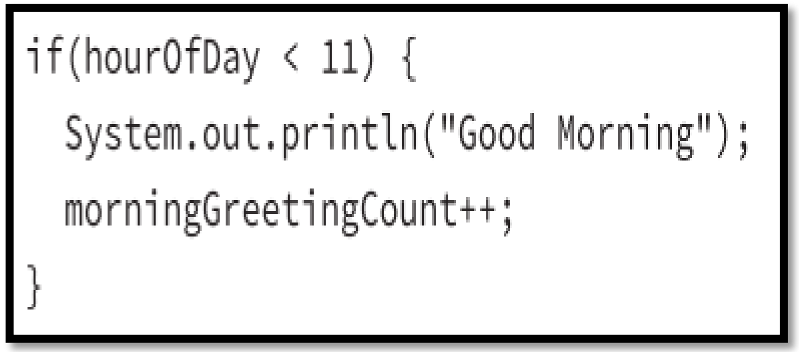
• Chapter 2: Operators and Statements

Understanding Java Statements

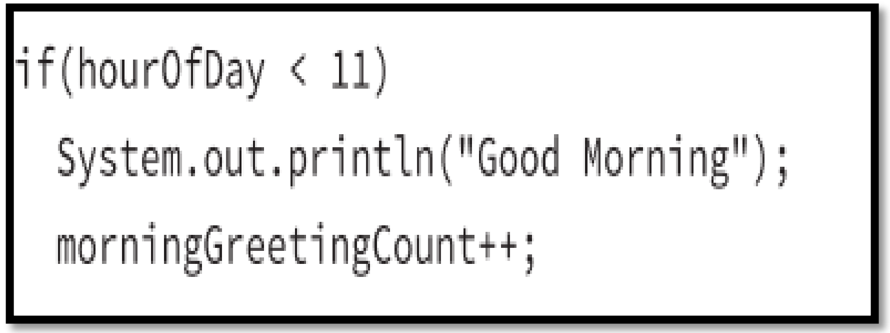
Exp : if-then Statement



```
if(hourOfDay < 11)
      System.out.println("Good Morning");
```



```
if(hourOfDay < 11) {
  System.out.println("Good Morning");
  morningGreetingCount++;
}
```



```
if(hourOfDay < 11)
  System.out.println("Good Morning");
  morningGreetingCount++;
```

• Chapter 2: Operators and Statements

Understanding Java Statements

Exp : if-then-else Statement

```
if(hourOfDay < 11) {
  System.out.println("Good Morning");
}
if(hourOfDay >= 11) {
  System.out.println("Good Afternoon");
}
```

```
if(hourOfDay < 11) {
  System.out.println("Good Morning");
} else {
  System.out.println("Good Afternoon");
}
```

```
if(hourOfDay < 11) {
  System.out.println("Good Morning");
} else if(hourOfDay < 15) {
  System.out.println("Good Afternoon");
} else {
  System.out.println("Good Evening");
}
```

```
if(hourOfDay < 15) {
  System.out.println("Good Afternoon");
} else if(hourOfDay < 11) {
  System.out.println("Good Morning");
} else {
  System.out.println("Good Evening");
}
```

```
// UNREACHABLE CODE
```

• Chapter 2: Operators and Statements

Understanding Java Statements

```
int x = 1;
if(x = 5) {  // DOES NOT COMPILE
  ...
}
```

```
int x = 1;
if(x) {  // DOES NOT COMPILE
  ...
}
```

Ternary Operator

Now that we have discussed if-then-else statements, we can briefly return to our discussion of operators and present the fi nal operator that you need to learn for the exam. The conditional operator, ? :, otherwise known as the ternary operator, is the only operator that takes three operands and is of the form:

```
booleanExpression ? expression1 : expression2
```

The first operand must be a boolean expression, and the second and third can be anyexpression that returns a value. The ternary operation is really a condensed form of an if-then-else statement that returns a value. For example, the following two snippets of code are equivalent:

• Chapter 2: Operators and Statements

Understanding Java Statements

Ternary Operator

```
int y = 10;
final int x;
if(y > 5) {
   x = 2 * y;
} else {
   x = 3 * y;
}
```

```
int y = 10;
int x = (y > 5) ? (2 * y) : (3 * y);
```

Note that it is often helpful for readability to add parentheses around the expressions in ternary operations, although it is certainly not required.
There is no requirement that second and third expressions in ternary operations have the same data types, although it may come into play when combined with the assignment operator. Compare the following two statements:

```
System.out.println((y > 5) ? 21 : "Zebra");
int animal = (y < 91) ? 9 : "Horse";  // DOES NOT COMPILE
```

• Chapter 2: Operators and Statements

Understanding Java Statements

Ternary Operator

As of Java 7, only one of the right-hand expressions of the ternary operator will be evaluated at runtime. In a manner similar to the short-circuit operators, if one of the two right hand expressions in a ternary operator performs a side effect, then it may not be applied at runtime.
Let’s illustrate this principle with the following example:

```
int y = 1;
int z = 1;
final int x = y<10 ? y++ : z++;
System.out.println(y+","+z); // Outputs 2,1
```

```
int y = 1;
int z = 1;
final int x = y>=10 ? y++ : z++;
System.out.println(y+","+z); // Outputs 1,2
```

For the exam, be wary of any question that includes a ternary expression in which a variable is modified in one of the right-hand side expressions.

• Chapter 2: Operators and Statements

Understanding Java Statements

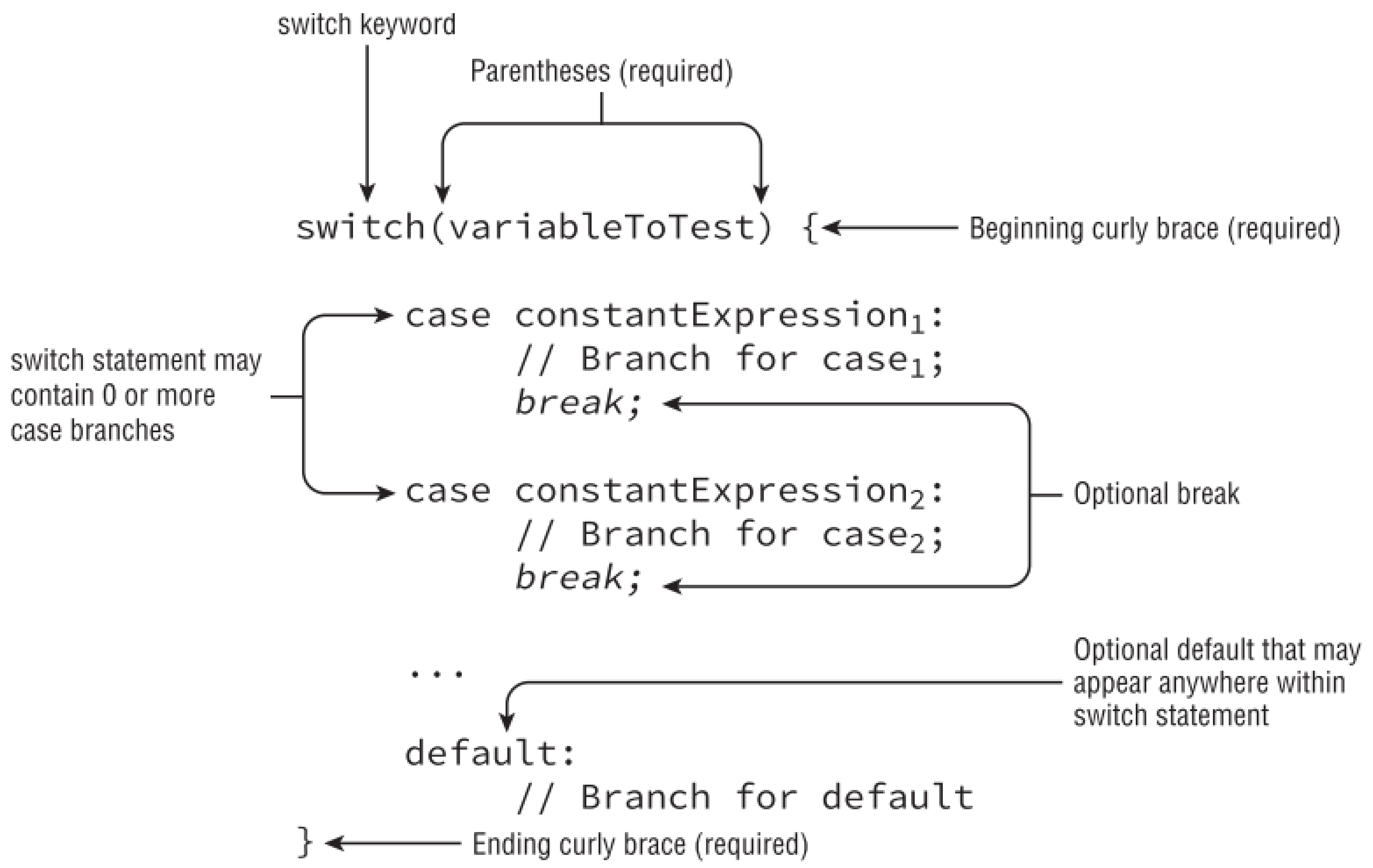
The switch Statement

A *switch* statement is a complex decision-making structure in which a single value is evaluated and flow is redirected to the first matching branch, known as a *case* statement. If no such case statement is found that matches the value, an optional *default* statement will be called. If no such default option is available, the entire switch statement will be skipped.

• Chapter 2: Operators and Statements

Understanding Java Statements

The switch Statement



```
switch(variableToTest) {
    case constantExpression1:
        // Branch for case1;
        break;

    case constantExpression2:
        // Branch for case2;
        break;

    ...

    default:
        // Branch for default
}
```

switch keyword
Parentheses (required)
Beginning curly brace (required)
switch statement may contain 0 or more case branches
Optional break
Optional default that may appear anywhere within switch statement
Ending curly brace (required)

• Chapter 2: Operators and Statements

Understanding Java Statements

The switch Statement

Data types supported by switch statements include the following:

■ int and Integer
■ byte and Byte
■ short and Short
■ char and Character
■ String
■ enum values

For the exam, we recommend you memorize this list. Note that boolean and long, and their associated wrapper classes, are not supported by switch statements.

• Chapter 2: Operators and Statements

Understanding Java Statements

The switch Statement : **Compile-time Constant Values**

The values in each case statement must be compile-time constant values of the same data type as the switch value. This means you can use only literals, enum constants, or final constant variables of the same data type. By final constant, we mean that the variable must be marked with the final modifier and initialized with a literal value in the same expression in which it is declared.

Chapter 2: Operators and Statements

Understanding Java Statements

The switch Statement : **Compile-time Constant Values**

```
int dayOfWeek = 5;
switch(dayOfWeek) {
  default:
    System.out.println("Weekday");
    break;
  case 0:
    System.out.println("Sunday");
    break;
  case 6:
    System.out.println("Saturday");
    break;
}
```

With a value of dayOfWeek of 5, this code will output:

```
Weekday
```

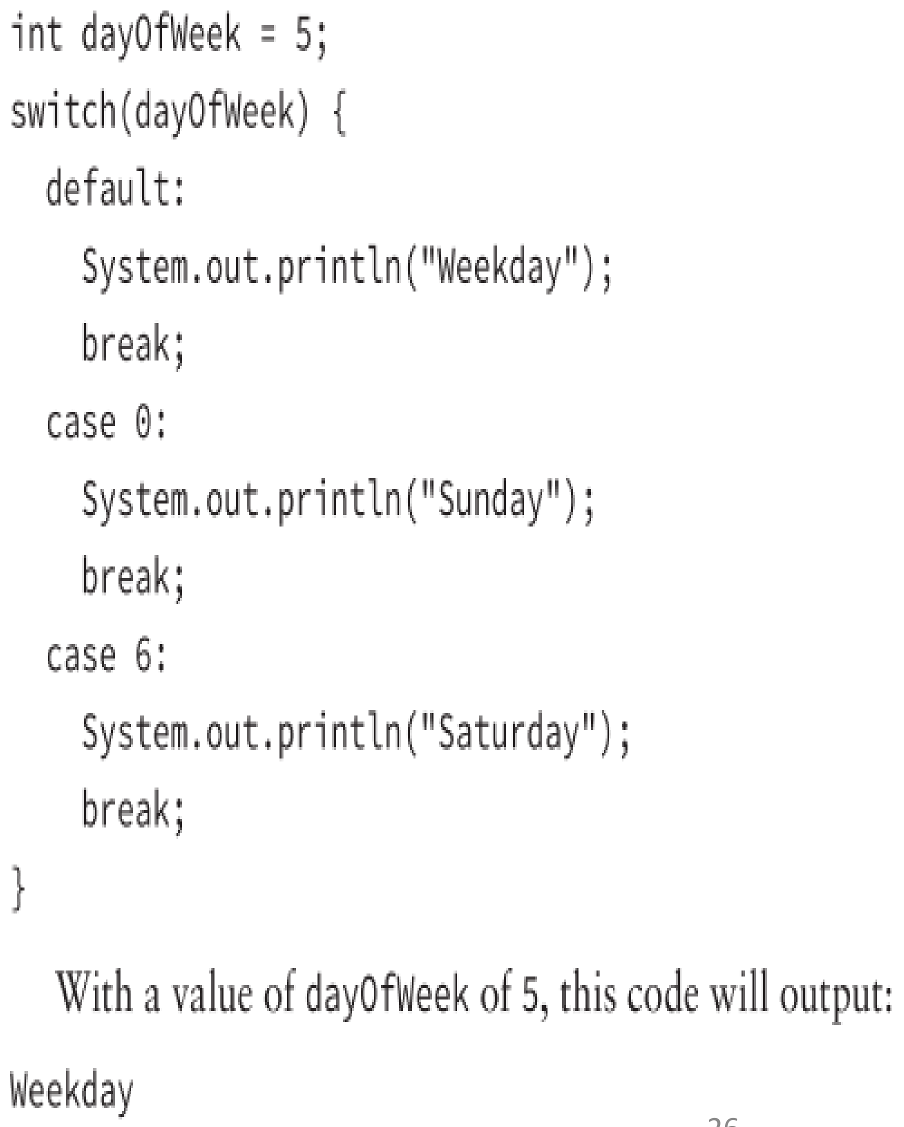
• Chapter 2: Operators and Statements

Understanding Java Statements

The switch Statement

The first thing you may notice is that there is a break statement at the end of each case and default section. We'll discuss break statements in detail when we discuss loops, but for now all you need to know is that they terminate the switch statement and return flow control to the enclosing statement. As we'll soon see, if you leave out the break statement, flow will continue to the next proceeding case or default block automatically.

Another thing you might notice is that the default block is not at the end of the switch statement. There is no requirement that the case or default statements be in a particular order, unless you are going to have pathways that reach multiple sections of the switch block in a single execution.



```
int dayOfWeek = 5;
switch(dayOfWeek) {
  default:
    System.out.println("Weekday");
    break;
  case 0:
    System.out.println("Sunday");
    break;
  case 6:
    System.out.println("Saturday");
    break;
}
```

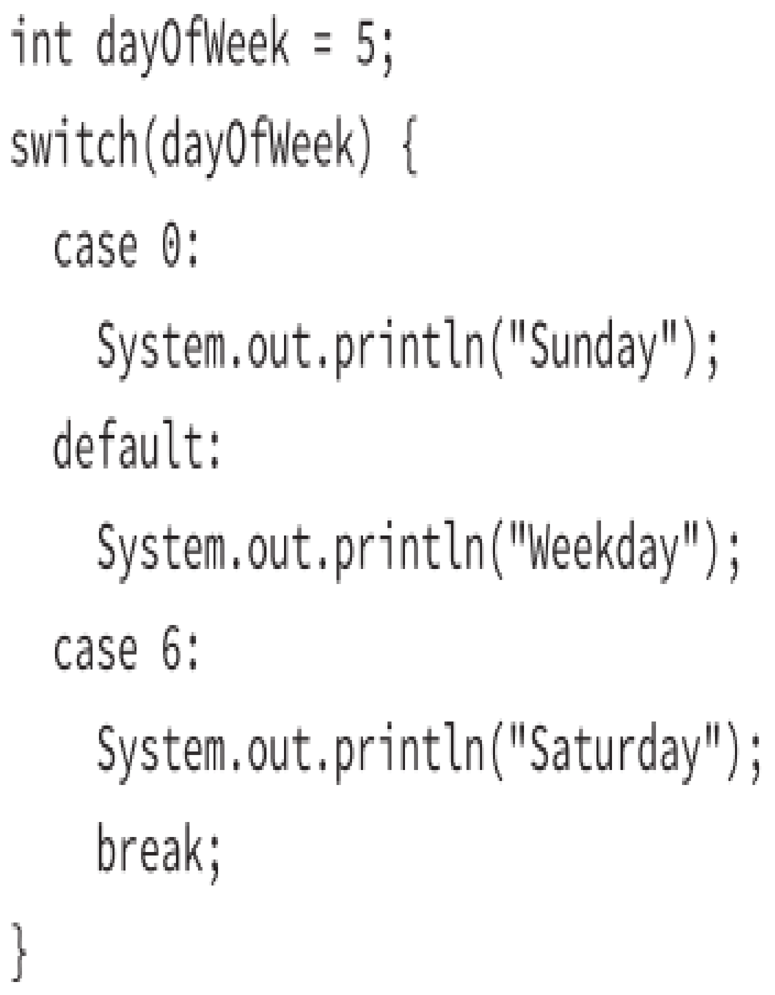
With a value of dayOfWeek of 5, this code will output:

```
Weekday
```

• Chapter 2: Operators and Statements

Understanding Java Statements

The switch Statement



```
int dayOfWeek = 5;
switch(dayOfWeek) {
  case 0:
    System.out.println("Sunday");
  default:
    System.out.println("Weekday");
  case 6:
    System.out.println("Saturday");
    break;
}
```

• Chapter 2: Operators and Statements

Understanding Java Statements

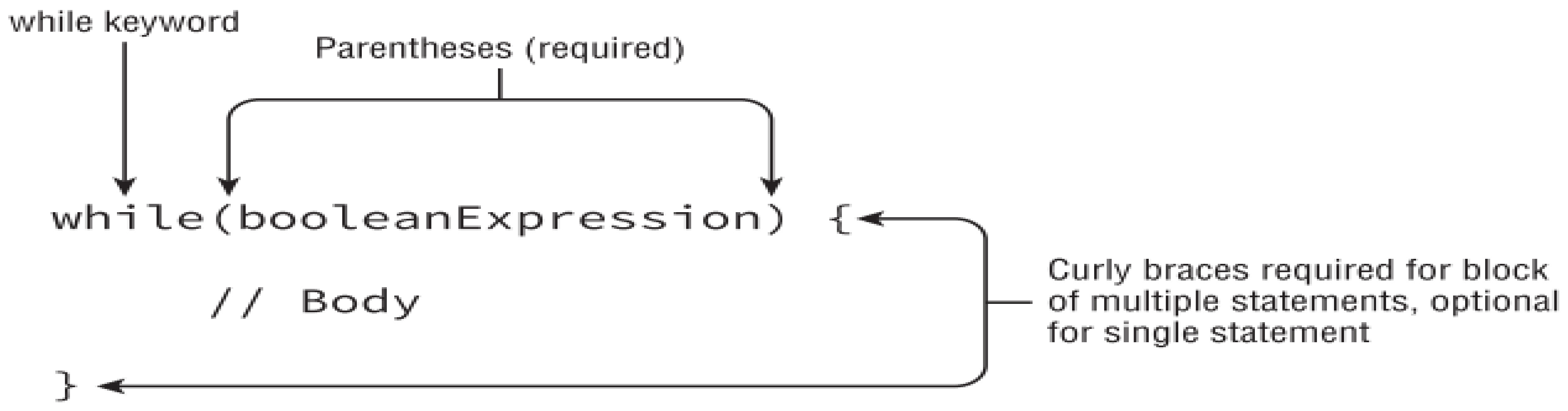
```
private int getSortOrder(String firstName, final String lastName) {
 String middleName = "Patricia";
 final String suffix = "JR";
 int id = 0;
 switch(firstName) {
   case "Test":
     return 52;
   case middleName:   // DOES NOT COMPILE
     id = 5;
     break;
   case suffix:
     id = 0;
     break;
   case lastName:   // DOES NOT COMPILE
     id = 8;
     break;
   case 5:   // DOES NOT COMPILE
     id = 7;
     break;
   case 'J':   // DOES NOT COMPILE
     id = 10;
     break;
   case java.time.DayOfWeek.SUNDAY:   // DOES NOT COMPILE
     id=15;
     break;
 }
 return id;
}
```

Chapter 2: Operators and Statements

Understanding Java Statements

The while Statement

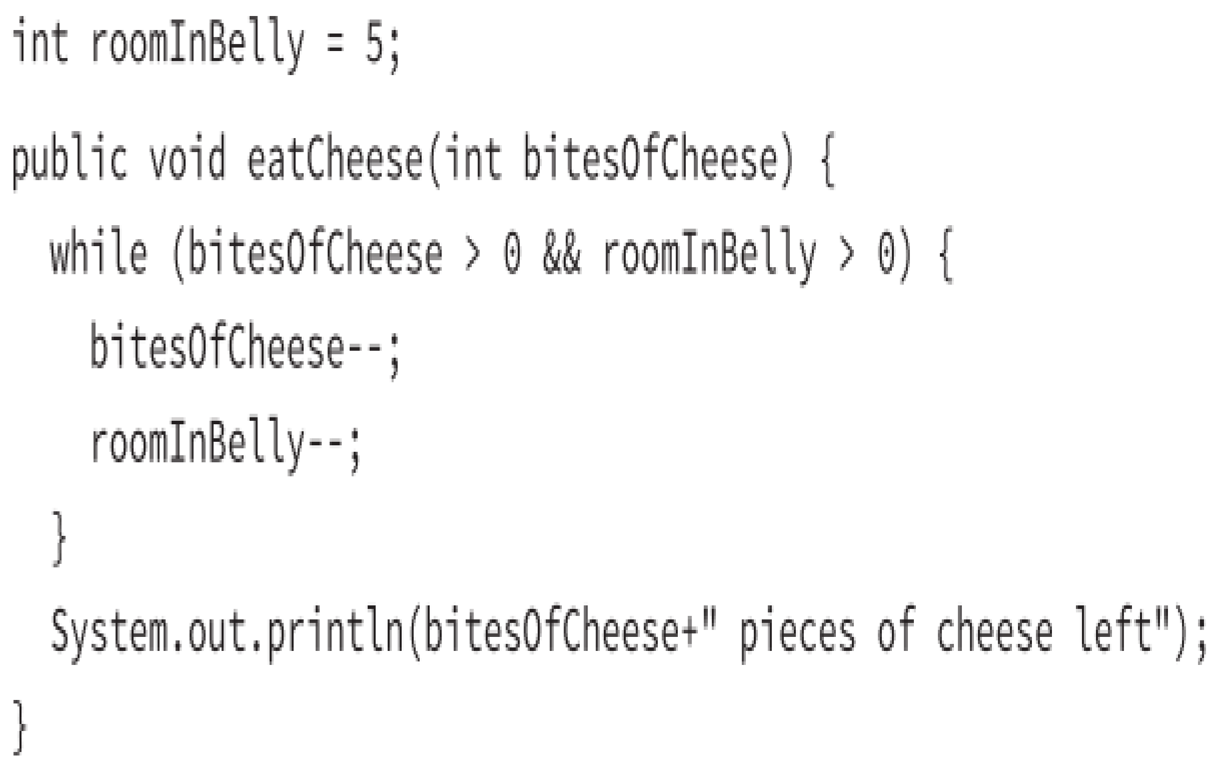
A repetition control structure, which we refer to as a *loop*, executes a statement of code multiple times in succession. By using non constant variables, each repetition of the statement may be different.


while keyword
Parentheses (required)
while(booleanExpression) {
// Body
}
Curly braces required for block of multiple statements, optional for single statement

Chapter 2: Operators and Statements

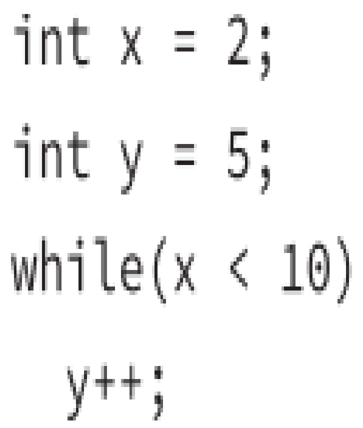
Understanding Java Statements

The while Statement



```
int roomInBelly = 5;

public void eatCheese(int bitesOfCheese) {
  while (bitesOfCheese > 0 && roomInBelly > 0) {
    bitesOfCheese--;
    roomInBelly--;
  }
  System.out.println(bitesOfCheese+" pieces of cheese left");
}
```



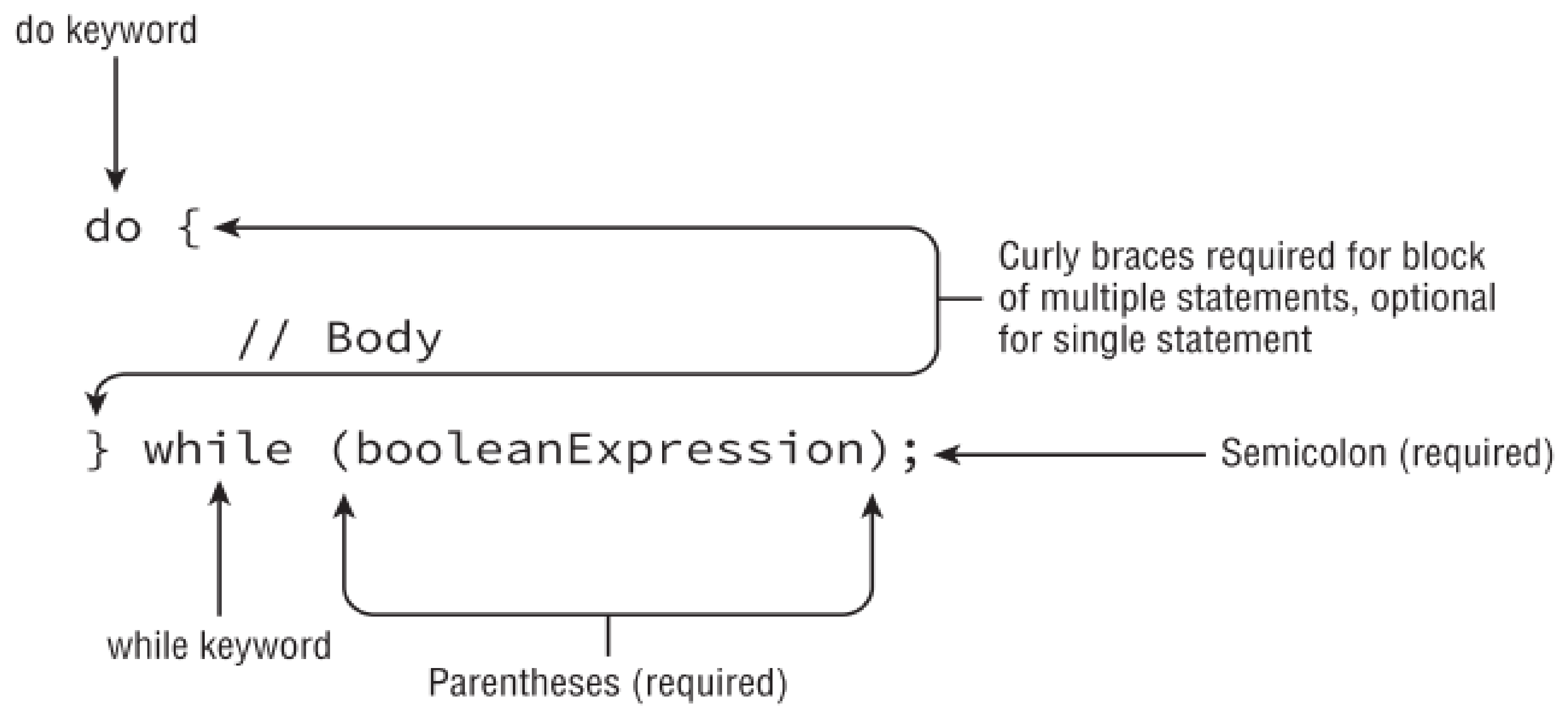
```
int x = 2;
int y = 5;
while(x < 10)
  y++;
```

Chapter 2: Operators and Statements

Understanding Java Statements

The do-while Statement

Java also allows for the creation of a *do-while* loop, which like a while loop, is a repetition control structure with a termination condition and statement, or block of statements. Unlike a while loop, though, a do-while loop guarantees that the statement or block will be executed at least once.

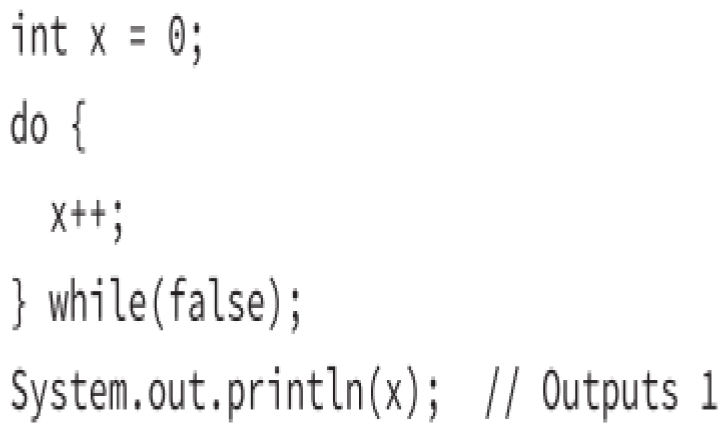


do keyword
do {
// Body
} while (booleanExpression);
Curly braces required for block of multiple statements, optional for single statement
Semicolon (required)
while keyword
Parentheses (required)

- # Chapter 2: Operators and Statements

Understanding Java Statements

The do-while Statement



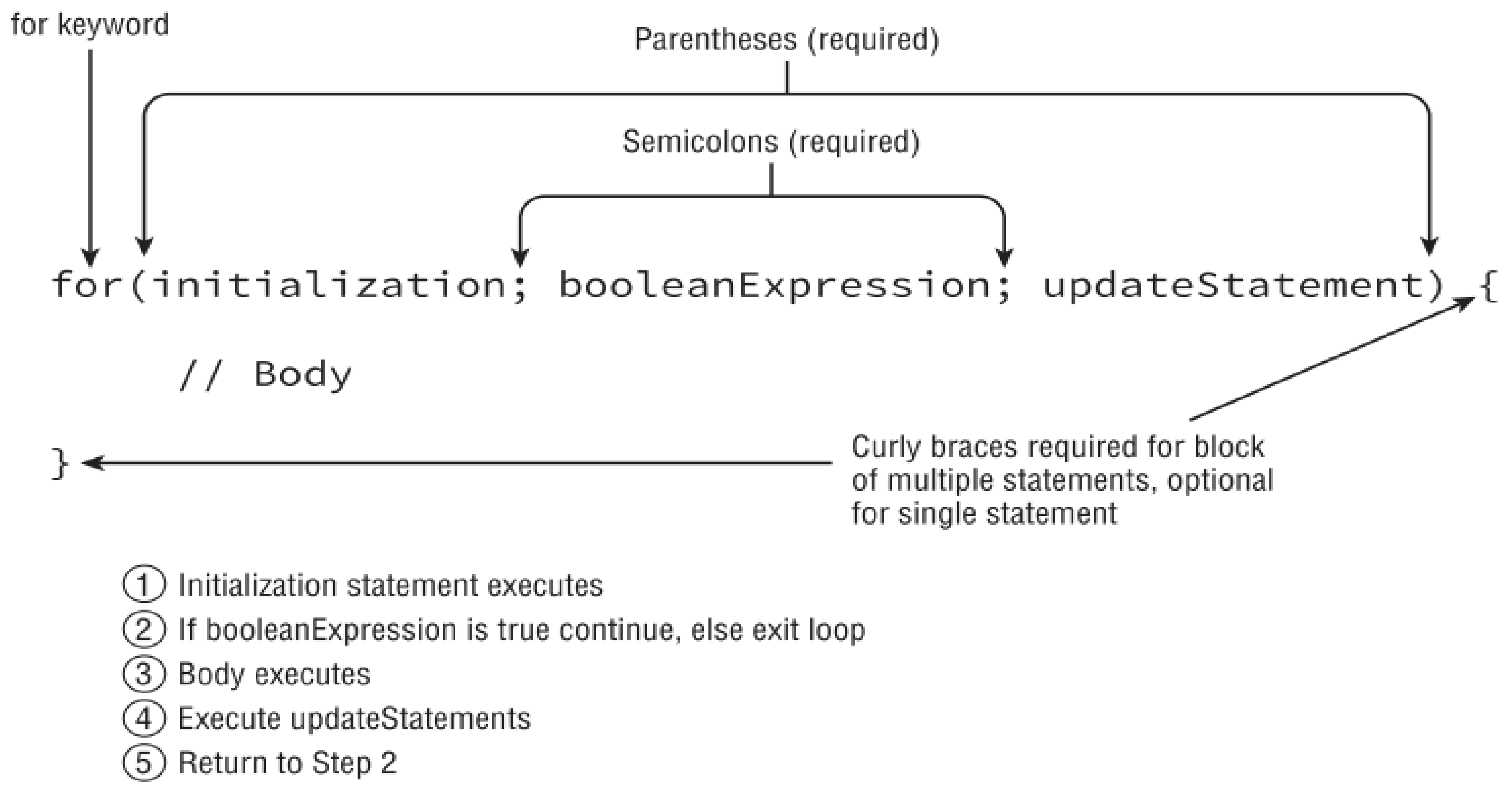
```
int x = 0;
do {
   x++;
} while(false);
System.out.println(x);  // Outputs 1
```

Java will execute the statement block first, and then check the loop condition. Even though the loop exits right away, the statement block was still executed once and the program outputs a 1.

• Chapter 2: Operators and Statements

Understanding Java Statements

The Basic For Statement


for keyword
Parentheses (required)
Semicolons (required)
for(initialization; booleanExpression; updateStatement) {
// Body
}
Curly braces required for block of multiple statements, optional for single statement
① Initialization statement executes
② If booleanExpression is true continue, else exit loop
③ Body executes
④ Execute updateStatements
⑤ Return to Step 2

• Chapter 2: Operators and Statements

Understanding Java Statements

The Basic For Statement

Note that each section is separated by a semicolon. The initialization and update sections may contain multiple statements, separated by commas.

Variables declared in the initialization block of a for loop have limited scope and are only accessible within the for loop. Be wary of any exam questions in which a variable declared within the initialization block of a for loop is available outside the loop.

Alternatively, variables declared before the for loop and assigned a value in the initialization block may be used outside the for loop because their scope precedes the for loop creation.

```
for(int i = 0; i < 10; i++) {
  System.out.print(i + " ");
}
```

```
for( ; ; ) {
  System.out.println("Hello World");
}
```

Chapter 2: Operators and Statements

Understanding Java Statements

The infinite loop

```
for( ; ; ) {
  System.out.println("Hello World");
}
```

Although this for loop may look like it will throw a compiler error, it will in fact compile and run without issue. It is actually an infinite loop that will print the same statement repeatedly. This example reinforces the fact that the components of the for loop are each optional. Note that the semicolons separating the three sections are required, as for(;) and for( ) will not compile.

• Chapter 2: Operators and Statements

Understanding Java Statements

Adding Multiple Terms to the for Statement

```
int x = 0;
for(long y = 0, z = 4; x < 5 && y < 10; x++, y++) {
  System.out.print(y + " ");
}
```

This code demonstrates three variations of the for loop you may not have seen. First, you can declare a variable, such as x in this example, before the loop begins and use it after it completes. Second, your initialization block, boolean expression, and update statements can include extra variables that may not reference each other..

For example, z is defined in the initialization block and is never used. Finally, the update statement can modify multiple variables. This code will print the following when executed: 0 1 2 3 4

• Chapter 2: Operators and Statements

Understanding Java Statements

Redeclaring a Variable in the Initialization Block

```
int x = 0;
for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) {   // DOES NOT COMPILE
  System.out.print(x + " ");
}
```

This example looks similar to the previous one, but it does not compile because of the initialization block. The difference is that x is repeated in the initialization block after already being declared before the loop, resulting in the compiler stopping because of a duplicate variable declaration.

We can fix this loop by changing the declaration of x and y as follows :

```
int x = 0;
long y = 10;
for(y = 0, x = 4; x < 5 && y < 10; x++, y++) {
  System.out.print(x + " ");
}
```

Note that this variation will now compile because the initialization block simply assigns a value to x and does not declare it.

• Chapter 2: Operators and Statements

Understanding Java Statements

Using Incompatible Data Types in the Initialization Block

```
for(long y = 0, int x = 4; x < 5 && y<10; x++, y++) {   // DOES NOT COMPILE
System.out.print(x + " ");
}
```

The variables in the initialization block must all be of the same type.

Using Loop Variables Outside the Loop

```
for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) {
  System.out.print(y + " ");
}
System.out.print(x);   // DOES NOT COMPILE
```

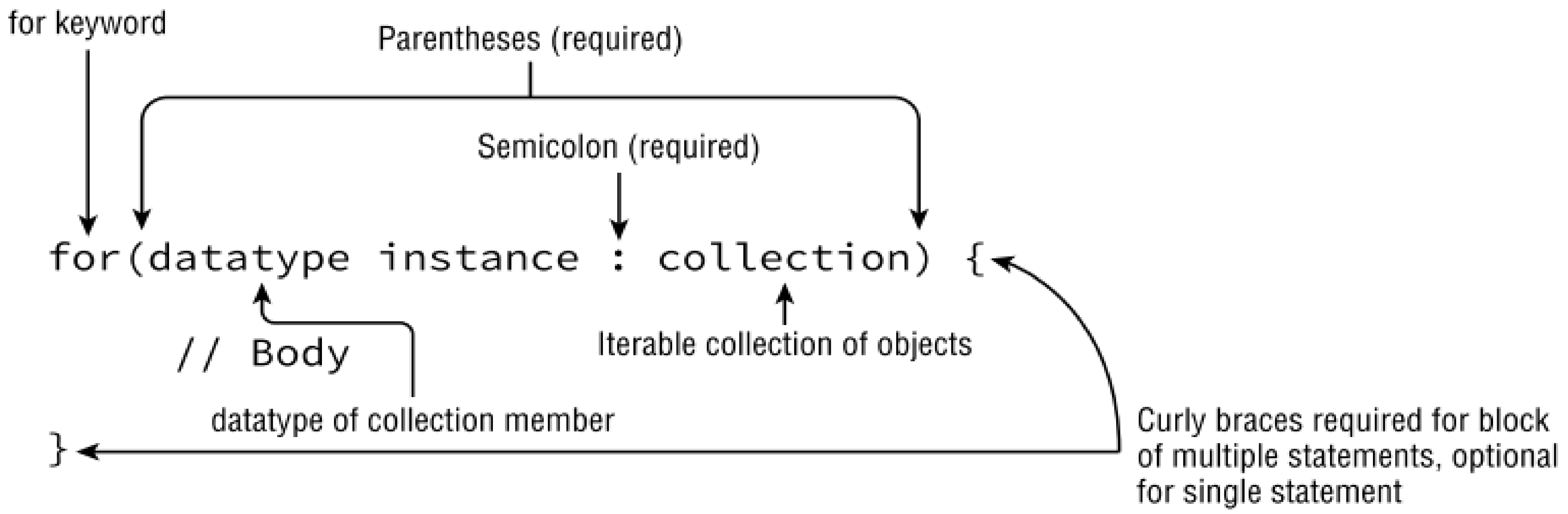
The final variation on the second example will not compile for a different reason than the previous examples. If you notice, x is defined in the initialization block of the loop, and then used after the loop terminates. Since x was only scoped for the loop, using it outside the loop will throw a compiler error.

- # Chapter 2: Operators and Statements

Understanding Java Statements

The for-each Statement

Starting with Java 5.0, Java developers have had a new type of enhanced for loop at their disposal, one specifically designed for iterating over arrays and Collection objects. This enhanced for loop, which for clarity we'll refer to as a *for-each* loop,



for keyword
Parentheses (required)
Semicolon (required)
for(datatype instance : collection) {
// Body
}
Iterable collection of objects
datatype of collection member
Curly braces required for block of multiple statements, optional for single statement

• Chapter 2: Operators and Statements

Understanding Java Statements

The for-each Statement

- The for-each loop declaration is composed of an initialization section and an object to be iterated over.
- The right-hand side of the for-each loop statement must be a built-in Java array or an object whose class implements java.lang.Iterable, which includes most of the Java Collections framework.

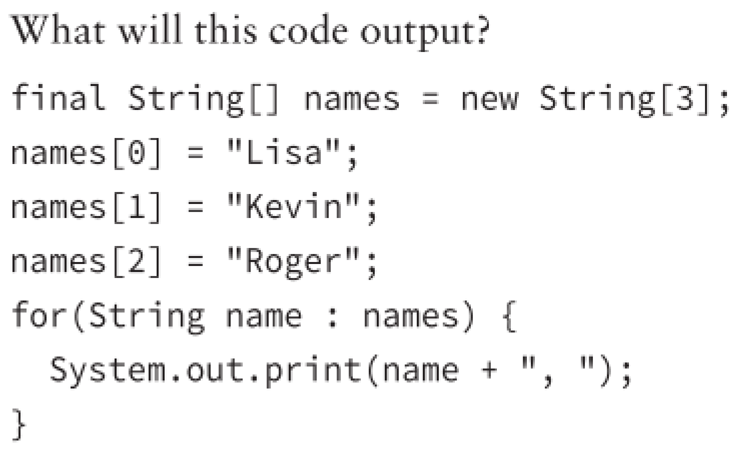
For the OCA exam, the only members of the Collections framework that you need to be aware of are List and ArrayList.

- The left-hand side of the for-each loop must include a declaration for an instance of a variable, whose type matches the type of a member of the array or collection in the right-hand side of the statement.
- On each iteration of the loop, the named variable on the left-hand side of the statement is assigned a new value from the array or collection on the right-hand side of the statement.

Chapter 2: Operators and Statements

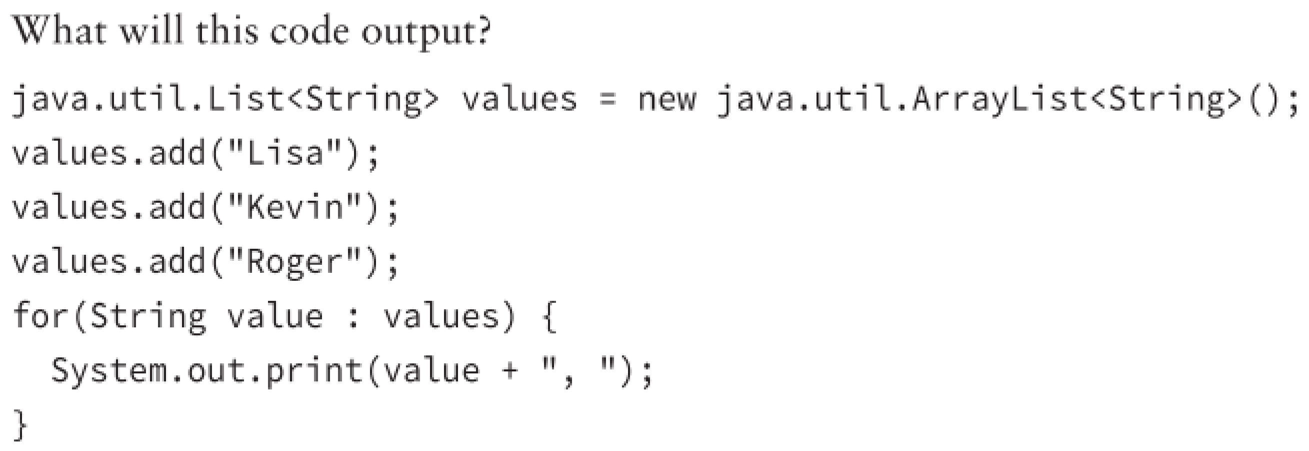
Understanding Java Statements

The for-each Statement



- What will this code output?

```
final String[] names = new String[3];
names[0] = "Lisa";
names[1] = "Kevin";
names[2] = "Roger";
for(String name : names) {
  System.out.print(name + ", ");
}
```



What will this code output?

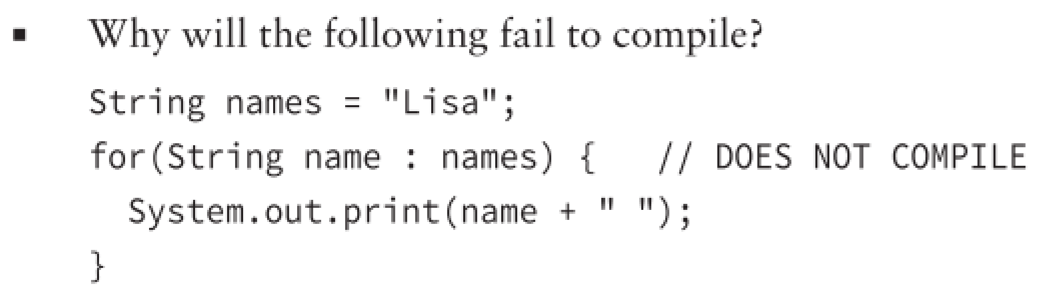
```
java.util.List<String> values = new java.util.ArrayList<String>();
values.add("Lisa");
values.add("Kevin");
values.add("Roger");
for(String value : values) {
  System.out.print(value + ", ");
}
```

• Chapter 2: Operators and Statements

Understanding Java Statements

The for-each Statement

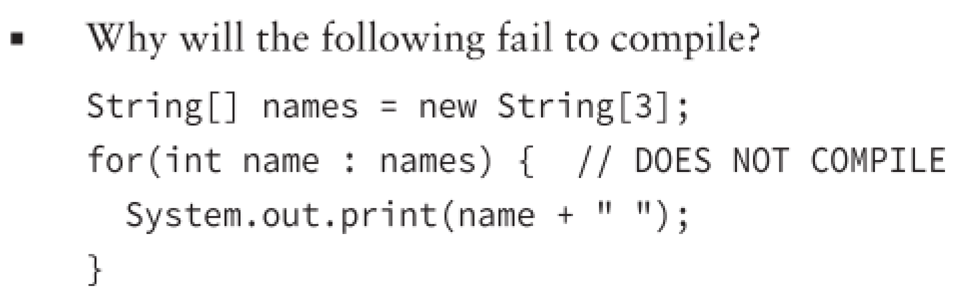
When you see a for-each loop on the exam, make sure the right-hand side is an array or Iterable object and the left-hand side has a matching type. For example, the two examples that follow will not compile.



- Why will the following fail to compile?

```
String names = "Lisa";
for(String name : names) {    // DOES NOT COMPILE
  System.out.print(name + " ");
}
```

In this example, the String names is not an array, nor does it implement java.lang.Iterable, so the compiler will throw an exception since it does not know how to iterate over the String.



- Why will the following fail to compile?

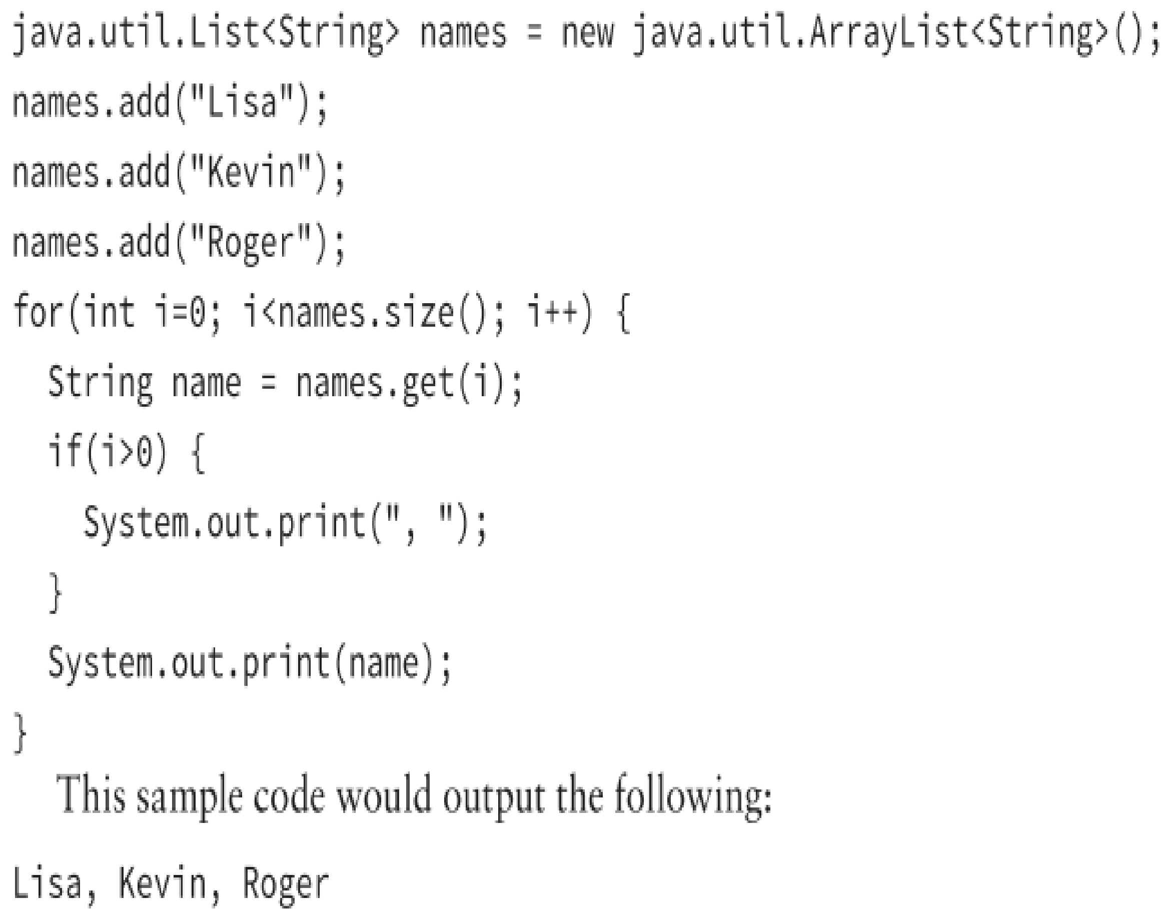
```
String[] names = new String[3];
for(int name : names) {   // DOES NOT COMPILE
  System.out.print(name + " ");
}
```

This code will fail to compile because the left-hand side of the for-each statement does not define an instance of String. Notice that in this last example, the array is initialized with three null pointer values. In and of itself, that will not cause the code to not compile, as a corrected loop would just output null three times.

• Chapter 2: Operators and Statements

Understanding Java Statements

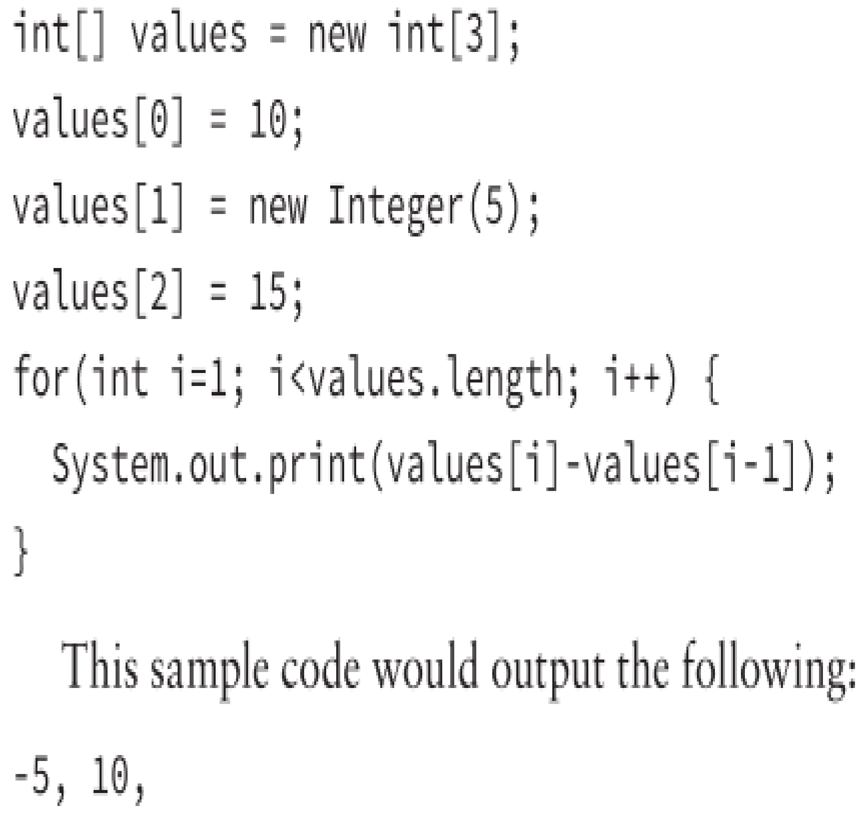
The for-each Statement



```
java.util.List<String> names = new java.util.ArrayList<String>();
names.add("Lisa");
names.add("Kevin");
names.add("Roger");
for(int i=0; i<names.size(); i++) {
  String name = names.get(i);
  if(i>0) {
    System.out.print(", ");
  }
  System.out.print(name);
}
```

This sample code would output the following:

```
Lisa, Kevin, Roger
```



```
int[] values = new int[3];
values[0] = 10;
values[1] = new Integer(5);
values[2] = 15;
for(int i=1; i<values.length; i++) {
  System.out.print(values[i]-values[i-1]);
}
```

This sample code would output the following:

```
-5, 10,
```

• Chapter 2: Operators and Statements

Understanding Advanced Flow Control

Nested Loops

First off, loops can contain other loops. For example, consider the following code that iterates over a two-dimensional array, an array that contains other arrays as its members. We'll cover multidimensional arrays in detail in Chapter 3, but for now assume the following is how you would declare a two-dimensional array.

```
int[][] myComplexArray = {{5,2,1,3},{3,9,8,9},{5,7,12,7}};
for(int[] mySimpleArray : myComplexArray) {
  for(int i=0; i<mySimpleArray.length; i++) {
    System.out.print(mySimpleArray[i]+"\t");
  }
  System.out.println();
}
```

```
5    2    1    3
3    9    8    9
5    7    12    7
```

• Chapter 2: Operators and Statements

Understanding Advanced Flow Control

Nested Loops

Nested loops can include while and do-while, as shown in this example. See if you can determine what this code will output.

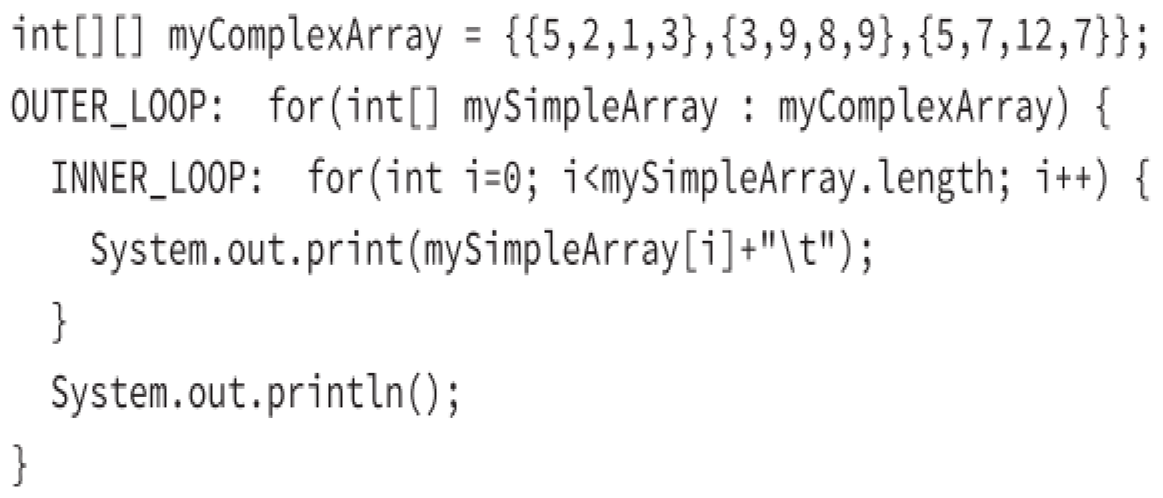
```
int x = 20;
while(x>0) {
  do {
    x -= 2
  } while (x>5);
  x--;
  System.out.print(x+"\t");
}
```

• Chapter 2: Operators and Statements

Understanding Advanced Flow Control

Adding Optional Labels

One thing we skipped when we presented if-then statements, switch statements, and loops is that they can all have optional labels. A *label* is an optional pointer to the head of a statement that allows the application flow to jump to it or break from it. It is a single word that is proceeded by a colon (:). For example, we can add optional labels to one of the previous examples:



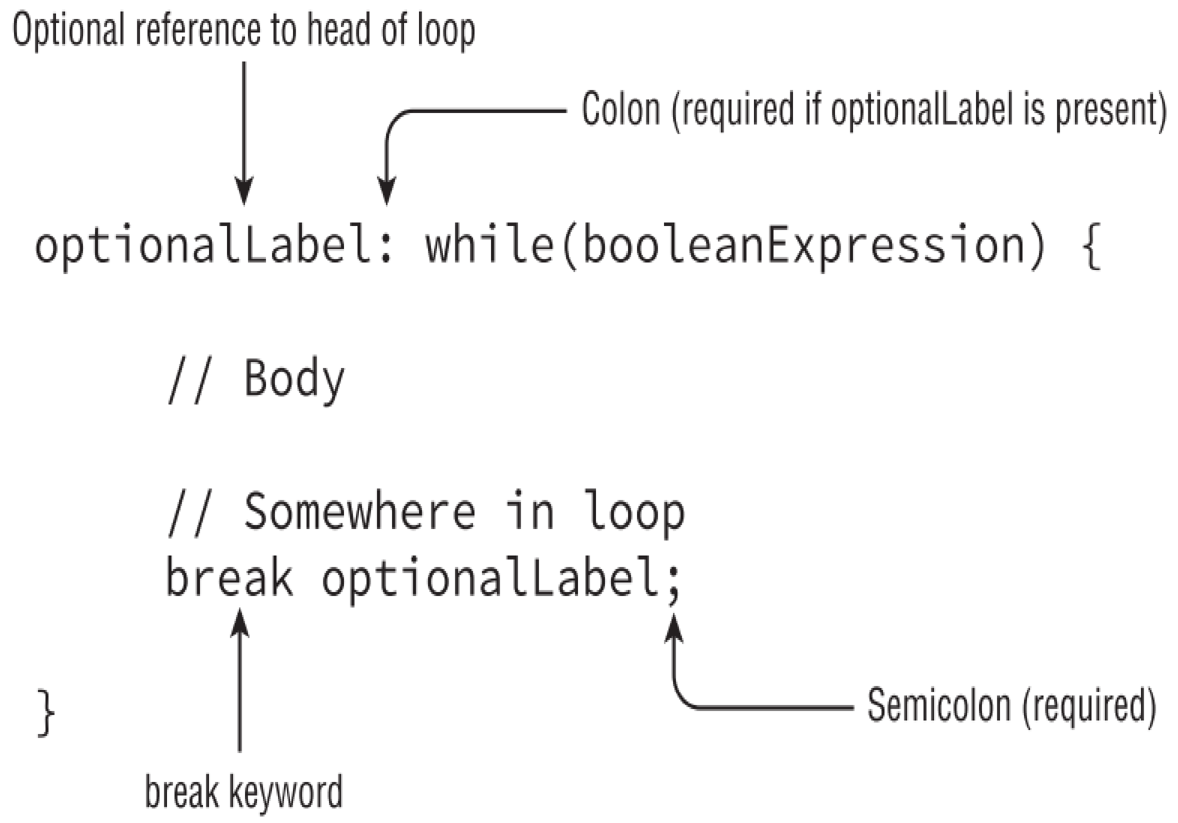
```
int[][] myComplexArray = {{5,2,1,3},{3,9,8,9},{5,7,12,7}};
OUTER_LOOP:  for(int[] mySimpleArray : myComplexArray) {
  INNER_LOOP:  for(int i=0; i<mySimpleArray.length; i++) {
    System.out.print(mySimpleArray[i]+"\t");
  }
  System.out.println();
}
```

Chapter 2: Operators and Statements

Understanding Advanced Flow Control

The break Statement

As you saw when working with switch statements, a *break* statement transfers the flow of control out to the enclosing statement. The same holds true for break statements that appear inside of while, do-while, and for loops.



```
optionalLabel: while(booleanExpression) {

      // Body

      // Somewhere in loop
      break optionalLabel;

}
```

Optional reference to head of loop
Colon (required if optionalLabel is present)
Semicolon (required)
break keyword

Notice in the figure that the break statement can take an optional label parameter. Without a label parameter, the break statement will terminate the nearest inner loop it is currently in the process of executing. The optional label parameter allows us to break out of a higher level outer loop.

- Chapter 2: Operators and Statements

Understanding Advanced Flow Control

The break Statement

```
public class SearchSample {
  public static void main(String[] args) {
    int[][] list = {{1,13,5},{1,2,5},{2,7,2}};
    int searchValue = 2;
    int positionX = -1;
    int positionY = -1;
    PARENT_LOOP: for(int i=0; i<list.length; i++) {
      for(int j=0; j<list[i].length; j++) {
        if(list[i][j]==searchValue) {
          positionX = i;
          positionY = j;
          break PARENT_LOOP;
        }
      }
    }
    if(positionX==-1 || positionY==-1) {
      System.out.println("Value "+searchValue+" not found");
    } else {
      System.out.println("Value "+searchValue+" found at: " +
        "("+positionX+","+positionY+")");
    }
  }
}
```

```
Value 2 found at: (1,1)
```

- # Chapter 2: Operators and Statements

Understanding Advanced Flow Control

The break Statement

.

```
if(list[i][j]==searchValue) {
  positionX = i;
  positionY = j;
  break;
}
```

```
Value 2 found at: (2,0)
```

```
if(list[i][j]==searchValue) {
  positionX = i;
  positionY = j;
}
```

```
Value 2 found at: (2,2)
```

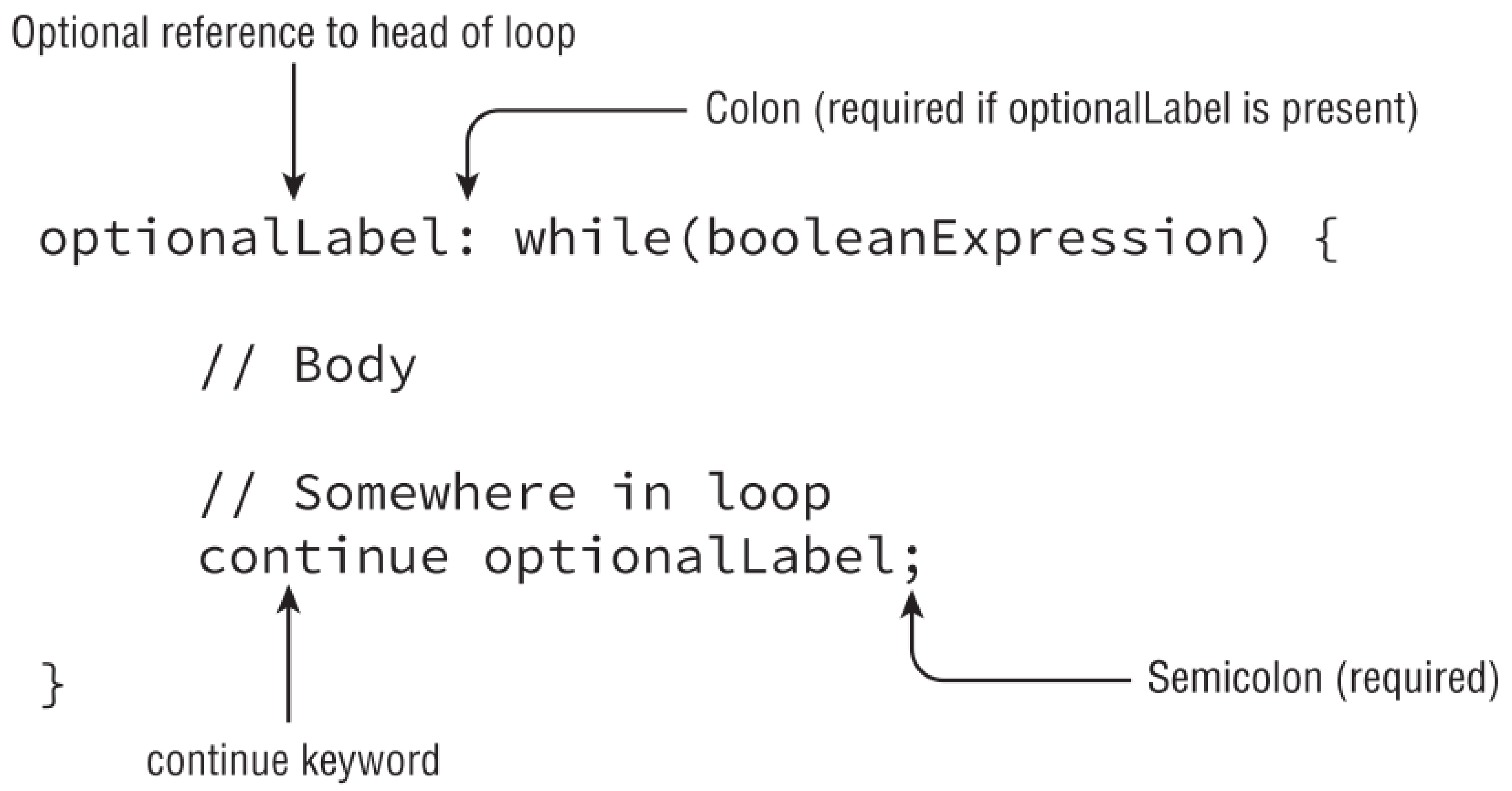
• Chapter 2: Operators and Statements

Understanding Advanced Flow Control

The continue Statement

the *continue* statement, is a statement that causes flow to finish the execution of the current loop

.



```
optionalLabel: while(booleanExpression) {

    // Body

    // Somewhere in loop
    continue optionalLabel;

}
```

Optional reference to head of loop
Colon (required if optionalLabel is present)
Semicolon (required)
continue keyword

• Chapter 2: Operators and Statements

Understanding Advanced Flow Control

The continue Statement

```
public class SwitchSample {
  public static void main(String[] args) {
    FIRST_CHAR_LOOP: for (int a = 1; a <= 4; a++) {
      for (char x = 'a'; x <= 'c'; x++) {
        if (a == 2 || x == 'b')
          continue FIRST_CHAR_LOOP;
        System.out.print(" " + a + x);
      }
    }
  }
}
```

```
 1a 3a 4a
```

If we remove the FIRST_CHAR_LOOP label the result will be:

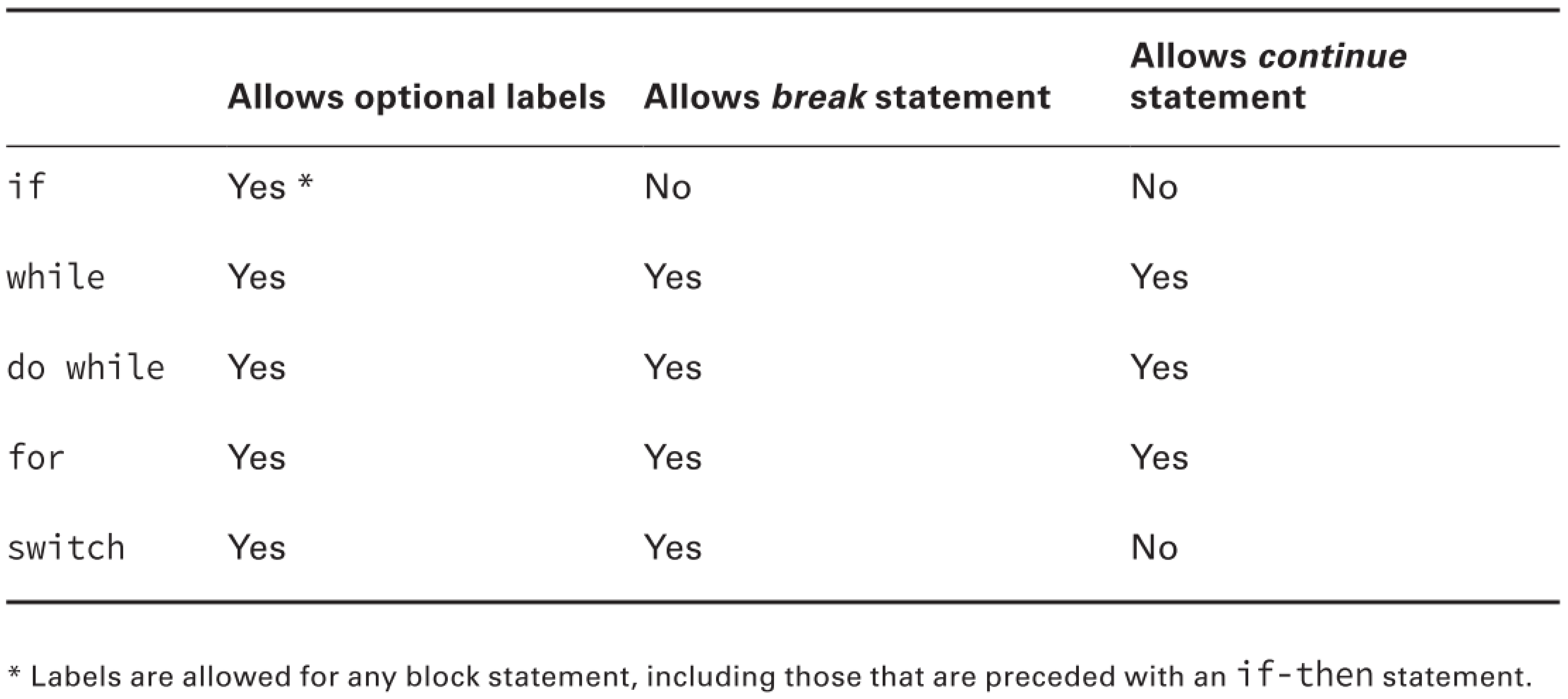
```
1a 1c 3a 3c 4a 4c
```

If we remove the continue

```
1a 1b 1c 2a 2b 2c 3a 3b 3c 4a 4b 4c
```

Chapter 2: Operators and Statements

Understanding Advanced Flow Control



	Allows optional labels	Allows *break* statement	Allows *continue* statement
`if`	Yes *	No	No
`while`	Yes	Yes	Yes
`do while`	Yes	Yes	Yes
`for`	Yes	Yes	Yes
`switch`	Yes	Yes	No

* Labels are allowed for any block statement, including those that are preceded with an `if-then` statement.