Control Flow 4

Chapter Topics

- Choosing between alternative actions with the selection statements: the `if` statement, the `if-else` statement, the `switch` statement, and the `switch` expression
- Repeatedly executing code with the iteration statements: the `for` loop (`for(;;)`), the enhanced `for` loop (`for(:)`), the `while` loop, and the `do-while` loop
- Understanding control transfer with the `yield`, `break`, `continue`, and `return` statements, including labeled statements

Java SE 17 Developer Exam Objectives	
[2.1] Create program flow control constructs including if/else, switch statements and expressions, loops, and break and continue statements	§4.1, p. 152 *to* §4.13, p. 184
Java SE 11 Developer Exam Objectives	
[2.1] Create and use loops, if/else, and switch statements	§4.1, p. 152 *to* §4.8, p. 176

Control flow statements determine *the flow of control* in a program during execution, meaning the order in which statements are executed in a running program. There are three main categories of control flow statements:

- *Selection* statements: `if`, `if-else`, and `switch`
- *Iteration* statements: `while`, `do-while`, basic `for`, and enhanced `for` loops
- *Transfer* statements: `yield`, `break`, `continue`, and `return`

Each category of statements is discussed in subsequent sections.

4.1 Selection Statements

Java provides selection statements that allow the program to choose between alternative actions during execution. The choice is based on criteria specified in the selection statement. These selection statements are

- The simple `if` statement

- The `if-else` statement
- The `switch` statement and the `switch` expression

The Simple `if` Statement

The simple `if` statement has the following syntax:

```
if (condition)
  statement
```

It is used to decide whether an action is to be performed or not, based on a *condition*. The action to be performed is specified by *statement*, which can be a single statement or a code block. The *condition* must evaluate to a `boolean` or `Boolean` value. In the latter case, the `Boolean` value is unboxed to the corresponding `boolean` value.

The semantics of the simple `if` statement are straightforward. The *condition* is evaluated first. If its value is `true`, *statement* (called the `if` block) is executed and then execution continues with the rest of the program. If the value is `false`, the `if` block is skipped and execution continues with the rest of the program. The semantics are illustrated by the activity diagram in **Figure 4.1a**.

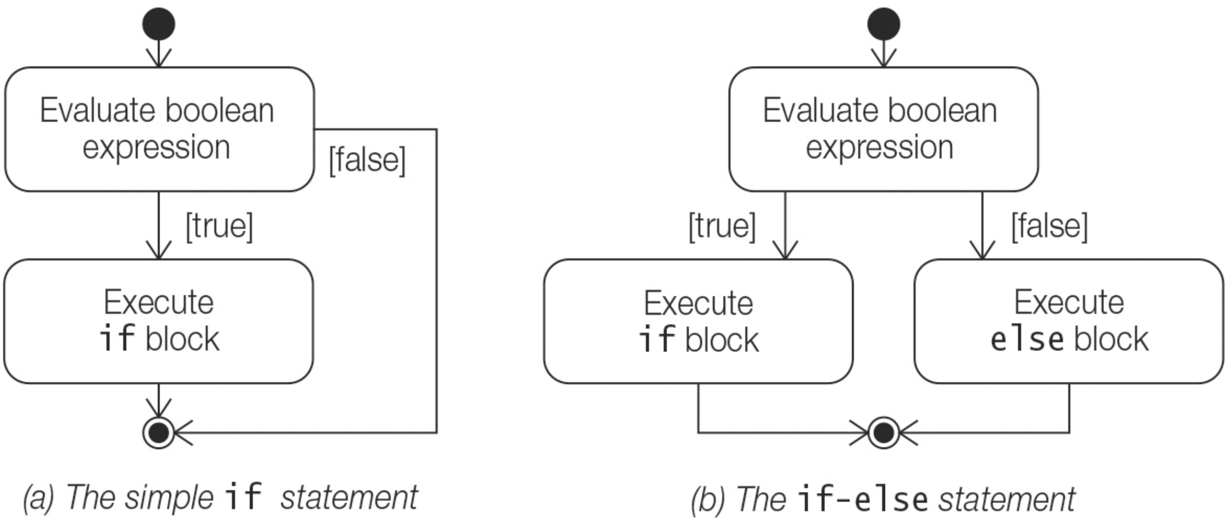

Figure 4.1 *Activity Diagram for* `if` *Statements*

In the following examples of the `if` statement, it is assumed that the variables and the methods have been appropriately defined:

Click here to view code image

```
if (emergency)                // emergency is a boolean variable
  operate();

if (temperature > critical)
  soundAlarm();
```

```
if (isLeapYear() && endOfCentury())
  celebrate();

if (catIsAway()) {         // Block
  getFishingRod();
  goFishing();
}
```

Note that *statement* can be a *block*, and the block notation is necessary if more than one statement is to be executed when the *condition* is `true`.

Since the *condition* evaluates to a `boolean` value, it avoids a common programming error: using an expression of the form `(a=b)` as the condition, where inadvertently an assignment operator is used instead of a relational operator. The compiler will flag this as an error, unless both `a` and `b` are boolean.

Note that the `if` block can be any valid statement. In particular, it can be the empty statement (`;`) or the empty block (`{}`). A common programming error is inadvertent use of the empty statement.

Click here to view code image

```
if (emergency); // Empty if block
  operate();    // Executed regardless of whether it was an emergency
```

The `if-else` Statement

The `if-else` statement is used to decide between two actions, based on a *condition*. It has the following syntax:

```
if (condition)
  statement1
else
  statement2
```

The *condition* is evaluated first. If its value is `true` (or unboxed to `true`), *statement*$_1$ (the `if` block) is executed and then execution continues with the rest of the program. If the value is `false` (or unboxed to `false`), *statement*$_2$ (the `else` block) is executed and then execution continues with the rest of the program. In other words, one of two mutually exclusive actions is performed. The `else` clause is optional; if omitted, the construct is equivalent to the simple `if` statement. The semantics are illustrated by the activity diagram in Figure 4.1b.

In the following examples of the `if-else` statement, it is assumed that all variables and methods have been appropriately defined:

Click here to view code image

```
if (emergency)
  operate();
else
  joinQueue();

if (temperature > critical)
  soundAlarm();
else
  businessAsUsual();

if (catIsAway()) {
  getFishingRod();
  goFishing();
} else
  playWithCat();
```

Since actions can be arbitrary statements, the `if` statements can be nested.

Click here to view code image

```
if (temperature >= upperLimit) {        // (1)
  if (danger)                           // (2) Simple if.
    soundAlarm();
  if (critical)                         // (3)
    evacuate();
  else                                  // Goes with if at (3).
    turnHeaterOff();
} else                                  // Goes with if at (1).
    turnHeaterOn();
```

The use of block notation, `{}`, can be critical to the execution of `if` statements. The `if` statements (A) and (B) in the following examples do *not* have the same meaning. The `if` statements (B) and (C) are the same, with extra indentation used in (C) to make the meaning evident. Leaving out the block notation in this case could have catastrophic consequences: The heater could be turned on when the temperature is above the upper limit.

Click here to view code image

```
// (A):
if (temperature > upperLimit) {          // (1) Block notation.
  if (danger) soundAlarm();              // (2)
```

```
} else                                      // Goes with if at (1).
  turnHeaterOn();

// (B):
if (temperature > upperLimit)               // (1) Without block notation.
  if (danger) soundAlarm();                 // (2)
else turnHeaterOn();                        // Goes with if at (2).

// (C):
if (temperature > upperLimit)               // (1)

  if (danger)                               // (2)
    soundAlarm();
  else                                      // Goes with if at (2).
    turnHeaterOn();
```

The rule for matching an `else` clause is that an `else` clause always refers to the nearest `if` that is not already associated with another `else` clause. Block notation and proper indentation can be used to make the meaning obvious.

Cascading of `if-else` statements comprises a sequence of nested `if-else` statements where the `if` block of the next `if-else` statement is joined to the `else` clause of the previous `if-else` statement. The decision to execute a block is then based on all the conditions evaluated so far.

Click here to view code image

```
if (temperature >= upperLimit) {                          // (1)
  soundAlarm();
  turnHeaterOff();
} else if (temperature < lowerLimit) {                    // (2)
  soundAlarm();

  turnHeaterOn();
} else if (temperature == (upperLimit-lowerLimit)/2) {    // (3)
  doingFine();
} else                                                    // (4)
  noCauseToWorry();
```

The block corresponding to the first `if` condition that evaluates to `true` is executed, and the remaining `if` statements are skipped. In the preceding example, the block at (3) will execute only if the conditions at (1) and (2) are `false` and the condition at (3) is `true`. If none of the conditions is `true`, the block associated with the last `else` clause is executed. If there is no last `else` clause, no actions are performed.

4.2 The `switch` Statement

The `switch` construct implements a *multi-way branch* that allows program control to be transferred to a specific entry point in the code of the `switch` block based on a computed value. Java has two variants of the `switch` construct (the `switch` *statement* and the `switch` *expression*), and each of them can be written in two different ways (one using the *colon notation* and the other using the *arrow notation*). This section covers the two forms of the `switch` statement. Particular details of the `switch` expression are covered in the next section (p. 164).

The `switch` Statement with the Colon (`:`) Notation

We will first look at the `switch` *statement* defined using the *colon notation*, illustrated in Figure 4.2.

Figure 4.2 *Form of the* `switch` *Statement with the Colon Notation*

```
switch (selector_expression) {
  // Switch block with statement groups defined using colon notation:
  case CC:                                statements
  case CC_1: case CC_2: ... case CC_n:    statements
  case CC_3, CC_4, ..., CC_m:             statements
  ...
  default: ...
}
```


```
switch (selector_expression) {
  // Switch block with statement groups defined using colon notation:
  case CC:                                statements
  case CC_1: case CC_2: ... case CC_n:    statements
  case CC_3, CC_4, ..., CC_m:             statements
  ...
  default: ...
}
```


Conceptually, the `switch` statement can be used to choose one among many alternative actions, based on the value of an expression. The syntax of the `switch` statement comprises a *selector expression* followed by a `switch` *block*. The selector expression must evaluate to a value whose type must be one of the following:

- A primitive data type: `char`, `byte`, `short`, or `int`
- A wrapper type: `Character`, `Byte`, `Short`, or `Integer`
- An enum type (§5.13, p. 287)
- The type `String` (§8.4, p. 439)

Note that the type of the selector expression cannot be `boolean`, `long`, or floating-point. The statements in the `switch` block can have `case` *labels*, where each `case` label specifies one or more `case` *constants* (*CC*), thereby defining entry points in the `switch` block where control can be transferred depending on the value of the selector expression. The `switch` block must be compatible with the type of the selector expression, otherwise a compile-time error occurs.

The execution of the `switch` statement proceeds as follows:

- The selector expression is evaluated first. If the value is a wrapper type, an unboxing conversion is performed (§2.3, p. 45). If the selector expression evaluates to `null`, a `NullPointerException` is thrown.
- The value of the selector expression is compared with the constants in the `case` labels. Control is transferred to the start of the *statements* associated with the `case` label that has a `case` constant whose value is equal to the value of the selector expression. Note that a colon (:) prefixes the associated statements that can be any *group of statements*, including a statement block. After execution of the associated statements, control *falls through* to the *next* group of statements, unless this was the last group of statements declared or control was transferred out of the `switch` statement.
- If no `case` label has a `case` constant that is equal to the value of the selector expression, the statements associated with the `default` label are executed. After execution of the associated statements, control *falls through* to the *next* group of statements, unless this was the last group of statements declared or control was transferred out of the `switch` statement.

Figure 4.3 illustrates the flow of control through a `switch` statement where the `default` label is declared last and control is not transferred out of the `switch` statement in the preceding group of statements.

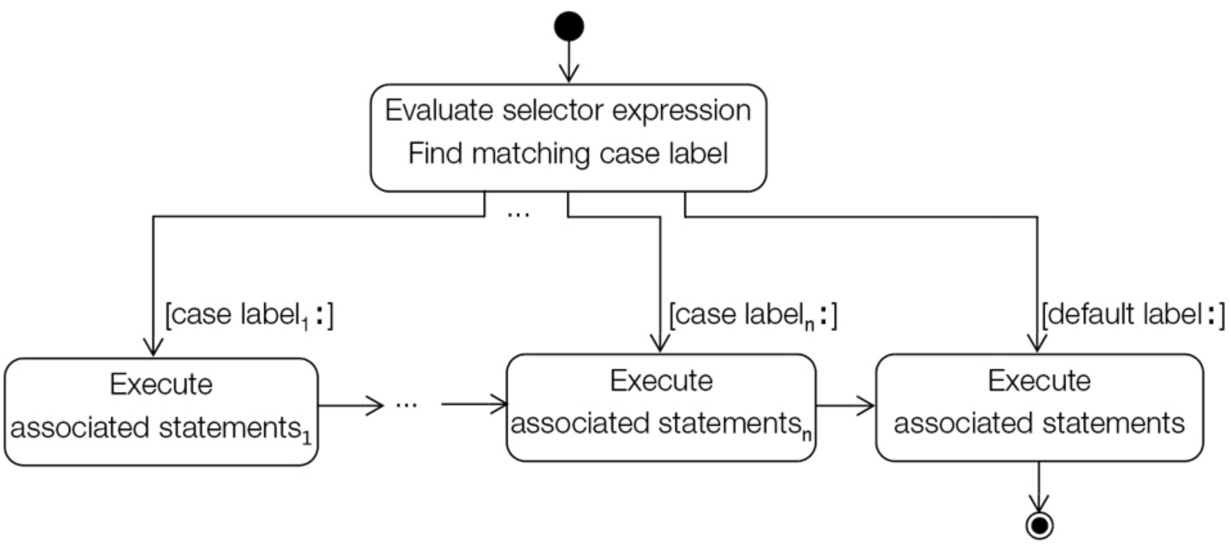

Figure 4.3 *Activity Diagram for the* `switch` *Statement with the Colon Notation*

All `case` labels (including the `default` label) are optional and can be defined in any order in the `switch` block. All `case` labels and the `default` label are separated from their associated group of statements by a colon (`:`). A list of `case` labels can be associated with the same statements, and a `case` label can specify a comma-separated list of `case` constants. At most, one `default` label can be present in a `switch` statement. If no valid `case` labels are found and the `default` label is omitted, the whole `switch` statement is skipped.

The `case` *constants* (*CC*) in the `case` labels are constant expressions whose values must be unique, meaning no duplicate values are allowed. In fact, a `case` constant must be a compile-time constant expression whose value is *assignable* to the type of the selector expression (§2.4, p. 46). In particular, all `case` constant values must be in the range of the type of the selector expression. The type of a `case` constant cannot be `boolean`, `long`, or floating-point.

The compiler is able to generate efficient code for a `switch` statement, as this statement only tests for *equality* between the selector expression and the constant expressions of the `case` labels, so as to determine which code to execute at runtime. In contrast, a sequence of `if` statements determines the flow of control at runtime, based on arbitrary conditions which might be determinable only at runtime.

In **Example 4.1**, depending on the value of the `howMuchAdvice` parameter, different advice is printed in the `switch` statement at (1) in the method `dispenseAdvice()`. The example shows the output when the value of the `howMuchAdvice` parameter is `LOTS_OF_ADVICE`. In the `switch` statement, the associated statement at (2) is executed, giving one piece of advice. Control then falls through to the statement at (3), giving the second piece of advice. Control next falls through to (4), dispensing the third piece of advice, and finally execution of the `break` statement at (5) causes control to exit the `switch` statement. Without the `break` statement at (5), control would continue to fall through the remaining statements—in this case, to the statement at (6) being executed. Execution of the `break` statement in a `switch` block transfers control out of the `switch` statement (p. 180). If the parameter `howMuchAdvice` has the value `MORE_ADVICE`, then the advice at both (3) and (4) is given. The value `LITTLE_ADVICE` results in only one piece of advice at (4) being given. Any other value results in the `default` action, which announces that there is no advice.

The associated statement of a `case` label can be a *group* of statements (which need *not* be a statement block). The `case` label is prefixed to the first statement in each `case`. This is illustrated by the associated statements for the `case` constant `LITTLE_ADVICE` in **Example 4.1**, which comprises statements (4) and (5).

Example 4.1 *Fall-Through in a* `switch` *Statement with the Colon Notation*

Click here to view code image

```
public class Advice {

  private static final int LITTLE_ADVICE = 0;
  private static final int MORE_ADVICE = 1;
  private static final int LOTS_OF_ADVICE = 2;

  public static void main(String[] args) {
    dispenseAdvice(LOTS_OF_ADVICE);
  }

  public static void dispenseAdvice(int howMuchAdvice) {
    switch (howMuchAdvice) {                                 // (1)
      case LOTS_OF_ADVICE: System.out.println("See no evil.");   // (2)
      case MORE_ADVICE:    System.out.println("Speak no evil."); // (3)
      case LITTLE_ADVICE:  System.out.println("Hear no evil.");  // (4)
                           break;                            // (5)
      default:             System.out.println("No advice.");     // (6)
    }
  }
}
```

Output from the program:

```
See no evil.
Speak no evil.
Hear no evil.
```

Several `case` labels can prefix the same group of statements. This is the equivalent of specifying the same `case` constants in a single `case` label. The latter syntax is preferable as it is more concise than the former. Such `case` constants will result in the associated group of statements being executed. This behavior is illustrated in Example 4.2 for the `switch` statement at (1).

At (2) in Example 4.2, three `case` labels are defined that are associated with the same action. At (3), (4), and (5), a list of `case` constants is defined for some of the `case` labels. Note also the use of the `break` statement to stop fall-through in the `switch` block after the statements associated with a `case` label are executed.

The first statement in the `switch` block must always have a `case` or `default` label; otherwise, it will be unreachable. This statement will never be executed because control can never be transferred to it. The compiler will flag this case (no pun intended) as an error. An empty `switch` block is perfectly legal, but not of much use.

Since each group of statements associated with a `case` label can be any arbitrary statement, it can also be another `switch` statement. In other words, `switch` statements can be nested. Since a `switch` statement defines its own local block, the `case` labels in an inner block do not conflict with any `case` labels in an outer block. Labels can be redefined in nested blocks; in contrast, variables cannot be redeclared in nested blocks (§6.6, p. 354). In **Example 4.2**, an inner `switch` statement is defined at (6), which allows further refinement of the action to take on the value of the selector expression in cases where multiple `case` labels are used in the outer `switch` statement. A `break` statement terminates the innermost `switch` statement in which it is executed.

The print statement at (7) is always executed for the case constants `9`, `10`, and `11`.

Note that the `break` statement is the last statement in the group of statements associated with each `case` label. It is easy to think that the `break` statement is a part of the `switch` statement syntax, but technically it is not.

Example 4.2 *Nested* `switch` *Statements with the Colon Notation*

Click here to view code image

```
public class Seasons {
  public static void main(String[] args) {
    int monthNumber = 11;
    switch(monthNumber) {                                  // (1) Outer
      case 12: case 1: case 2:                             // (2)
        System.out.println("Snow in the winter.");
        break;
      case 3, 4: case 5:                                   // (3)
        System.out.println("Green grass in the spring.");
        break;
      case 6, 7, 8:                                        // (4)
        System.out.println("Sunshine in the summer.");
        break;
      case 9, 10, 11:                                      // (5)
        switch(monthNumber) { // Nested switch                (6) Inner
          case 10:
            System.out.println("Halloween.");
            break;
          case 11:
            System.out.println("Thanksgiving.");
            break;
        } // End nested switch
        // Always printed for case constant 9, 10, 11
        System.out.println("Yellow leaves in the fall.");  // (7)
```

```
            break;
          default:
            System.out.println(monthNumber + " is not a valid month.");
        }
      }
    }
```

Output from the program:

[Click here to view code image](#)

```
Thanksgiving.
Yellow leaves in the fall.
```

The `switch` Statement with the Arrow (`->`) Notation

The form of the `switch` statement with the arrow notation is shown in **Figure 4.4**. This form defines *switch rules* in which each `case` label is associated with a corresponding action using the arrow (`->`) notation.

Figure 4.4 *Form of the* `switch` *Statement with the Arrow Notation*

```
switch (selector_expression) {
  // Switch block with switch rules defined using arrow notation:
  case CC                    -> expression_statement;
  case CC1, CC2, ..., CCm    -> block
  case CC4                   -> throw_statement
  ...
  default                    -> ...
}
```


```
switch (selector_expression) {
  // Switch block with switch rules defined using arrow notation:
  case CC                    -> expression_statement;
  case CC1, CC2, ..., CCm    -> block
  case CC4                   -> throw_statement
  ...
  default                    -> ...
}
```


Compared to the `switch` statement with the colon notation (**Figure 4.2**), there are a few things to note.

First, although the `case` labels (and the `default` label) are specified similarly, the arrow notation does not allow multiple `case` labels to be associated with a common action. However, the same result can be achieved by specifying a single `case` label with a list of `case` constants, thereby associating the `case` constants with a common action.

Second, the action that can be associated with the `case` labels in switch rules is restricted. The `switch` statement with the colon notation allows a group of statements, but the `switch` statement with the arrow notation only allows the following actions to be associated with `case` labels:

- *An expression statement (§3.3, p. 101)*
  By far, the canonical action of a `case` label in a switch rule is an expression statement. Such an expression statement is always terminated by a semicolon (`;`). Typically, the value returned by the expression statement is discarded. In the examples below, what is important is the side effect of evaluating the expression statements.
  Click here to view code image

```
...
case PASSED -> ++numbersPassed;
case FAILED -> ++numbersFailed;
...
```

- *A block (§6.6, p. 354)*
  A block of statements can be used if program logic should be refined.
  Click here to view code image

```
...
case ALARM ->  { soundTheAlarm();
                 callTheFireDepartment(); }
...
```

- *Throw an exception (§7.4, p. 386)*
  The switch rule below throws an exception when the value of the selector expression does not match any `case` constants:
  Click here to view code image

```
...
default -> throw new IllegalArgumentException("Not a valid value");
...
```

Third, the execution of the switch rules is *mutually exclusive* (Figure 4.5). Once the action in the switch rule has completed execution, the execution of the `switch` statement terminates. This is illustrated in Figure 4.5 where only one expression statement is executed, after which the `switch` statement also terminates. There is no fall-through and the `break` statement is not necessary.

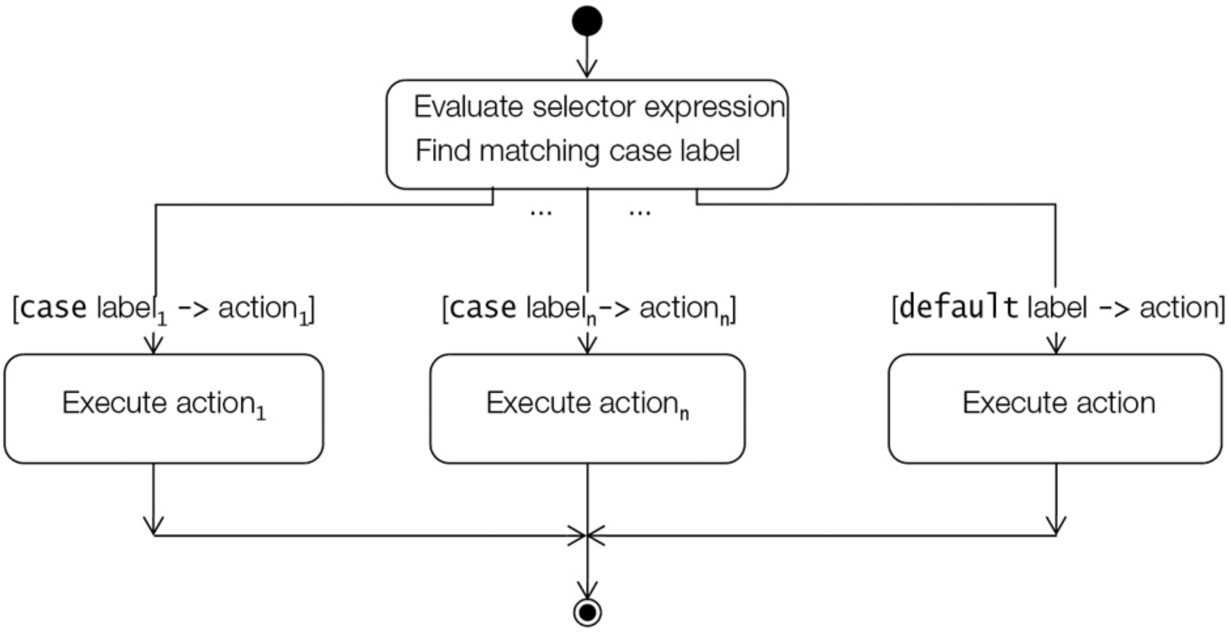

Figure 4.5 *Activity Diagram for the* `switch` *Statement with the Arrow Notation*

Example 4.3 is a refactoring of **Example 4.2** with a `switch` statement with the arrow notation. At (2), (3), (4), and (8), the action executed is an expression statement, whereas at (5), the action executed is a block. Using switch rules results in compact and elegant code that also improves the readability of the `switch` statement.

Example 4.3 *Nested* `switch` *Statements with the Arrow Notation*

Click here to view code image

```
public class SeasonsII {
  public static void main(String[] args) {
    int monthNumber = 11;
    switch(monthNumber) {                                             // (1) Outer
      case 12, 1,  2 -> System.out.println("Snow in the winter.");    // (2)
      case 3,  4,  5 -> System.out.println("Green grass in the spring.");   // (3)
      case 6,  7,  8 -> System.out.println("Sunshine in the summer.");      // (4)
      case 9, 10, 11 -> {                                             // (5)
        switch(monthNumber) { // Nested switch                           (6) Inner
          case 10 -> System.out.println("Halloween.");
          case 11 -> System.out.println("Thanksgiving.");
        }
        // Always printed for case constants 9, 10, 11:
        System.out.println("Yellow leaves in the fall.");             // (7)
      }
      default -> throw new IllegalArgumentException(monthNumber +
                                    " is not a valid month.");// (8)
    }
```

```
  }
}
```

Output from the program:

Click here to view code image

```
Thanksgiving.
Yellow leaves in the fall.
```

Using Strings as `case` Constants

Example 4.4 illustrates using strings in a `switch` statement. The thing to note is what constitutes a constant string expression that can be used as a `case` constant. The `case` constants at (3), (4), (5), and (6) are all valid *constant string expressions*, as the compiler can figure out their value at compile time. String literals, used at (3) and (6), and constant field values, declared at (1) and (2a) and used at (4) and (5), are all valid `case` constants. In contrast, the `HOT` reference from declarations (2b) and (2c) cannot be used as a `case` constant. From the declaration at (2a), the compiler cannot guarantee that the value of the reference will not change at runtime. From the declaration at (2c), it cannot deduce the value at compile time, as the constructor must be run to construct the value.

Switching on strings is essentially based on equality comparison of integer values that are hash values of strings, followed by an object equality test to rule out the possibility of collision between two different strings having the same hash value. Switching on strings should be used judiciously, as it is less efficient than switching on integers. Switching on strings is not advisable if the values being switched on are not already strings.

Example 4.4 *Strings in a* `switch` *Statement*

Click here to view code image

```
public class SwitchingOnAString {
  public static final String MEDIUM = "Medium";       // (1)
  public static final String HOT = "Hot";             // (2a)
//public static       String HOT = "Hot";             // (2b) Not OK as case label
//public static final String HOT = new String("Hot"); // (2c) Not OK as case label
  public static void main(String[] args) {
    String spiceLevel = "Medium_Hot";
    switch (spiceLevel) {
      case "Mild",                                                        // (3)
            MEDIUM + "_" + HOT -> System.out.println("Enjoy your meal!"); // (4)
      case HOT                 -> System.out.println("Have fun!");        // (5)
      case "Suicide"           -> System.out.println("Good luck!");       // (6)
      default                  -> System.out.println("You being funny?");
```

```
        }
      }
    }
```

Output from the program:

```
Enjoy your meal!
```

Using Enum Constants as `case` Constants

Example 4.5 illustrates the use of enum types (§5.13, p. 287) in a `switch` statement with the arrow notation. The enum type `SpiceGrade` is defined at (1). The type of the selector expression at (2) is the enum type `SpiceGrade`. Note that the enum constants are *not* specified with their fully qualified name (see (3a)). Using the fully qualified name results in a compile-time error, as shown at (3b). Only enum constants that have the same enum type as the selector expression can be specified as `case` label values.

The semantics of the `switch` statement are the same as described earlier. Switching on enum values is essentially based on equality comparison of unique integer values that are ordinal values assigned by the compiler to the constants of an enum type.

When the switch rules cover *all* values of the selector expression type, the `switch` statement is said to be *exhaustive*. Non-exhaustive `switch` statements are a common cause of programming errors. It is up to the programmer to ensure that the `switch` statement is exhaustive, as the compiler does not provide any help in this regard for the `switch` statement. Judicious use of the `default` label should be considered, as illustrated in the examples provided in this section that use the `switch` statement.

Example 4.5 *Enums in a* `switch` *Statement*

Click here to view code image

```
enum SpiceGrade { MILD, MEDIUM, MEDIUM_HOT, HOT, SUICIDE; }   // (1)

public class SwitchingFun {
  public static void main(String[] args) {
    SpiceGrade spicing = SpiceGrade.HOT;
    switch (spicing) {                                     // (2)
      case HOT ->  System.out.println("Have fun!");       // (3a) OK!
//    case SpiceGrade.HOT                                  // (3b) Compile-time error!
//           -> System.out.println("Have fun!");
      case SUICIDE -> System.out.println("Good luck!");
      default -> System.out.println("Enjoy your meal!");
    }
```

```
    }
  }
```

Output from the program:

```
Have fun!
```

4.3 The `switch` Expression

A `switch` expression evaluates to a value, as opposed to a `switch` statement that does not. Conceptually we can think of a `switch` expression as an augmented `switch` statement that returns a value. We look at both forms of the `switch` expression, defined using the colon notation and the arrow notation, how it yields a value, and compare it to the `switch` statement. The `switch` expression is analogous to the `switch` statement, except for the provision to return a value.

The `yield` Statement

The `yield` statement in a `switch` expression is analogous to the `break` statement in a `switch` statement. It can only be used in a `switch` expression, where the identifier `yield` is a *contextual keyword* only having a special meaning in the context of a `switch` expression.

```
yield expression;
```

Execution of the `yield` statement results in the expression being evaluated, and its value being returned as the value of the `switch` expression.

The `switch` Expression with the Colon (`:`) Notation

The `switch` expression with the colon notation has the same form as the `switch` statement with the colon notation (**Figure 4.2**), except that the execution of the `switch` body results in a value (or it throws an exception).

Example 4.6 is a reworking of **Example 4.2** with seasons, where the group of statements associated with a `case` label print information about the season and return a constant of the enum type `Season` that is defined at (1). Note that the `yield` statement is the last statement in the group of statements associated with each `case` label. Execution of the `yield` statement results in its expression being evaluated, and its value being returned as the value of the `switch` expression, thereby also terminating the execution of the `switch` expression. Not surprisingly, a `break` or a `return` statement is not allowed in a `switch` expression. Note that the `switch` expression is on the right-hand side of the assignment statement defined at (2) and is terminated by a semicolon (`;`).

The *fall-through* of execution in the `switch` expression with the colon notation is analogous to that of the `switch` statement with the colon notation (**Figure 4.3**). If a group of statements associated with a `case` label does not end in a `yield` statement, execution continues with the next group of statements, if any.

The `switch` expression with the colon notation must be *exhaustive*, meaning the `case` labels, and if necessary the `default` label, must cover *all* values of the selector expression type. Non-exhaustive `switch` expressions will result in a compile-time error. The `default` label is typically used to make the `switch` expression exhaustive. In **Example 4.6**, the type of the selector expression is `int`, but the `case` labels only cover the `int` values from 1 to 12. A `default` label is necessary to cover the other `int` values or to throw an exception, as in this case, and make the `switch` expression exhaustive.

Example 4.6 *A* `yield` *Statement in a* `switch` *Expression with the Colon Notation*

Click here to view code image

```
public class SeasonsIII {

  enum Season { WINTER, SPRING, SUMMER, FALL }                // (1)

  public static void main(String[] args) {
    int monthNumber = 11;
    Season season = switch(monthNumber) {                     // (2)
      case 12: case 1: case 2:                                // (3)
        System.out.println("Snow in the winter.");
        yield Season.WINTER;                                  // (4)
      case 3, 4: case 5:                                      // (5)
        System.out.println("Green grass in the spring.");
        yield Season.SPRING;                                  // (6)
      case 6, 7, 8:                                           // (7)
        System.out.println("Sunshine in the summer.");
        yield Season.SUMMER;                                  // (8)
      case 9, 10, 11:                                         // (9)
        System.out.println("Yellow leaves in the fall.");
        yield Season.FALL;                                    // (10)
      default:                                                // (11)
        throw new IllegalArgumentException(monthNumber + " not a valid month.");
    };                                                        // (12)
    System.out.println(season);
  }
}
```

Output from the program:

Click here to view code image

```
Yellow leaves in the fall.
FALL
```

The `switch` Expression with the Arrow (`->`) Notation

The `switch` expression with the arrow notation also has the same form as the `switch` statement with the arrow notation (Figure 4.4), except that the execution of the `switch` body must result in a value (or it must throw an exception).

The execution of the switch rules in a `switch` expression is *mutually exclusive*, analogous to the switch rules in a `switch` statement (Figure 4.5). Once the action in the switch rule has completed execution, the value computed by the action is returned and the execution of the `switch` expression terminates. There is no fall-through and no `break` statement is allowed.

Whereas the actions in the switch rules of a `switch` statement only allowed an *expression statement* (Figure 4.5), the actions in the switch rules of a `switch` expression allow *any expression*, in addition to allowing a block or throwing an exception, as in the switch rules of a `switch` statement.

- *Any expression (Chapter 2, p. 29)*
  By far, the canonical action of a `case` label in a switch rule of a `switch` expression is an arbitrary expression. Such an expression is always terminated by a semicolon (`;`). The expression value is returned as the value of the `switch` expression whose execution is then terminated. Note that no `yield` statement is necessary or allowed.
  Click here to view code image

```
...
case 1 -> "ONE";
case 2 -> yield "two";        // Compile-time error!
...
```

- *A block (§6.6, p. 354)*
  A *block* of statements can be used if program logic should be refined, but the last statement in the block should be a `yield` statement to return its value and terminate the execution of the `switch` expression (alternatively, the last statement can be a `throw` statement). In a `switch` expression with the arrow notation, the `yield` statement is only allowed as the last statement in a block that constitutes the action in a switch rule.

Click here to view code image

```
...
case ALARM     -> { soundTheAlarm();
                    callTheFireDepartment();
                    yield Status.EVACUATE; }  // OK
case ALL_CLEAR -> { yield Status.NORMAL; // Compile-time error: not last statement
                    standDown(); }       //                     in the block.
...
```

- *Throw an exception (§7.4, p. 386)*

  As the switch rules must be *exhaustive*, one way to achieve exhaustiveness is to throw an exception as the action in the `default` label.

  Click here to view code image

```
...
default -> throw new IllegalArgumentException("Not a valid value");
...
```

Example 4.6 has been refactored to use the `switch` expression with the arrow notation in Example 4.7. Each action associated with a `case` label of a switch rule is a block of statements, where a `yield` statement is the last statement in a block. If this is not the case, the code will not compile.

The `switch` expression with the arrow notation must also be *exhaustive*. Again a non-exhaustive `switch` expression with the arrow notation will result in a compile-time error. In Example 4.7, the type of the selector expression is `int`, but the switch rules only cover the `int` values from 1 to 12. A `default` label is necessary to make the `switch` expression exhaustive, as shown at (11).

Example 4.7 *Statement Blocks in a* `switch` *Expression with the Arrow Notation*

Click here to view code image

```
public class SeasonsIV {
  enum Season { WINTER, SPRING, SUMMER, FALL }                 // (1)

  public static void main(String[] args) {
    int monthNumber = 11;
    Season season = switch(monthNumber) {                      // (2)
      case 12, 1, 2 -> {                                       // (3)
        System.out.println("Snow in the winter.");
        yield Season.WINTER;                                   // (4)
      }
      case 3, 4, 5 -> {                                        // (5)
        System.out.println("Green grass in the spring.");
```

```
        yield Season.SPRING;                              // (6)
      }
      case 6, 7, 8 -> {                                   // (7)
        System.out.println("Sunshine in the summer.");
        yield Season.SUMMER;                              // (8)
      }
      case 9, 10, 11 -> {                                 // (9)
        System.out.println("Yellow leaves in the fall.");
        yield Season.FALL;                                // (10)
      }
      default ->                                          // (11)
        throw new IllegalArgumentException(monthNumber + " not a valid month.");
    };                                                    // (12)
    System.out.println(season);
  }
}
```

Output from the program:

Click here to view code image

```
Yellow leaves in the fall.
FALL
```

Example 4.8 is a reworking of Example 4.7 that defines *expressions* as the actions in the switch rules. No `yield` statement is necessary or allowed in this case. The `switch` expression is also exhaustive.

Example 4.8 *Expression Actions in a* `switch` *Expression with the Arrow Notation*

Click here to view code image

```
public class SeasonsV {
  enum Season { WINTER, SPRING, SUMMER, FALL }                 // (1)

  public static void main(String[] args) {
    int monthNumber = 11;
    Season season = switch(monthNumber) {                      // (2)
      case 12,  1,  2 -> Season.WINTER;                        // (3)
      case  3,  4,  5 -> Season.SPRING;                        // (4)
      case  6,  7,  8 -> Season.SUMMER;                        // (5)
      case  9, 10, 11 -> Season.FALL;                          // (6)
      default         -> throw new IllegalArgumentException(monthNumber +
                                                     " not a valid month.");
    };
    System.out.println(season);
```

```
    }
  }
```

Output from the program:

```
  FALL
```

The `switch` expression can only evaluate to a single value. Multiple values can be returned by constructing an object with the required values and returning the object as a result of evaluating the `switch` expression. Record classes are particularly suited for this purpose (§5.14, p. 299). The `switch` expression at (3) in Example 4.9 returns an object of the record class `SeasonInfo`, defined at (2), to return the month number and the season in which it occurs.

Example 4.9 *Returning Multiple Values as a Record from a* `switch` *Expression*

Click here to view code image

```
public class SeasonsVI {

  enum Season { WINTER, SPRING, SUMMER, FALL }                          // (1)
  record SeasonInfo(int month, Season season) {}                       // (2)

  public static void main(String[] args) {
    int monthNumber = 11;
    SeasonInfo seasonInfo = switch(monthNumber) {                       // (3)
      case 12,  1,  2 -> new SeasonInfo(monthNumber, Season.WINTER);    // (4)
      case  3,  4,  5 -> new SeasonInfo(monthNumber, Season.SPRING);    // (5)
      case  6,  7,  8 -> new SeasonInfo(monthNumber, Season.SUMMER);    // (6)
      case  9, 10, 11 -> new SeasonInfo(monthNumber, Season.FALL);      // (7)
      default         -> throw new IllegalArgumentException(monthNumber +
                                                     " not a valid month.");
    };
    System.out.println(seasonInfo);
  }
}
```

Output from the program:

Click here to view code image

```
SeasonInfo[month=11, season=FALL]
```

Local Variable Scope in the `switch` Body

The *scope* of a local variable declared in a `switch` statement or a `switch` expression is the *entire* `switch` block. Any local block in the `switch` body introduces a new *local scope*. Any local variable declared in it has *block scope*, and therefore, is only accessible in that block. A local variable declared in an enclosing local scope cannot be redeclared in a nested local scope (§6.6, p. 354).

Summary of the `switch` Statement and the `switch` Expression

Table 4.1 summarizes the features of the `switch` statement and the `switch` expression, and provides a comparison of the two constructs.

Table 4.1 *Comparing the* `switch` *Statement and the* `switch` *Expression*

Notation	The `switch` statement	The `switch` expression
The colon (`:`) notation: `case` *label* `:` *statements*	• Executes statements associated with the matching `case` label. • Fall-through can occur. • No compile-time check for exhaustiveness. • Only `break` and `return` statements allowed to control fall-through.	• Executes statements associated with the matching `case` label, but must have a `yield` statement to return a value. • Fall-through can occur. • Compile-time check for exhaustiveness. • No `break` or `return` statement allowed.
The arrow (`->`) notation: `case` *label* `->` *action*	• Action associated with a switch rule can be *an expression statement*, can be a *block*, or can *throw an exception*. • Mutually exclusive switch rules: no fall-through can occur. • No compile-time check for exhaustiveness. • `break` and `return` statements allowed.	• Action associated with a switch rule can be *any expression*, can be a *block*, or can *throw an exception*. • Mutually exclusive switch rules: no fall-through can occur. • Compile-time check for exhaustiveness. • No `break` or `return` statement allowed. • Must return a value that is either the value of a stand-alone *expression* or the value of the expression in a `yield` statement that can oc-

cur as the last statement in a *block*.

Review Questions

4.1 What will be the result of attempting to compile and run the following class?

Click here to view code image

```
public class IfTest {
  public static void main(String[] args) {
    if (true)
    if (false)
    System.out.println("a");
    else
    System.out.println("b");
  }
}
```

Select the one correct answer.

a. The code will fail to compile because the syntax of the `if` statement is incorrect.

b. The code will fail to compile because the compiler will not be able to determine which `if` statement the `else` clause belongs to.

c. The code will compile correctly and will display the letter `a` at runtime.

d. The code will compile correctly and will display the letter `b` at runtime.

e. The code will compile correctly but will not display any output.

4.2 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
public class Switching {
  public static void main(String[] args) {
    final int iLoc = 3;
    switch (6) {
      case 1:
      case iLoc:
      case 2 * iLoc:
```

```
        System.out.println("I am not OK.");
      default:
        System.out.println("You are OK.");
      case 4:
        System.out.println("It's OK.");
    }
  }
}
```

Select the one correct answer.

a. The code will fail to compile because of the `case` label value `2 * iLoc`.

b. The code will fail to compile because the `default` label is not specified last in the `switch` statement.

c. The code will compile correctly and will print the following at runtime:

d. `I am not OK.`

e. `You are OK.`

f. `It's OK.`

g. The code will compile correctly and will print the following at runtime:

h. `You are OK.`

i. `It's OK.`

j. The code will compile correctly and will print the following at runtime:

k. `It's OK.`

4.3 Which code option will print the string `"Prime"`?

Select the one correct answer.

Click here to view code image

```
char value = 3;
String result = "Unknown";
switch (value) {
  case 2,3,5,7:   result = "Prime";
  case 1,4,6,8,9: result = "Composite";
```

```
  }
  System.out.println(result);
```

Click here to view code image

```
char value = 3;
String result =
switch (value) {
  case 2,3,5,7:   yield "Prime";
  case 1,4,6,8,9: yield "Composite";
};
System.out.println(result);
```

Click here to view code image

```
char value = 3;
String yield =
switch (value) {
  case 2,3,5,7:   yield "Prime";
  case 1,4,6,8,9: yield "Composite";
  default:        yield "Unknown";
};
System.out.println(yield);
```

Click here to view code image

```
char value = 3;
String result =
switch (value) {
  case 2,3,5,7   -> "Prime";
  case 1,4,6,8,9 -> "Composite";
  default: { yield "Unknown"; }
};
System.out.println(result);
```

4.4 Given the following code:

Click here to view code image

```
public class RQ462 {
  public static void main(String[] args) {
    int price = 1;
    int discount = switch (price) {
      case 5, 1, 2 -> price - 1;
      case 4, 3, 6 -> price - 2;
      default      -> 0;
```

```
    };
    System.out.println(discount);
  }
}
```

What is the result?

Select the one correct answer.

a. `0`

b. `1`

c. `-1`

d. `-2`

e. The program will throw an exception at runtime.

f. The program will fail to compile.

4.4 Iteration Statements

Loops allow a single statement or a statement block to be executed repeatedly (i.e., iterated). A boolean condition (called the *loop condition*) is commonly used to determine when to terminate the loop. The statements executed in the loop constitute the *loop body*.

Java provides four language constructs for loop construction:

- The `while` statement
- The `do-while` statement
- The *basic* `for` statement
- The *enhanced* `for` statement

These loops differ in the order in which they execute the loop body and test the loop condition. The `while` loop and the basic `for` loop test the loop condition *before* executing the loop body, whereas the `do-while` loop tests the loop condition *after* execution of the loop body.

The *enhanced* `for` loop (also called the *for-each* loop) simplifies iterating over arrays and collections. We will use the notations `for(;;)` and `for(:)` to designate the basic `for` loop and the enhanced `for` loop, respectively.

4.5 The `while` Statement

The syntax of the `while` loop is

```
while (loop_condition)
  loop_body
```

The *loop condition* is evaluated before executing the *loop body*. The `while` statement executes the *loop body* as long as the *loop condition* is `true`. When the *loop condition* becomes `false`, the loop is terminated and execution continues with any statement immediately following the loop. If the *loop condition* is `false` to begin with, the *loop body* is not executed at all. In other words, a `while` loop can execute zero or more times. The *loop condition* must evaluate to a `boolean` or a `Boolean` value. In the latter case, the reference value is unboxed to a `boolean` value. The flow of control in a `while` statement is shown in **Figure 4.6**.

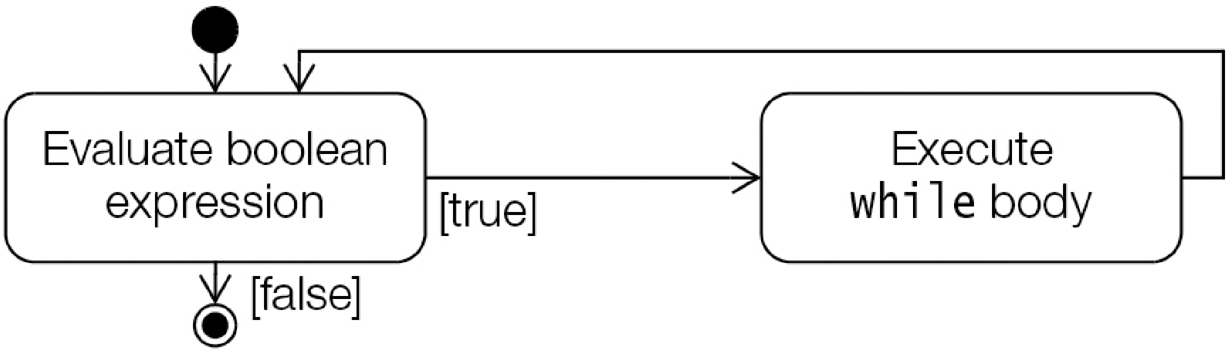

Figure 4.6 *Activity Diagram for the* `while` *Statement*

The `while` statement is normally used when the number of iterations is not known.

```
while (noSignOfLife())
  keepLooking();
```

Since the *loop body* can be any valid statement, inadvertently terminating each line with the empty statement (`;`) can give unintended results. Always using a block statement as the *loop body* helps to avoid such problems.

Click here to view code image

```
while (noSignOfLife());     // Empty statement as loop body!
  keepLooking();            // Statement not in the loop body.
```

4.6 The `do-while` Statement

The syntax of the `do-while` loop is

```
do
  loop_body
while (loop_condition);
```

In a `do-while` statement, the *loop condition* is evaluated *after* executing the *loop body*. The *loop condition* must evaluate to a `boolean` or `Boolean` value. The value of the *loop condition* is subjected to unboxing if it is of the type `Boolean`. The `do-while` statement executes the *loop body* until the *loop condition* becomes `false`. When the *loop condition* becomes `false`, the loop is terminated and execution continues with any statement immediately following the loop. Note that the *loop body* is executed at least once. **Figure 4.7** illustrates the flow of control in a `do-while` statement.

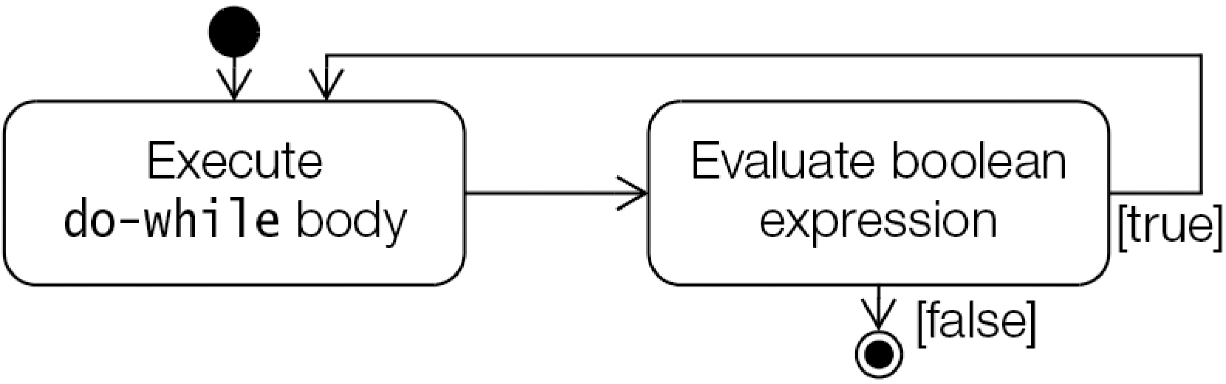

Figure 4.7 *Activity Diagram for the* `do-while` *Statement*

The *loop body* in a `do-while` loop is invariably a statement block. It is instructive to compare the `while` and `do-while` loops. In the examples that follow, the mice might never get to play if the cat is not away, as in the loop at (1). The mice do get to play at least once (at the peril of losing their life) in the loop at (2).

Click here to view code image

```
while (cat.isAway()) {        // (1)
  mice.play();
}
do {                          // (2)
  mice.play();
} while (cat.isAway());
```

4.7 The `for(;;)` Statement

The `for(;;)` loop is the most general of all the loops. It is mostly used for *counter-controlled loops*, in which the number of iterations is known beforehand.

The syntax of the loop is as follows:

Click here to view code image

```
for (initialization; loop_condition; update_expression)
        loop_body
```

The *initialization* usually declares and initializes a *loop variable* that controls the execution of the *loop body*. The loop body can be a single statement or a statement block. The *loop condition* must evaluate to a `boolean` or `Boolean` value. In the latter case, the reference value is converted to a `boolean` value by unboxing. The *loop condition* usually involves the loop variable, and if the loop condition is `true`, the *loop body* is executed; otherwise, execution continues with any statement following the `for(;;)` loop. After each iteration (i.e., execution of the loop body), the *update expression* is executed. This usually modifies the value of the loop variable to ensure eventual loop termination. The *loop condition* is then tested to determine whether the loop body should be executed again. Note that the *initialization* is executed only once, on entry into the loop. The semantics of the `for(;;)` loop are illustrated in **Figure 4.8**, and are summarized by the following equivalent `while` loop code template:

Click here to view code image

```
initialization
while (loop_condition) {
    loop_body
    update_expression
}
```

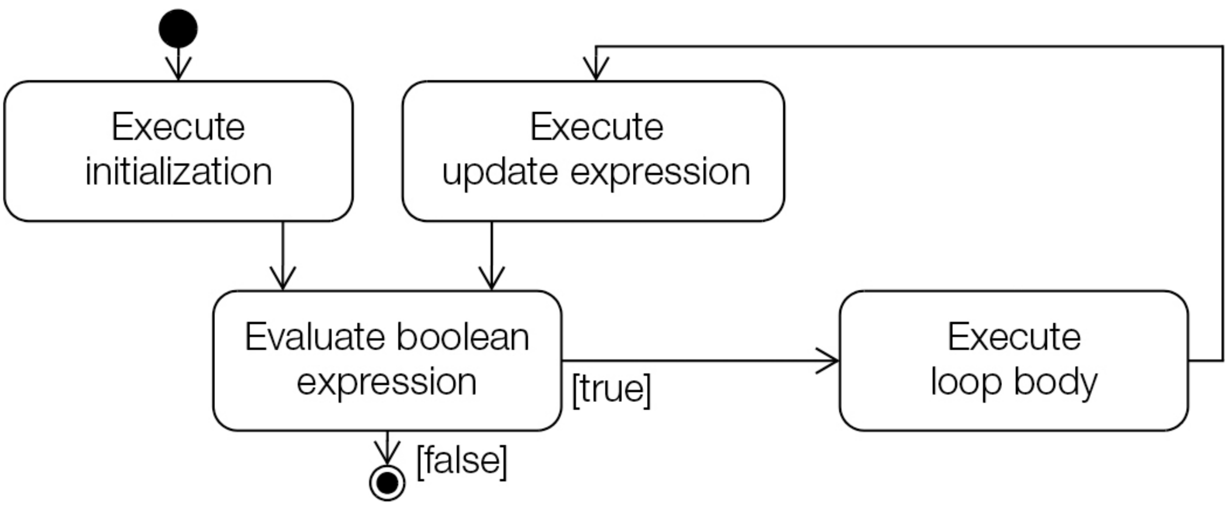

Figure 4.8 *Activity Diagram for the* `for` *Statement*

The following code creates an `int` array and sums the values in the array:

Click here to view code image

```
int sum = 0;
int[] array = {12, 23, 5, 7, 19};
```

```
for (int index = 0; index < array.length; index++)   // (1)
  sum += array[index];
```

The loop variable `index` is declared and initialized in the *initialization* section of the loop. It is incremented in the *update expression* section. This loop is an example of a *forward* `for(;;)` loop, where the loop variable is incremented.

The next code snippet is an example of a *backward* `for(;;)` loop, where the loop variable is decremented to sum the values in the array:

Click here to view code image

```
int sum = 0;
int[] array = {12, 23, 5, 7, 19};
for (int index = array.length - 1; index >= 0; index--)
  sum += array[index];
```

It is instructive to compare the specification of the loop header in the forward and backward `for(;;)` loops in these examples.

The loop at (1) earlier showed how a declaration statement can be specified in the *initialization* section. Such a declaration statement can also specify a comma-separated list of variables:

Click here to view code image

```
for (int i = 0, j = 1, k = 2; ... ; ...) ...;      // (2)
```

The variables `i`, `j`, and `k` in the declaration statement all have type `int`. All variables declared in the *initialization* section are local variables in the `for(;;)` statement and obey the scope rules for local blocks, as do any variables declared in the *loop body*. The following code will not compile, however, as variable declarations of different types (in this case, `int` and `String`) require declaration statements that are terminated by semicolons:

Click here to view code image

```
for (int i = 0, String str = "@"; ... ; ...) ...;  // (3) Compile-time error
```

The *initialization* section can also be a comma-separated list of *expression* statements (§3.3, p. 101). Any value returned by an expression statement is discarded. For example, the loop at (2) can be rewritten by factoring out the variable declarations:

Click here to view code image

```
int i, j, k;                                          // Variable declaration
for (i = 0, j = 1, k = 2; ... ; ...) ...;             // (4) Only initialization
```

The *initialization* section is now a comma-separated list of three expressions. The expressions in such a list are always evaluated from left to right, and their values are discarded. Note that the variables `i`, `j`, and `k` at (4) are not local to the loop.

Declaration statements cannot be mixed with expression statements in the *initialization* section, as is the case at (5) in the following example. Factoring out the variable declaration, as at (6), leaves a legal comma-separated list of expression statements.

Click here to view code image

```
// (5) Not legal and ugly:
for (int i = 0, System.out.println("This won't do!"); flag; i++) { // Error!
  // loop body
}

// (6) Legal, but still ugly:
int i;                                       // Declaration factored out.
for (i = 0, System.out.println("This is legal!"); flag; i++) {     // OK.
  // loop body
}
```

The *update expression* can also be a comma-separated list of expression statements. The following code specifies a `for(;;)` loop that has a comma-separated list of three variables in the *initialization* section, and a comma-separated list of two expressions in the *update expression* section:

Click here to view code image

```
// Legal usage but not recommended, as it can affect code comprehension.
int[][] sqMatrix = { {3, 4, 6}, {5, 7, 4}, {5, 8, 9} };
for (int i = 0, j = sqMatrix[0].length - 1, asymDiagonal = 0;  // initialization
     i < sqMatrix.length;                                      // loop condition
     i++, j--)                                              // update expression
  asymDiagonal += sqMatrix[i][j];                           // loop body
```

All sections in the `for(;;)` header are optional. Any or all of them can be left empty, but the two semicolons are mandatory. In particular, leaving out the *loop condition* signifies that the loop condition is `true`. The “crab”, `(;;)`, can be used to construct an infinite loop, where termination is presumably achieved through code in the loop body (see the next section on transfer statements):

Click here to view code image

```
for (;;) doProgramming();       // Infinite loop
```

4.8 The `for(:)` Statement

The enhanced `for` loop is convenient when we need to iterate over an array or a collection, especially when some operation needs to be performed on each element of the array or collection. In this section we discuss iterating over arrays. In §15.2, p. 795, we take a look at the `for(:)` loop for iterating over collections.

Earlier in this chapter we used a `for(;;)` loop to sum the values of elements in an `int` array:

Click here to view code image

```
int sum = 0;
int[] intArray = {12, 23, 5, 7, 19};
for (int index = 0; index < intArray.length; index++) { // (1) using for(;;) loop
  sum += intArray[index];
}
```

The `for(;;)` loop at (1) is rewritten using the `for(:)` loop in Figure 4.9.

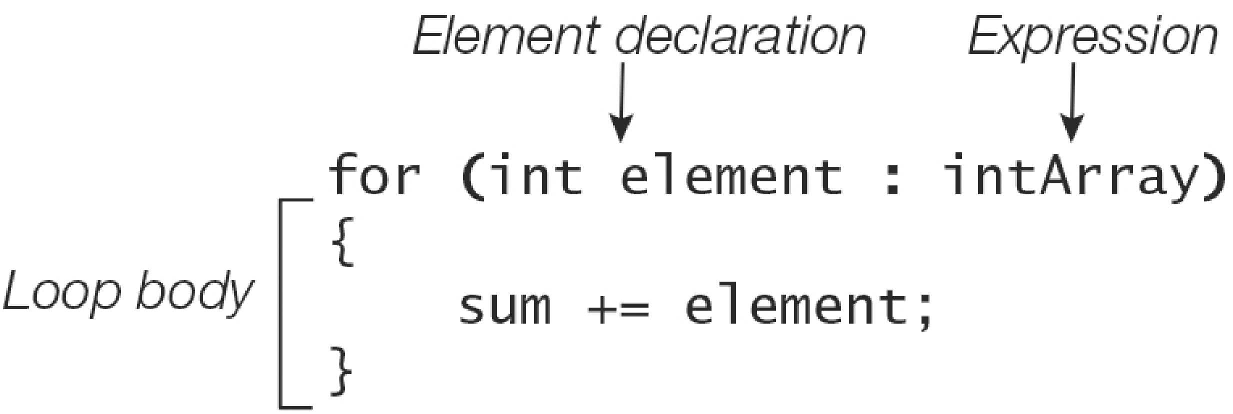

Figure 4.9 *Enhanced* `for` *Statement*

The body of the loop is executed for each element in the array, where the variable `element` successively denotes the current element in the array `intArray`. When the loop terminates, the variable `sum` will contain the sum of all elements in the array. We do not care about the *position* of the elements in the array, just that the loop iterates over *all* elements of the array.

From Figure 4.9 we see that the `for(:)` loop header has two parts. The *expression* must evaluate to a reference value that refers to an *array* (or an object which implements the `Iterable<E>` interface (§15.2, p. 791)). The array can be an array of primitive values or objects, or even an array of arrays. The *expression* is evaluated only once. The *element declaration* specifies a local variable that can be assigned a value of the element type of the array. The type of the array `intArray` in **Figure 4.9** is `int[]`, and the element type is `int`. The element variable of type `int` can be assigned `int` values from the array of `int`. However, this assignment might require either a boxing or an unboxing conversion, with optional widening conversion.

The element variable is local to the loop block and is not accessible after the loop terminates. Also, changing the value of the current variable does *not* change any value in the array. The *loop body*, which can be a simple statement or a statement block, is executed for each element in the array and there is no danger of any out-of-bounds errors.

The `for(:)` loop has its limitations. Specifically, we cannot change element values, and this kind of loop does not provide any provision for positional access using an index. The `for(:)` loop only increments by one and always in a forward direction. It does not allow iterations over several arrays simultaneously. Under such circumstances, the `for(;;)` loop can be more convenient.

Here are some code examples of legal `for(:)` loops:

Click here to view code image

```
// Some one-dim arrays:
int[]     intArray =    {10, 20, 30};
Integer[] intObjArray = {10, 20, 30};
String[]  strArray =    {"one", "two"};

// Some two-dim arrays:
Object[][] objArrayOfArrays = {intObjArray, strArray};
Number[][] numArrayOfArrays = {{1.5, 2.5}, intObjArray, {100L, 200L}};
int[][]    intArrayOfArrays = {{20}, intArray, {40}};

// Iterate over a String array.
// Expression type is String[], and element type is String.
// String is assignable to Object (widening conversion).
for (Object obj : strArray) {}

// Iterate over an int array.
// Expression type is int[], and element type is int.
// int is assignable to Integer (boxing conversion)
for (Integer iRef : intArrayOfArrays[0]){}

// Iterate over an Integer array.
// Expression type is Integer[], and element type is Integer.
// Integer is assignable to int (unboxing conversion)
for (int i : intObjArray){}

// Iterate over a two-dim int array.
// Outer loop: expression type is int[][], and element type is int[].
```

```
// Inner loop: expression type is int[], and element type is int.
for (int[] row : intArrayOfArrays)
  for (int val : row) {}

// Iterate over a two-dim Number array.
// Outer loop: expression type is Number[][], and element type is Number[].
// Outer loop: Number[] is assignable to Object[] (widening conversion).
// Inner loop: expression type is Object[], and element type is Object.
for (Object[] row : numArrayOfArrays)
  for (Object obj : row) {}

// Outer loop: expression type is Integer[][], and element type is Integer[].
// Outer loop: Integer[] is assignable to Number[].
// Inner loop: expression type is int[], and element type is int.
// Inner loop: int is assignable to double.
for (Number[] row : new Integer[][] {intObjArray, intObjArray, intObjArray})
  for (double num : intArray) {}
```

Here are some code examples of `for(:)` loops that are not legal:

Click here to view code image

```
// Expression type is Number[][], and element type is Number[].
// Number[] is not assignable to Number.
for (Number num : numArrayOfArrays) {}       // Compile-time error!

// Expression type is Number[], and element type is Number.
// Number is not assignable to int.
for (int row : numArrayOfArrays[0]) {}       // Compile-time error!

// Outer loop: expression type is int[][], and element type is int[].
// int[] is not assignable to Integer[].
for (Integer[] row : intArrayOfArrays)       // Compile-time error!
  for (int val : row) {}

// Expression type is Object[][], and element type is Object[].
// Object[] is not assignable to Integer[].
for (Integer[] row : objArrayOfArrays) {}    // Compile-time error!

// Outer loop: expression type is String[], and element type is String.
// Inner loop: expression type is String, which is not legal here. Not an array.
for (String str : strArray)
  for (char val : str) {}                    // Compile-time error!
```

When using the `for(:)` loop to iterate over an array, the two main causes of errors are an expression in the loop header that does not represent an array and/or an element type of the array that is not assignable to the local variable declared in the loop header.

4.9 Transfer Statements

Java provides the following language constructs for transferring control in a program:

- The `yield` statement (p. 164)
- The `break` statement
- The `continue` statement
- The `return` statement

The `throw` statement can also transfer control in a program (§7.4, p. 386).

4.10 Labeled Statements

A statement may have a *label*:

```
label : statement
```

A label is any valid identifier; it always immediately precedes the statement. Label names exist in their own namespace, so that they do not conflict with names of packages, classes, interfaces, methods, fields, and local variables. The scope of a label is the statement prefixed by the label, meaning that it cannot be redeclared as a label inside the labeled statement—analogous to the scope of local variables.

Click here to view code image

```
L1: if (i > 0) {
  L1: System.out.println(i);    // (1) Not OK. Label L1 redeclared.
}

L1: while (i < 0) {             // (2) OK.
  L2: System.out.println(i);
}

L1: {                           // (3) OK. Labeled block.
  int j = 10;
  System.out.println(j);
}

L1: try {                       // (4) OK. Labeled try-catch-finally block.
  int j = 10, k = 0;
  L2: System.out.println(j/k);
} catch (ArithmeticException ae) {
  L3: ae.printStackTrace();
} finally {
```

```
    L4: System.out.println("Finally done.");
  }
```

A statement can have multiple labels:

Click here to view code image

```
LabelA: LabelB: System.out.println("Multiple labels. Use judiciously.");
```

A declaration statement cannot have a label:

Click here to view code image

```
L0: int i = 0;                  // Compile-time error!
```

A labeled statement is executed as if it were unlabeled, unless it is the `break` or `continue` statement. This behavior is discussed in the next two subsections.

4.11 The `break` Statement

The `break` statement comes in two forms: *unlabeled* and *labeled*.

Click here to view code image

```
break;              // the unlabeled form
break label;         // the labeled form
```

The unlabeled `break` statement terminates loops (`for(;;)` , `for(:)` , `while` , `do-while`) and `switch` statements, and transfers control out of the current context (i.e., the closest enclosing block). The rest of the statement body is skipped, and execution continues after the enclosing statement.

In **Example 4.10**, the `break` statement at (1) is used to terminate a `for(;;)` loop. Control is transferred to (2) when the value of `i` is equal to `4` at (1), skipping the rest of the loop body and terminating the loop.

Example 4.10 also shows that the unlabeled `break` statement terminates only the innermost loop or `switch` statement that contains the `break` statement. The `break` statement at (3) terminates the inner `for(;;)` loop when `j` is equal to `2`, and execution continues in the outer `switch` statement at (4) after the `for(;;)` loop.

Example 4.10 *The* `break` *Statement*

Click here to view code image

```
class BreakOut {

  public static void main(String[] args) {
    System.out.println("i    sqrt(i)");
    for (int i = 1; i <= 5; ++i) {
      if (i == 4)
        break;                              // (1) Terminate loop. Control to (2).
      // Rest of loop body skipped when i gets the value 4.
      System.out.printf("%d    %.2f%n", i, Math.sqrt(i));
    } // end for
    // (2) Continue here.
    int n = 2;
    switch (n) {
      case 1:
        System.out.println(n);
        break;
      case 2:
        System.out.println("Inner for(;;) loop: ");
        for (int j = 0; j <= n; j++) {
          if (j == 2)
            break;                          // (3) Terminate loop. Control to (4).
          System.out.println("j = " + j);
        }
      default:
        System.out.println("default: n = " + n); // (4) Continue here.
    }
  }
}
```

Output from the program:

Click here to view code image

```
i    sqrt(i)
1    1.00
2    1.41
3    1.73
Inner for(;;) loop:
j = 0
j = 1
default: n = 2
```

A labeled `break` statement can be used to terminate *any* labeled statement that contains the `break` statement. Control is then transferred to the statement following the enclosing labeled statement. In the case of a labeled block, the rest of the block is skipped and execution continues with the statement following the block:

Click here to view code image

```
out:                         // Label.
{                            // (1) Labeled block.
  // ...
  if (j == 10) break out;    // (2) Terminate block. Control to (3).
  System.out.println(j);     // Rest of the block not executed if j == 10.
  // ...
}
// (3) Continue here.
```

In **Example 4.11**, the program continues to add the elements below the diagonal of a square matrix until the sum is greater than 10. Two nested `for` loops are defined at (1) and (2). The outer loop is labeled `outer` at (1). The unlabeled `break` statement at (3) transfers control to (5) when it is executed; that is, it terminates the inner loop and control is transferred to the statement after the inner loop. The labeled `break` statement at (4) transfers control to (6) when it is executed; that is, it terminates both the inner and outer loops, transferring control to the statement after the loop labeled `outer`.

Example 4.11 *Labeled* `break` *Statement*

Click here to view code image

```
class LabeledBreakOut {
  public static void main(String[] args) {
    int[][] squareMatrix = {{4, 3, 5}, {2, 1, 6}, {9, 7, 8}};
    int sum = 0;
    outer: for (int i = 0; i < squareMatrix.length; ++i){   // (1) label
        for (int j = 0; j < squareMatrix[i].length; ++j) {  // (2)
          if (j == i) break;         // (3) Terminate inner loop. Control to (5).
          System.out.println("Element[" + i + ", " + j + "]: " +
                                squareMatrix[i][j]);
          sum += squareMatrix[i][j];
          if (sum > 10) break outer; // (4) Terminate both loops. Control to (6).
        } // end inner loop
        // (5) Continue with update expression in the outer loop header.
      } // end outer loop
    // (6) Continue here.
    System.out.println("sum: " + sum);
  }
}
```

Output from the program:

```
Element[1, 0]: 2
Element[2, 0]: 9
sum: 11
```

4.12 The `continue` Statement

Like the `break` statement, the `continue` statement comes in two forms: *unlabeled* and *labeled*.

Click here to view code image

```
continue;              // the unlabeled form
continue label;         // the labeled form
```

The `continue` statement can be used only in a `for(;;)`, `for(:)`, `while`, or `do-while` loop to prematurely stop the current iteration of the loop body and proceed with the next iteration, if possible. In the case of the `while` and `do-while` loops, the rest of the loop body is skipped—that is, the current iteration is stopped, with execution continuing with the *loop condition*. In the case of the `for(;;)` loop, the rest of the loop body is skipped, with execution continuing with the *update expression*.

In Example 4.12, an unlabeled `continue` statement is used to skip an iteration in a `for(;;)` loop. Control is transferred to (2) when the value of `i` is equal to `4` at (1), skipping the rest of the loop body and continuing with the *update expression* in the `for(;;)` statement.

Example 4.12 *The* `continue` *Statement*

Click here to view code image

```
class Skip {
  public static void main(String[] args) {
    System.out.println("i    sqrt(i)");
    for (int i = 1; i <= 5; ++i) {
      if (i == 4) continue;             // (1) Control to (2).
      // Rest of loop body skipped when i has the value 4.
      System.out.printf("%d    %.2f%n", i, Math.sqrt(i));
      // (2) Continue with update expression in the loop header.
    } // end for
  }
}
```

Output from the program:

```
i    sqrt(i)
1    1.00
2    1.41
3    1.73
5    2.24
```

A labeled `continue` statement must occur within a labeled loop that has the same label. Execution of the labeled `continue` statement then transfers control to the end of that enclosing labeled loop. In **Example 4.13**, the unlabeled `continue` statement at (3) transfers control to (5) when it is executed; that is, the rest of the loop body is skipped and execution continues with the update expression in the inner loop. The labeled `continue` statement at (4) transfers control to (6) when it is executed; that is, it terminates the inner loop but execution continues with the update expression in the loop labeled `outer`. It is instructive to compare the output from **Example 4.11** (labeled `break`) with that from **Example 4.13** (labeled `continue`).

Example 4.13 *Labeled* `continue` *Statement*

Click here to view code image

```
class LabeledSkip {
  public static void main(String[] args) {
    int[][] squareMatrix = {{4, 3, 5}, {2, 1, 6}, {9, 7, 8}};
    int sum = 0;
    outer: for (int i = 0; i < squareMatrix.length; ++i){   // (1) label
        for (int j = 0; j < squareMatrix[i].length; ++j) {  // (2)
          if (j == i) continue;                             // (3) Control to (5).
          System.out.println("Element[" + i + ", " + j + "]: " +
              squareMatrix[i][j]);
          sum += squareMatrix[i][j];
          if (sum > 10) continue outer;                     // (4) Control to (6).
          // (5) Continue with update expression in the inner loop header.
        } // end inner loop
        // (6) Continue with update expression in the outer loop header.
      } // end outer loop
    System.out.println("sum: " + sum);
  }
}
```

Output from the program:

```
Element[0, 1]: 3
Element[0, 2]: 5
Element[1, 0]: 2
Element[1, 2]: 6
```

```
Element[2, 0]: 9
sum: 25
```

4.13 The `return` Statement

The `return` statement is used to stop execution of a method (or a constructor) and transfer control back to the calling code (also called the *caller* or *invoker*). The usage of the two forms of the `return` statement is dictated by whether that statement is used in a `void` or a non-`void` method (**Table 4.2**). The first form does not return any value to the calling code, but the second form does. Note that the keyword `void` does not represent any type.

In **Table 4.2**, the *expression* must evaluate to a primitive value or a reference value, and its type must be *assignable* to the *return type* specified in the method header (§2.7, **p. 54**, and §5.9, **p. 261**). See also the discussion on covariant return in connection with method overriding in §5.1, **p. 201**.

As can be seen from **Table 4.2**, a `void` method need not have a `return` statement—in which case the control typically returns to the caller after the last statement in the method body has been executed. However, a `void` method can specify only the first form of the `return` statement. This form of the `return` statement can also be used in constructors, as they likewise do not return a value.

Table 4.2 also shows that the first form of the `return` statement is not allowed in a non-`void` method. The second form of the return statement is mandatory in a non-`void` method, if the method execution is not terminated programmatically—for example, by throwing an exception. **Example 4.14** illustrates the use of the `return` statement summarized in **Table 4.2**. A recommended best practice is to document the value returned by a method in a Javadoc comment using the `@return` tag.

Table 4.2 *The* `return` *Statement*

Form of `return` statement	In `void` method or in constructor	In non-`void` method
`return;`	Optional	Not allowed
`return` *expression* `;`	Not allowed	Mandatory, if the method is not terminated explicitly

Example 4.14 *The* `return` *Statement*

Click here to view code image

```
public class ReturnDemo {

  public static void main (String[] args) { // (1) void method can use return.
    if (args.length == 0) return;
    output(checkValue(args.length));
  }

  static void output(int value) {  // (2) void method need not use return.
    System.out.println(value);
    return 'a';                    // Not OK. Cannot return a value.
  }

  static int checkValue(int i) {   // (3) Non-void method: Any return statement
                                   //     must return a value.
    if (i > 3)
      return i;                    // OK.
    else
      return 2.0;                  // Not OK. double not assignable to int.
  }

  static int absentMinded() {      // (4) Non-void method.
    throw new RuntimeException();  // OK: No return statement provided, but
                                   // method terminates by throwing an exception.
  }
}
```

Review Questions

4.5 What will be the result of attempting to compile and run the following code?

Click here to view code image

```
class MyClass {
  public static void main(String[] args) {
    boolean b = false;
    int i = 1;
    do {
      i++;
      b = ! b;
    } while (b);
    System.out.println(i);
  }
}
```

Select the one correct answer.

a. The code will fail to compile because `b` is an invalid condition for the `do-while` statement.

b. The code will fail to compile because the assignment `b = ! b` is not allowed.

c. The code will compile without error and will print `1` at runtime.

d. The code will compile without error and will print `2` at runtime.

e. The code will compile without error and will print `3` at runtime.

4.6 What will be the output when running the following program?

Click here to view code image

```
public class StillMyClass {
  public static void main(String[] args) {
    int i = 0;
    int j;
    for (j = 0; j < 10; ++j) { i++; }
    System.out.println(i + " " + j);
  }
}
```

Select the two correct answers.

a. The first number printed will be `9`.

b. The first number printed will be `10`.

c. The first number printed will be `11`.

d. The second number printed will be `9`.

e. The second number printed will be `10`.

f. The second number printed will be `11`.

4.7 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
class AnotherClass {
  public static void main(String[] args) {
```

```
    int i = 0;
    for (; i < 10; i++) ;        // (1)
    for (i = 0;; i++) break;     // (2)
    for (i = 0; i < 10;) i++;    // (3)
    for (;;) ;                   // (4)
  }
}
```

Select the one correct answer.

a. The code will fail to compile because of errors in the `for` loop at (1).

b. The code will fail to compile because of errors in the `for` loop at (2).

c. The code will fail to compile because of errors in the `for` loop at (3).

d. The code will fail to compile because of errors in the `for` loop at (4).

e. The code will compile without error, and the program will run and terminate without any output.

f. The code will compile without error, but will never terminate when run.

4.8 Given the following code fragment, which of the following lines will be a part of the output?

Click here to view code image

```
outer:
for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 2; j++) {
    if (i == j) {
      continue outer;
    }
    System.out.println("i=" + i + ", j=" + j);
  }
}
```

Select the two correct answers.

a. `i=1, j=0`

b. `i=0, j=1`

c. `i=1, j=2`

d. `i=2, j=1`

e. `i=2, j=2`

f. `i=3, j=3`

g. `i=3, j=2`

4.9 Which declarations, when inserted at (1), will result in the program compiling and printing `90` at runtime?

Click here to view code image

```
public class RQ400A10 {
  public static void main(String[] args) {
    // (1) INSERT DECLARATION HERE
    int sum = 0;
    for (int i : nums)
      sum += i;
    System.out.println(sum);
  }
}
```

Select the two correct answers.

a. `Object[] nums = {20, 30, 40};`

b. `Number[] nums = {20, 30, 40};`

c. `Integer[] nums = {20, 30, 40};`

d. `int[] nums = {20, 30, 40};`

e. None of the above

4.10 Which method declarations, when inserted at (1), will result in the program compiling and printing `90` when run?

Click here to view code image

```
public class RQ400A30 {
  public static void main(String[] args) {
    doIt();
  }
  // (1) INSERT METHOD DECLARATION HERE
}
```

Select the two correct answers.

a.

Click here to view code image

```
public static void doIt() {
  int[] nums = {20, 30, 40};
  for (int sum = 0, i : nums)
    sum += i;
  System.out.println(sum);
}
```

b.

Click here to view code image

```
public static void doIt() {
  for (int sum = 0, i : {20, 30, 40})
    sum += i;
  System.out.println(sum);
}
```

c.

Click here to view code image

```
public static void doIt() {
  int sum = 0;
  for (int i : {20, 30, 40})
    sum += i;
  System.out.println(sum);
}
```

d.

Click here to view code image

```
public static void doIt() {
  int sum = 0;
  for (int i : new int[] {20, 30, 40})
    sum += i;
  System.out.println(sum);
}
```

e.

[Click here to view code image]

```
public static void doIt() {
  int[] nums = {20, 30, 40};
  int sum = 0;
  for (int i : nums)
    sum += i;
  System.out.println(sum);
}
```

4.11 Which of the following statements are true about the following `for(:)` loop?

[Click here to view code image]

```
for (type variable : expression) statement
```

Select the three correct answers.

a. The *variable* is only accessible in the `for(:)` loop body.

b. The *expression* is only evaluated once.

c. The type of the expression must be `java.lang.Iterable<E>` or an array type.

d. Changing the value of the *variable* in the loop body affects the data structure represented by the *expression*.

e. The loop runs backward if the *expression* is negated as follows: `!` *expression*.

f. We can iterate over several data structures simultaneously in a `for(:)` loop.