5 Flow control

Exam objectives covered in this chapter	What you need to know
[3.4] Create `if` and `if-else` constructs.	How to use `if`, `if-else`, `if-else-if-else`, and nested `if` constructs. The differences between using these `if` constructs with and without curly braces `{}`.
[3.5] Use a `switch` statement.	How to use a `switch` statement by passing the correct type of arguments to the `switch` statement and `case` and `default` labels. The change in the code flow when `break` and `return` statements are used in the `switch` statement.
[5.1] Create and use `while` loops.	How to use the `while` loop, including determining when to apply the `while` loop.
[5.2] Create and use `for` loops, including the enhanced `for` loop.	How to use `for` and enhanced `for` loops. The advantages and disadvantages of the `for` loop and enhanced `for` loop. Scenarios when you may not be able to use the enhanced `for` loop.
[5.3] Create and use `do-while` loops.	Creation and use of `do-while` loops. Every `do-while` loop executes at least once, even if its condition evaluates to `false` for the first iteration.
[5.4] Compare loop constructs.	The differences and similarities between `for`, enhanced `for`, `do-while`, and `while` loops. Given a scenario or a code snippet, knowing which is the most appropriate loop.

Exam objectives covered in this chapter	What you need to know
[5.5] Use `break` and `continue` statements.	The use of `break` and `continue` statements. A `break` statement can be used within loops and `switch` statement. A `continue` statement can be used only within loops. The difference in the code flow when a `break` or `continue` statement is used. Identify the right scenarios for using `break` and `continue` statements.

We all make multiple decisions on a daily basis, and we often have to choose from a number of available options to make each decision. These decisions range from the complex, such as selecting what subjects to study in school or which profession to choose, to the simple, such as what food to eat or what clothes to wear. The option you choose that leads to a decision can potentially change the course of your life, in a small or big way. For example, if you choose to study medicine at a university, you have the option to become a research scientist; if you choose fine arts, you have the option to become a painter. But deciding whether to eat pasta or pizza for dinner isn't likely to have a huge impact on your life.

You may also repeat particular sets of actions. These actions can range from eating an ice cream cone every day to phoning a friend until you connect, or passing exams at school or university in order to achieve a desired degree. These repetitions can also change the course of your life: you might relish having ice cream everyday, or enjoy the benefits that come from earning a higher degree.

In Java, the selection statements (`if` and `switch`) and looping statements (`for`, enhanced `for`, `while`, and `do-while`) are used to define and choose among different courses of action, as well as to repeat lines of code. You use these types of statements to define the flow of control in code.

In the OCA Java SE 7 Programmer I exam, you'll be asked how to define and control the flow in your code. To prepare you, I'll cover the following topics in this chapter:

- Creating and using `if`, `if-else`, and `switch` constructs to execute statements selectively
- Creating and using loops—`while`, `do-while`, `for`, and enhanced `for`
- Creating nested constructs for selection and iteration statements
- Comparing the `do-while`, `while`, `for`, and enhanced `for` loop constructs
- Using `break` and `continue` statements

In Java, you can execute your code conditionally by using either the `if` or `switch` constructs. Let's start with the `if` construct.

5.1 The if and if-else constructs

[3.4] Create if and if-else constructs

In this section, we'll cover `if` and `if-else` constructs. We'll examine what happens when these constructs are used with and without curly braces {}. We'll also cover nested `if` and `if-else` constructs.

5.1.1 The if construct and its flavors

An `if` construct enables you to execute a set of statements in your code based on the result of a condition. This condition must always evaluate to a `boolean` or a `Boolean` value. (The `Boolean` wrapper class isn't covered in the OCA Java SE 7 Programmer I exam, so you won't see it being used in the coding examples in this book.) You can specify a set of statements to execute when this condition evaluates to `true` or `false`. (In many Java books, the terms *constructs* and *statements* are used interchangeably.)

Multiple flavors of the `if` statement are illustrated in figure 5.1:

- `if`
- `if-else`
- `if-else-if-else`

In figure 5.1, *condition1* and *condition2* refer to a variable or an expression that must evaluate to a `boolean` or `Boolean` value. In the figure, *statement1, statement2, and statement3* refer to either a single line of code or a code block.

Because the `Boolean` wrapper class isn't covered in the OCA Java SE 7 Programmer I exam, I won't cover it in the coding examples in this book. We'll work with only the `boolean` data type.

EXAM TIP In Java, *then* isn't a keyword, so it shouldn't be used with the `if` statement.

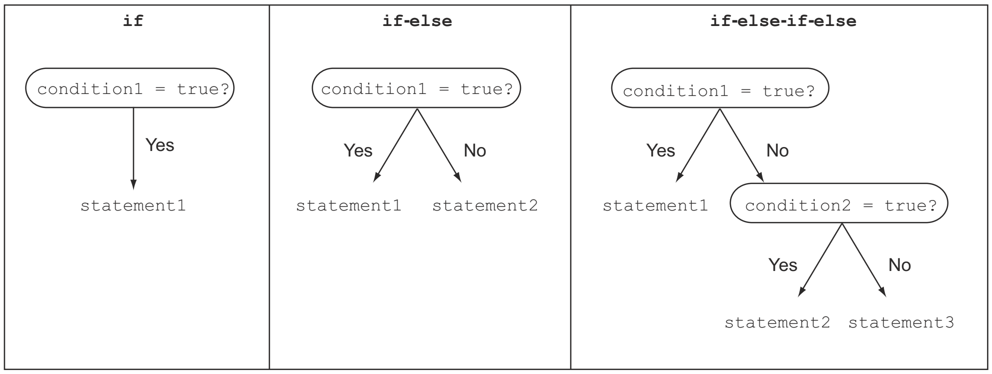


Figure 5.1 Multiple flavors of the `if` statement: `if`, `if-else`, and `if-else-if-else`

Let's look at the use of the previously mentioned `if` statement flavors by first defining a set of variables: `score`, `result`, `name`, and `file`, as follows:

```
int score = 100;
String result = "";
String name = "Lion";
java.io.File file = new java.io.File("F");
```

Figure 5.2 shows the use of `if`, `if-else`, and `if-else-if-else` constructs and compares them by showing the code side by side.

Let's quickly go through the code used in figure 5.2's `if`, `if-else`, and `if-else-if-else` statements. In the following example code, if the condition `name.equals("Lion")` evaluates to `true`, a value of `200` is assigned to the variable `score`:

```
if (name.equals("Lion"))
   score = 200;
```

Example of if construct

In the following example, if the condition `name.equals("Lion")` evaluates to `true`, a value of `200` is assigned to the variable `score`. If this condition were to evaluate to `false`, a value of `300` would be assigned to the variable `score`:

```
if (name.equals("Lion"))
     score = 200;
else
     score = 300;
```

Example of if-else construct

In the following example, if `score` is equal to `100`, the variable `result` is assigned a value of `A`. If `score` is equal to `50`, the variable `result` is assigned a value of `B`. If `score` is equal to `10`, the variable `result` is assigned a value of `C`. If `score` doesn't match any of `100`, `50`, or `10`, a value of `F` is assigned to the variable `result`. An `if-else-if-else` construct can use different conditions for all its `if` constructs:

`if`	**`if-else`**	**`if-else-if-else`**
`if (name.equals("Lion"))`	`if (name.equals("Lion"))`	`if (score == 100)`
`    score = 200;`	`    score = 200;`	`    result = "A";`
	`else`	`else if (score == 50)`
	`    score = 300;`	`    result = "B";`
		`else if (score == 10)`
		`    result = "C";`
		`else`
		`    result = "F";`

Figure 5.2 Multiple flavors of `if` statements implemented in code

Condition 1 -> score == 100

```
if (score == 100)
    result = "A";
else if (score == 50)
    result = "B";
else if (score == 10)
    result = "C";
else
     result = "F";
```

Condition 2 -> score == 50

Condition 3 -> score == 10

If none of previous conditions evaluates to true, execute else statement

Figure 5.3 illustrates the previous code and makes several points clear:

- The last `else` statement is part of the last `if` construct, not any of the `if` constructs before it.
- The `if-else-if-else` is an `if-else` construct in which the `else` part defines another `if` construct. A few other programming languages, such as VB and C#, use `if-elsif` and `if-elseif` (without a space) constructs to define `if-else-if` constructs. If you've programmed with any of these languages, note the difference in Java. The following code is equal to the previous code:

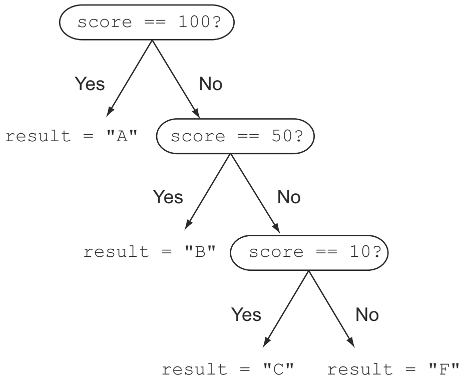


Figure 5.3 Execution of the `if-else-if-else` code

```
if (score == 100)
    result = "A";
else
    if (score == 50)
        result = "B";
    else
        if (score == 10)
            result = "C";
        else
            result="F";
```

Again, note that none of the previous `if` constructs use *then* to define the code to execute if a condition evaluates to `true`. Unlike other programming languages, *then* isn't a keyword in Java and isn't used with the `if` construct.

EXAM TIP The `if-else-if-else` is an `if-else` construct in which the `else` part defines another `if` construct.

The `boolean` expression used as a condition for the `if` construct can also include assignment operation. The following Twist in the Tale exercise throws in a twist by modifying the value of a variable that the `if` statement is comparing. Let's see if you can answer it correctly (answer in the appendix).

Twist in the Tale 5.1

Let's modify the code used in the previous example as follows. What is the output of this code?

```
String result = "1";
int score = 10;
if ((score = score+10) == 100)
    result = "A";
else if ((score = score+29) == 50)
    result = "B";
else if ((score = score+200) == 10)
    result = "C";
else
     result = "F";

System.out.println(result + ":" + score);
```

- **a** A:10
- **b** C:10
- **c** A:20
- **d** B:29
- **e** C:249
- **f** F:249

5.1.2 Missing else blocks

What happens if you don't define the `else` statement for an `if` construct? It's acceptable to define one course of action for an `if` construct as follows (omitting the `else` part):

```
boolean testValue = false;
if (testValue == true)
    System.out.println("value is true");
```

But you can't define the `else` part for an `if` construct, skipping the `if` code block. The following code won't compile:

```
boolean testValue = false;
if (testValue == true)
else                                                    <- Won't compile
    System.out.println("value is false");
```

Here is another interesting and bizarre piece of code:

```
int score = 100;
if((score=score+10) > 110);                       <- ❶ Missing then or else part
```

❶ is a valid line of code, even if it doesn't define either the *then* or `else` part of the `if` statement. In this case, the `if` condition evaluates and that's it. The `if` construct doesn't define any code that should execute based on the result of this condition.

NOTE Using `if(testValue==true)` is the same as using `if(testValue)`. Similarly, `if(testValue==false)` is the same as using `if(!testValue)`. This book includes examples of both these approaches. A lot of beginners in Java programming find the latter approach (without the explicit `==`) confusing.

5.1.3 Implications of the presence and absence of {} in if-else constructs

You can execute a single statement or a block of statements when an `if` condition evaluates to `true` or `false`. An `if` block is marked by enclosing one or more statements within a pair of curly braces, `{}`. An `if` block will execute a single line of code if there are no braces, but will execute an unlimited number of lines if they're contained within a block (defined using braces). The braces are optional if there's only one line in the `if` statement.

The following code executes only one statement of assigning value `200` to variable `score` if the expression used in the `if` statement evaluates to `true`:

```
String name = "Lion";
int score = 100;
if (name.equals("Lion"))
    score = 200;
```

What happens if you want to execute another line of code if the value of the variable `name` is equal to `Lion`? Is the following code correct?

```
String name = "Lion";
int score = 100;
if (name.equals("Lion"))
    score = 200;
    name = "Larry";                    <- Set name to Larry
```

The statement `score = 200;` executes if the `if` condition is `true`. Although it looks like the statement `name = "Larry";` is part of the `if` statement, it isn't. It will execute regardless of the result of the `if` condition because of the lack of braces, `{}`.

EXAM TIP In the exam, watch out for code like this that uses misleading indentation in `if` constructs. In the absence of a defined code block (marked with a pair of `{}`), only the statement following the `if` construct will be considered to be part of it.

What happens to the same code if you define an `else` part for your `if` construct, as follows:

```
String name = "Lion";
int score = 100;
if (name.equals("Lion"))
    score = 200;
    name = "Larry";                    <- This statement isn't part of the if construct
else
    score = 129;
```

In this case, the code won't compile. The compiler will report that the else part is defined without an if statement. If this leaves you confused, examine the following code, which is indented in order to emphasize the fact that the name = "Larry" line isn't part of the else construct:

```
String name = "Lion";
int score = 100;
if (name.equals ("Lion"))
    score = 200;

name = "Larry";

else
    score = 129;
```

This statement isn't part of the if construct

The else statement seems to be defined without a preceding if construct

If you want to execute multiple statements for an if construct, define them within a block of code. You can do so by defining all this code within curly braces: {}. Here's an example:

```
String name = "Lion";
int score = 100;
if (name.equals("Lion")) {
    score = 200;
    name = "Larry";
}
else
    score = 129;
```

Start of code block

Statements to execute if (name.equals("Lion")) evaluates to true

End of code block

Similarly, you can define multiple lines of code for the else part. The following example does so incorrectly:

```
String name = "Lion";
if (name.equals("Lion"))
    System.out.println("Lion");
else
    System.out.println("Not a Lion");
    System.out.println("Again, not a Lion");
```

1 Not part of the else construct. Will execute regardless of value of variable name.

The output of the previous code is as follows:

```
Lion
Again, not a Lion
```

Though the code at 1 *looks* like it will execute only if the value of the variable name matches the value Lion, this is not the case. It is indented incorrectly to trick you into believing that it is a part of the else block. The previous code is the same as the following code (with correct indentation):

```
String name = "Lion";
if (name.equals("Lion"))
    System.out.println("Lion");
else
    System.out.println("Not a Lion");
System.out.println("Again, not a Lion");
```

Not part of the else construct. Will execute regardless of value of variable name.

If you wish to execute the last two statements in the previous code only if the `if` condition evaluates to `false`, you can do so by using {}:

```
String name = "Lion";
if (name.equals("Lion"))
    System.out.println("Lion");
else {
    System.out.println("Not a Lion");
    System.out.println("Again, not a Lion");
   }
```

Now part of the else construct. Will execute only when if condition evaluates to false.

You can define another statement, construct, or loop to execute for an `if` condition, without using {}, as follows:

```
String name = "Lion";
if (name.equals("Lion"))
    for (int i = 0; i < 3; ++i)
        System.out.println(i);
```

An if condition

A for loop is a single construct that will execute if name.equals("Lion") evaluates to true

This code is part of the for loop defined on previous line

`System.out.println(i)` is part of the `for` loop, not an unrelated statement that follows the `for` loop. So this code is correct and gives the following output:

```
0
1
2
```

5.1.4 Appropriate versus inappropriate expressions passed as arguments to an if statement

The result of a variable or an expression used in an `if` construct must evaluate to `true` or `false`. Assume the following definitions of variables:

```
int score = 100;
boolean allow = false;
String name = "Lion";
```

Let's look at a few examples of some of the valid variables and expressions that can be passed to an `if` statement.

```
(score == 100)
(name == "Lio")
(score <= 100 || allow)
(allow)
```

Evaluates to true

Evaluates to false

Evaluates to true

Evaluates to false

Note that using `==` is not good practice for comparing two `String` objects for equality. As mentioned in chapter 4, the correct approach for comparing two `String` objects is to use the `equals` method from the `String` class. But comparing two `String` values using `==` is a valid expression that returns a `boolean` value, and it may be used in the exam.

Now comes the tricky part of passing an assignment operation to an `if` construct. What do you think is the output of the following code?

```
boolean allow = false;
if (allow = true)
    System.out.println("value is true");
```

This is assignment, not comparison

```
else
    System.out.println("value is false");
```

You may think that because the value of the boolean variable allow is set to false, the previous code output's value is false. Revisit the code and notice that the assignment operation allow = true assigns the value true to the boolean variable allow. Also, its result is also a boolean value, which makes it eligible to be passed on as an argument to the if construct.

Although the previous code has no syntactical errors, there is a *logical error*—an error in the program logic. The correct code to compare a boolean variable with a boolean literal value is as follows:

```
boolean allow = false;
if (allow == true)
    System.out.println("value is true");
else
    System.out.println("value is false");
```

This is comparison

EXAM TIP Watch out for code in the exam that uses the assignment operator (=) to compare a boolean value in the if condition. It won't compare the boolean value; it'll assign a value to it. The correct operator for comparing a boolean value is the equality operator (==).

5.1.5 *Nested if constructs*

A nested if construct is an if construct defined within another if construct. Theoretically, there is no limit on the levels of nested if and if-else constructs.

Whenever you come across nested if and if-else constructs, you need to be careful about determining the else part of an if statement. If this statement doesn't make a lot of sense, take a look at the following code and determine its output:

```
int score = 110;
if (score > 200)
    if (score <400)
        if (score > 300)
            System.out.println(1);
        else
            System.out.println(2);
else
    System.out.println(3);
```

❶ **if (score>200)**

❷ **if (score<400)**

❸ **To which if does this else belong?**

Based on the way the code is indented, you may believe that the else at ❸ belongs to the if defined at ❶. But it belongs to the if defined at ❷. Here's the code with the correct indentation:

```
int score = 110;
if (score > 200)
    if (score <400)
        if (score > 300)
            System.out.println(1);
        else
            System.out.println(2);
```

```
else
        System.out.println(3);
```

This else belongs to the if with condition (score<400)

Next, you need to understand how to do the following:

- How to define an `else` for an outer `if` other than the one that it'll be assigned to by default
- How to determine to which `if` an `else` belongs in nested `if` constructs

Both of these tasks are simple. Let's start with the first one.

How to define an else for an outer if other than the one that it'll be assigned to by default

The key point is to use curly braces, as follows:

```
int score = 110;
if (score > 200) {
    if (score < 400)
        if (score > 300)
            System.out.println(1);
        else
            System.out.println(2);
}
else
    System.out.println(3);
```

❶ Start if construct for score > 200

❷ End if construct for score > 200

❸ else for score > 200

Curly braces at ❶ and ❷ mark the start and end of the `if` condition (`score > 200`) defined at ❶. Hence, the `else` at ❸ that follows ❷ belongs to the `if` defined at ❶.

How to determine to which if an else belongs in nested if constructs

If code uses curly braces to mark the start and end of the territory of an `if` or `else` construct, it can be simple to determine which `else` goes with which `if`, as mentioned in the previous section. When the `if` constructs don't use curly braces, don't get confused by the code indentation, which may or may not be correct.

Try to match the `if`s with their corresponding `else`s in the following poorly indented code:

```
if (score > 200)
if (score < 400)
if (score > 300)
    System.out.println(1);
else
    System.out.println(2);
else
    System.out.println(3);
```

Start working from the inside out, with the innermost `if-else` statement, matching each `else` with its nearest unmatched `if` statement. Figure 5.4 shows how to match the `if-else` pairs for the previous code, marked with *1*, *2*, and *3*.

You can alternatively use the `switch` statement to execute code conditionally. Even though both `if-else` constructs and `switch` statements are used to execute statements selectively, they differ in their usage, which you'll notice as you work with the `switch` statement in the next section.

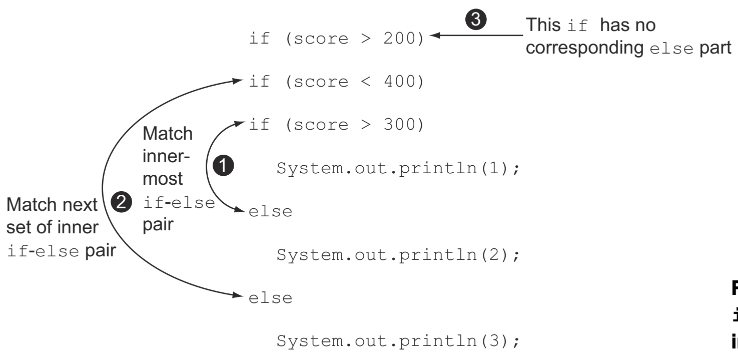

```
if (score > 200)
if (score < 400)
if (score > 300)
    System.out.println(1);
else
    System.out.println(2);
else
    System.out.println(3);
```
3 This if has no corresponding else part
1 Match inner-most if-else pair
2 Match next set of inner if-else pair

Figure 5.4 How to match if-else pairs for poorly indented code

5.2 The switch statement

[3.5] Use a switch statement

In this section, you'll learn how to use the `switch` statement and see how it compares to nested `if-else` constructs. You'll learn the right ingredients for defining values that are passed to the `switch` labels and the correct use of the `break` statement in these labels.

5.2.1 Create and use a switch statement

You can use a `switch` statement to compare the value of a variable with multiple values. For each of these values, you can define a set of statements to execute.

The following example uses a `switch` statement to compare the value of the variable `marks` with the literal values `10`, `20`, and `30`, defined using the `case` keyword:

```
int marks = 20;
switch (marks) {                                  // Compare value of marks
    case 10: System.out.println(10);              // Values that are
         break;                                   // Marks the end of a case label
    case 20: System.out.println(20);              // compared to
         break;
    case 30: System.out.println(30);              // variable marks
         break;
    default: System.out.println("default");       // Default case to execute if
         break;                                   // no matching cases found
}
```

A `switch` statement can define multiple `case` labels within its switch block, but only a single `default` label. The `default` label executes when no matching value is found in the `case` labels. A `break` statement is used to exit a `switch` statement, after the code completes its execution for a matching `case`.

5.2.2 Comparing a switch statement with multiple if-else constructs

A `switch` statement can improve the readability of your code by replacing a set of (rather complicated-looking) related `if-else-if-else` statements with a `switch` and multiple `case` statements.

Examine the following code, which uses `if-else-if-else` statements to check the value of a `String` variable `day` and display an appropriate message:

```
String day = "SUN";
if (day.equals("MON") || day.equals("TUE")||
    day.equals("WED") || day.equals("THU"))
    System.out.println("Time to work");
else if (day.equals("FRI"))
    System.out.println("Nearing weekend");
else if (day.equals("SAT") || day.equals("SUN"))
    System.out.println("Weekend!");
else
    System.out.println("Invalid day?");
```

Multiple comparisons

Now examine this implementation of the previous logic using the `switch` statement:

```
String day = "SUN";
switch (day) {
    case "MON":
    case "TUE":
    case "WED":
    case "THU": System.out.println("Time to work");
                break;
    case "FRI": System.out.println("Nearing weekend");
                break;
    case "SAT":
    case "SUN": System.out.println("Weekend!");
                break;
    default: System.out.println("Invalid day?");
}
```

The two previous snippets of code perform the same function of comparing the value of the variable `day` and printing an appropriate value. But the latter code, which uses the `switch` statement, is simpler and easier to read and follow.

Note that the previous `switch` statement doesn't define code for all the `case` values. What happens if the value of the variable `day` matches `TUE`? When the code control enters the label matching `TUE` in the `switch` construct, it'll execute all of the code until it encounters a `break` statement or it reaches the end of the `switch` statement.

Figure 5.5 depicts the execution of the multiple `if-else-if-else` statements used in the example code in this section. You can compare it to a series of questions and answers that continue until a match is found or all the conditions are evaluated.

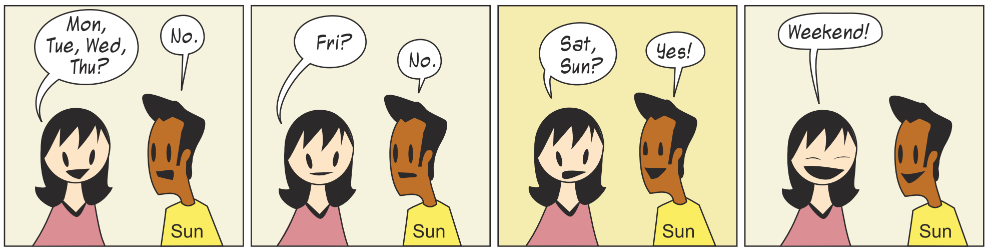


Figure 5.5 **The `if-else-if-else` construct is like a series of questions and answers.**

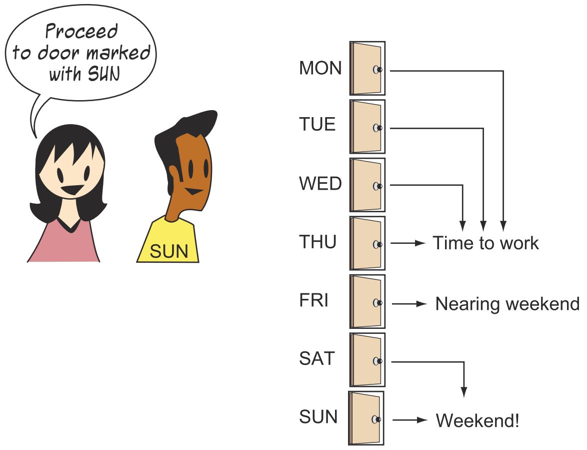


Figure 5.6 A `switch` statement is like asking a question and acting on the answer.

As opposed to an `if-else-if-else` construct, you can compare a `switch` statement to asking a single question and evaluating its answer to determine which code to execute. Figure 5.6 illustrates the `switch` statement and its `case` labels.

EXAM TIP The `if-else-if-else` construct evaluates all of the conditions until it finds a match. A `switch` construct compares the argument passed to it with its labels.

Let's see if you can find the twist in the next exercise. Hint: it defines code to compare `String` values (answers can be found in the appendix).

Twist in the Tale 5.2

Let's modify the code used in the previous example as follows. What is the output of this code?

```
String day = new String("SUN");
switch (day) {
    case "MON":
    case "TUE":
    case "WED":
    case "THU": System.out.println("Time to work");
                break;
    case "FRI": System.out.println("Nearing weekend");
                break;
    case "SAT":
    case "SUN": System.out.println("Weekend!");
                break;
    default: System.out.println("Invalid day?");
}
```

- **a** Time to work
- **b** Nearing weekend
- **c** Weekend!
- **d** Invalid day?

5.2.3 Arguments passed to a switch statement

You can't use the `switch` statement to compare all types of values, such as all types of objects and primitives. There are limitations on the types of arguments that a `switch` statement can accept.

Figure 5.7 shows the types of arguments that can be passed to a `switch` statement and to an `if` construct.

A `switch` statement accepts arguments of type `char`, `byte`, `short`, `int`, and `String` (starting in Java version 7). It also accepts arguments and expressions of types `enum`, `Character`, `Byte`, `Integer`, and `Short`, but because these aren't on the OCA Java SE 7 Programmer I exam objectives, I won't cover them any further. The `switch` statement doesn't accept arguments of type `long`, `float`, or `double`, or any object besides `String`.

Apart from passing a variable to a `switch` statement, you can also pass an expression to the `switch` statement as long as it returns one of the allowed types. The following code is valid:

```
int score =10, num = 20;
switch (score+num) {
    // ..code
}
```

Type of score+num is int and can thus be passed as an argument to switch statement

The following code won't compile because the type of `history` is `double`, which is a type that isn't accepted by the `switch` statement:

```
double history = 20;
switch (history) {
    // ..code
}
```

Double variable can't be passed as an argument to switch statement

EXAM TIP Watch out for questions in the exam that try to pass a primitive decimal type such as `float` or `double` to a `switch` statement. Code that tries to do so will not compile.

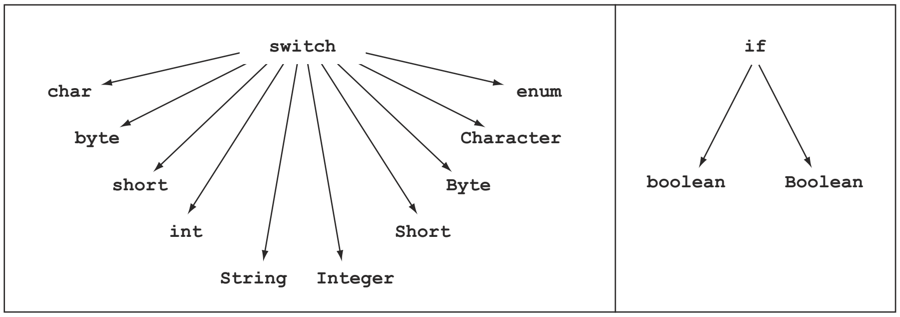


Figure 5.7 Types of arguments that can be passed to a `switch` statement and an `if` construct

5.2.4 *Values passed to the label case of a switch statement*

You're constrained in a couple of ways when it comes to the value that can be passed to the `case` label in a `switch` statement, as the following subsections explain.

CASE VALUES SHOULD BE COMPILE-TIME CONSTANTS

The value of a `case` label must be a compile-time constant value; that is, the value should be known at the time of code compilation:

```
int a=10, b=20, c=30;
switch (a) {
    case b+c: System.out.println(b+c); break;            ❶ Not allowed
    case 10*7: System.out.println(10*7512+10); break;    ❷ Allowed
}
```

Note that `b+c` in the previous code defined at ❶ can't be determined at the time of compilation and isn't allowed. But `10*7` defined at ❷ is a valid `case` label value.

You can use variables in an expression if they're marked `final` because the value of `final` variables can't change once they're initialized:

```
final int a = 10;
final int b = 20;
final int c = 30;

switch (a) {
    case b+c: System.out.println(b+c); break;    ❶ Expression b+c is compile-time constant
}
```

Because the variables `b` and `c` are `final` variables here, at ❶ the value of `b+c` can be known at compile time. This makes it a compile-time constant value, which can be used in a `case` label.

You may be surprised to learn that if you don't assign a value to a `final` variable with its declaration, it isn't considered a compile-time constant:

```
final int a = 10;
final int b = 20;
final int c;              ❶ final variable c is defined but not initialized
c = 30;                   ❷ c is initialized

switch (a) {
    case b+c: System.out.println(b+c); break;    ❸ Code doesn't compile. b+c isn't considered a constant expression because the variable c wasn't initialized with its declaration.
}
```

This code defines a `final` variable `c` at line ❶ but doesn't initialize it. The `final` variable `c` is initialized at line ❷. Because the `final` variable `c` isn't initialized with its declaration, at ❸ the expression `b+c` isn't considered a compile-time constant, so it can't be used as a case label.

CASE VALUES SHOULD BE ASSIGNABLE TO THE ARGUMENT PASSED TO THE SWITCH STATEMENT

Examine the following code, in which the type of argument passed to the `switch` statement is `byte` and the `case` label value is of the type `float`. Such code won't compile:

```
byte myByte = 10;
switch (myByte) {
    case 1.2: System.out.println(1); break;
}
```

Floating-point number can't be assigned to byte variable

NULL ISN'T ALLOWED AS A CASE LABEL

Code that tries to compare the variable passed to the `switch` statement with `null` won't compile, as demonstrated in the following code:

```
String name = "Paul";
switch (name) {
    case "Paul": System.out.println(1);
            break;
    case null: System.out.println("null");
}
```

null isn't allowed as a case label

ONE CODE BLOCK CAN BE DEFINED FOR MULTIPLE CASES

It's acceptable to define a single code block for multiple `case` labels in a `switch` statement, as shown by the following code:

```
int score =10;
switch (score) {
    case 100:
    case 50 :
    case 10 : System.out.println("Average score");
          break;
    case 200: System.out.println("Good score");
}
```

You can define multiple cases, which should execute the same code block

This example code will output `Average score` if the value of the variable `score` matches any of the values `100`, `50`, and `10`.

5.2.5 Use of break statements within a switch statement

In the previous examples, note the use of `break` to exit the `switch` construct once a matching case is found. In the absence of the `break` statement, control will *fall through* the remaining code and execute the code corresponding to all the *remaining* cases that *follow* that matching case.

Consider the examples shown in figure 5.8—one with a `break` statement and the other without a `break` statement. Examine the flow of code (depicted using arrows) in this figure when the value of the variable `score` is equal to `50`.

Our (hypothetical) enthusiastic programmers, Harry and Selvan, who are also preparing for this exam, sent in some of their code. Can you choose the correct code for them in the following Twist in the Tale exercise? (Answers are in the appendix).

Twist in the Tale 5.3

Which of the following code submissions by our two hypothetical programmers, Harry and Selvan, examines the value of the long variable `dayCount` and prints out the name of any one month that matches the day count?

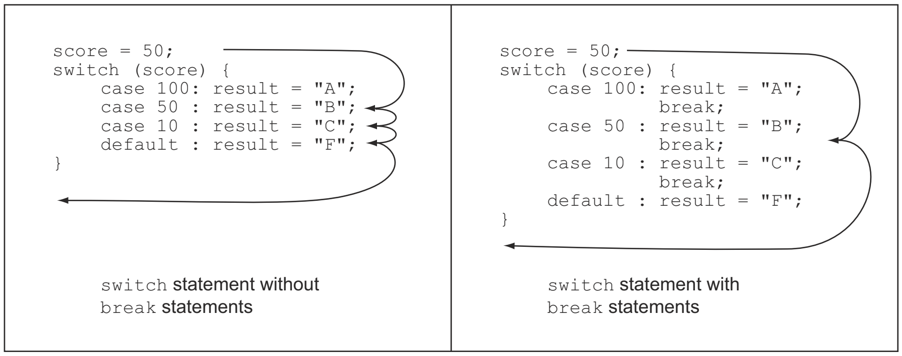
a Submission by Harry:

```
long dayCount = 31;
if (dayCount == 28 || dayCount == 29)
    System.out.println("Feb");
else if (dayCount == 30)
    System.out.println("Apr");
else if (dayCount == 31)
    System.out.println("Jan");
```

b Submission by Selvan:

```
long dayCount = 31;
switch (dayCount) {
    case 28:
    case 29: System.out.println("Feb"); break;
    case 30: System.out.println("Apr"); break;
    case 31: System.out.println("Jan"); break;
}
```

In the next section, I'll cover the iteration statements known as loop statements. Just as you'd like to repeat the action of "eating an ice cream" every day, loops are used to execute the same lines of code multiple times. You can use a `for` loop, an enhanced `for` (`for`-each) loop, or the `do-while` and `while` loops to repeat a block of code. Let's start with the `for` loop.



```
score = 50;
switch (score) {
    case 100: result = "A";
    case 50 : result = "B";
    case 10 : result = "C";
    default : result = "F";
}
```

`switch` statement without `break` statements

```
score = 50;
switch (score) {
    case 100: result = "A";
              break;
    case 50 : result = "B";
              break;
    case 10 : result = "C";
              break;
    default : result = "F";
}
```

`switch` statement with `break` statements

Figure 5.8 Differences in code flow for a `switch` statement with and without `break` statements

5.3 The for loop

[5.2] Create and use for loops including the enhanced for loop

In this section, I'll cover the regular `for` loop. The enhanced `for` loop is covered in the next section.

A `for` loop is *usually* used to execute a set of statements a fixed number of times. It takes the following form:

```
for (initialization; condition; update) {
     statements;
}
```

Here's a simple example:

```
int tableOf = 25;
for (int ctr = 1; ctr <= 5; ctr++) {
    System.out.println(tableOf * ctr);            ❶ Executes multiple times
}
```

The output of the previous code is as follows:

```
25
50
75
100
125
```

In the previous example, the code at ❶ will execute five times. It'll start with an initial value of `1` for the variable `ctr` and execute while the value of the variable `ctr` is less than or equal to `5`. The value of variable `ctr` will increment by `1` (`ctr++`) after the execution of the code at ❶.

The code at ❶ executes for `ctr` values `1`, `2`, `3`, `4`, and `5`. Because `6 <= 5` evaluates to `false`, the `for` loop completes its execution without executing the code at ❶ any further.

In the previous example, notice that the `for` loop defines three types of statements separated with semicolons (`;`), as follows:

- Initialization statements
- Termination condition
- Update clause (executable statement)

This loop is depicted as a flowchart in figure 5.9. The statements defined within the loop body execute until the termination condition is `false`.

One important point to note with respect to the `for` loop is that the update clause executes after all the statements defined within the `for` loop body. In other words, you can consider the update clause to be a last statement in the `for` loop. The initialization section, which executes only once, may define multiple initialization statements. Similarly, the update clause may define multiple statements. But there can be only one termination condition for a `for` loop.

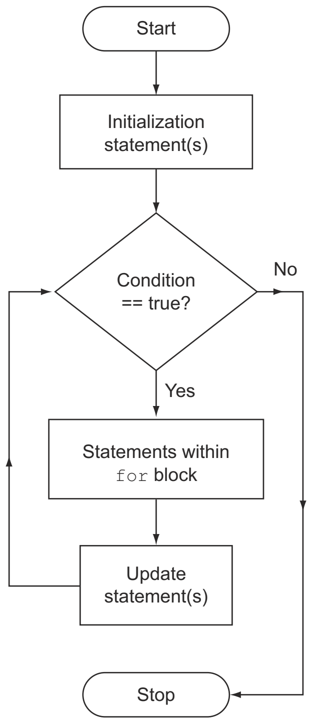
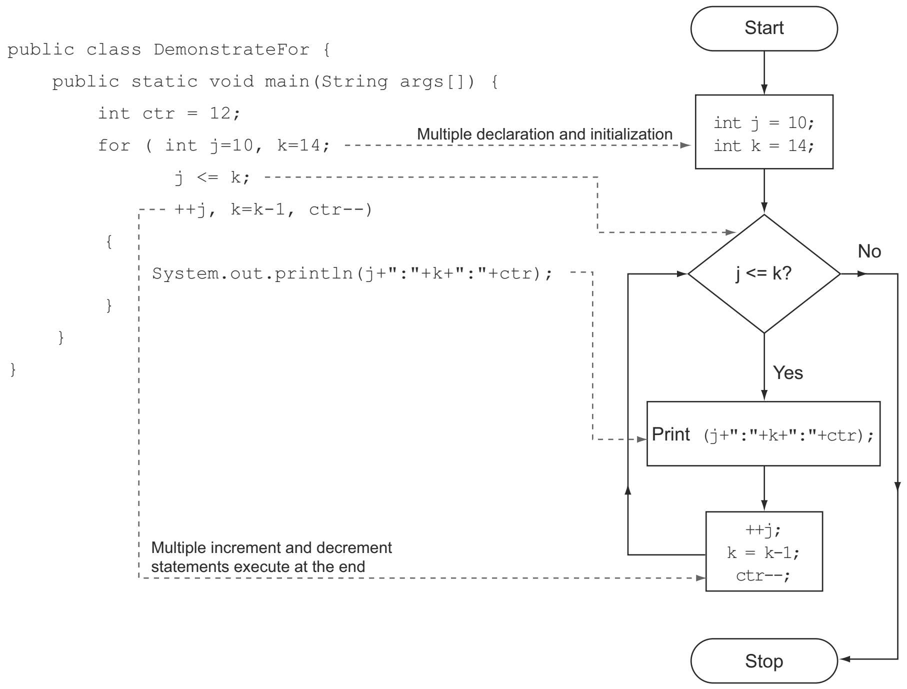


Figure 5.9 The flow of control in a `for` loop



```
public class DemonstrateFor {
    public static void main(String args[]) {
        int ctr = 12;
        for ( int j=10, k=14;
                j <= k;
              ++j, k=k-1, ctr--)
        {
            System.out.println(j+":"+k+":"+ctr);
        }
    }
}
```

Multiple declaration and initialization
Multiple increment and decrement statements execute at the end
Start
int j = 10;
int k = 14;
j <= k?
No
Yes
Print (j+":"+k+":"+ctr);
++j;
k = k-1;
ctr--;
Stop

Figure 5.10 The flow of control in a `for` loop using a code example

In figure 5.10, I've provided a code snippet and a flowchart that depicts the corresponding flow of execution of statements to explain the previous concept.

Let's explore the initialization block, termination condition, and update clause of a `for` loop in detail.

5.3.1 Initialization block

An initialization block executes only once. A `for` loop can declare and initialize multiple variables in its initialization block, but the variables it declares should be of the same type. The following code is valid:

```
int tableOf = 25;
for (int ctr = 1, num = 100000; ctr <= 5; ++ctr) {
    System.out.println(tableOf * ctr);
    System.out.println(num * ctr);
}
```

Define and assign multiple variables

But you can't declare variables of different types in an initialization block. The following code will fail to compile:

```
for (int j=10, long longVar = 10; j <= l; ++j) { }
```

Can't define variables of different types in an initialization block

It's a common programming mistake to try to use the variables defined in a `for`'s initialization block outside the `for` block. Please note that the scope of the variables declared in the initialization block is limited to the `for` block. An example follows:

```
int tableOf = 25;
for (int ctr = 1; ctr <= 5; ++ctr) {
    System.out.println(tableOf * ctr);
}
ctr = 20;
```

Variable ctr is accessible only within for loop body

Variable ctr isn't accessible outside for loop

5.3.2 Termination condition

The termination condition is evaluated once for each iteration before executing the statements defined within the body of the loop. The `for` loop terminates when the termination condition evaluates to `false`:

```
for (int ctr = 1; ctr <= 5; ++ctr) {
    System.out.println(ctr);
}
...
```

1 for loop body

2 Code following the for loop

The termination condition—`ctr <= 5` in this example—is checked before 1 executes. If the condition evaluates to `false`, control is transferred to 2. A `for` loop can define exactly one termination condition—no more, no less.

5.3.3 The update clause

Usually, you'd use this block to manipulate the value of the variable that you used to specify the termination condition. In the previous example, I defined the following code in this section:

```
++ctr;
```

Code defined in this block executes *after* all the code defined in the body of the `for` loop. The previous code increments the value of the variable `ctr` by `1` after the following code executes:

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

The termination condition is evaluated next. This execution continues until the termination condition evaluates to `false`.

You can define multiple statements in the update clause, including calls to other methods. The only limit is that these statements will execute in the order in which they appear, at the end of all the statements defined in the `for` block. Examine the following code, which calls a method in the update block:

```
public class ForIncrementStatements {
    public static void main(String args[]) {
        String line = "ab";
        for (int i=0; i < line.length(); ++i, printMethod())
            System.out.println(line.charAt(i));
    }
    private static void printMethod() {
        System.out.println("Happy");
    }
}
```

The increment block can also call methods

printMethod is called by the for loop's increment block

The output of this code is as follows:

```
a
Happy
b
Happy
```

5.3.4 Nested for loop

If a loop encloses another loop, they are called *nested loops*. The loop that encloses another loop is called the *outer loop*, and the enclosed loop is called the *inner loop*. Theoretically, there are no limits on the levels of nesting `for` loops.

Let's get started with a single-level nested loop. For an example, you can compare the hour hand of a clock to an outer loop and its minute hand to an inner loop. Each hour can be compared with an iteration of the outer loop, and each minute can be compared with an iteration of the inner loop. Because an hour has 60 minutes, the inner loop should iterate 60 times *for each iteration* of the outer loop. This comparison between a clock and a nested loop is shown in figure 5.11.

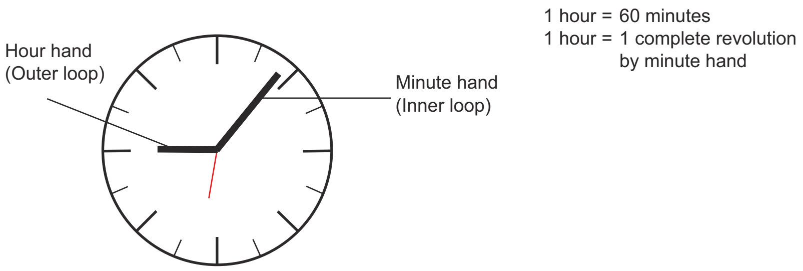


Figure 5.11 Comparison of the hands of a clock to a nested loop

You can use the following nested `for` loops to print out each minute (`1` to `60`) for hours from `1` to `6`:

```
for (int hrs=1; hrs<=6; hrs++) {
    for (int min=1; min<=60; min++) {
        System.out.println(hrs+":"+min);
    }
}
```

Outer loop iterates for values 1 through 6

Inner loop iterates for values 1 through 60

Executes 6 × 60 times (total outer loop iterations × total inner loop iterations)

Nested loops are often used to initialize or iterate multidimensional arrays. The following code initializes a multidimensional array using nested `for` loops:

```
int multiArr[][];
multiArr = new int[2][3];

for (int i=0; i<multiArr.length; i++) {
    for (int j=0; j<multiArr[i].length; j++) {
        multiArr[i][j] = i + j;
    }
}
```

1 Array declaration

2 Array allocation

3 Outer for loop

4 Inner for loop

Inner for loop ends

Outer for loop ends

❶ defines a two-dimensional array `multiArr`. ❷ allocates this array, creating two rows and three columns, and assigns all array members the default `int` value of `0`.

❸ defines an outer `for` loop. Because the value of `multiArr.length` is `2` (the value of the first subscript at ❷), the outer `for` loop executes twice, with variable `i` having values `0` and `1`. The inner `for` loop is defined at ❹. Because the length of each of the rows of the `multiArr` array is `3` (the value of the second subscript at ❷), the inner loop executes three times for each iteration of the outer `for` loop, with variable `j` having values `0`, `1`, and `2`.

Figure 5.12 illustrates the array `multiArr` after it's initialized using the previous code.

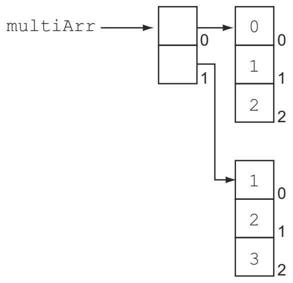


Figure 5.12 The array `multiArr` after it's initialized using the previous code

In the next section, I discuss another flavor of the `for` loop: the *enhanced* `for` loop or `for`-each loop.

5.4 The enhanced for loop

[5.2] Create and use for loops including the enhanced for loop

The enhanced `for` loop is also called the *for-each* loop, and it offers some advantages over the regular `for` loop. It also has some limitations.

To start with, the regular `for` loop is cumbersome to use when it comes to iterating through a collection or an array. You need to create a looping variable and specify the start and end positions of the collection or the array, even if you want to iterate through the complete collection or list. The enhanced `for` loop makes the previously mentioned routine task quite a breeze, as the following example demonstrates for an `ArrayList myList`:

```
ArrayList<String> myList= new ArrayList<String>();
myList.add("Java");
myList.add("Loop");
```

The following code uses the regular `for` loop to iterate through this list:

```
for(Iterator<String> i = myList.iterator(); i.hasNext();)
    System.out.println(i.next());
```

The following code uses the enhanced `for` loop to iterate through the list `myList`:

```
for (String val : myList)
    System.out.println(val);
```

You can read the colon (`:`) in a `for`-each loop as "in".

The `for`-each loop is a breeze to implement: there's no code clutter, and the code is easy to write and comprehend. In the previous example, the `for`-each loop is read as "for each element `val` in collection `myList`, print the value of `val`."

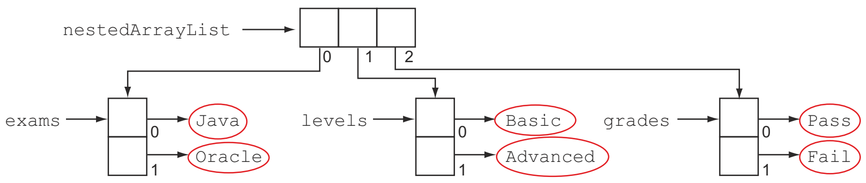


Figure 5.13 Pictorial representation of `nestedArrayList`

You can also easily iterate through *nested collections* using the enhanced `for` loop. In this example, assume that an `ArrayList` of `exams`, `levels`, and `grades` are defined as follows:

```
ArrayList<String> exams= new ArrayList<String>();
exams.add("Java"); exams.add("Oracle");

ArrayList<String> levels= new ArrayList<String>();
levels.add("Basic"); levels.add("Advanced");

ArrayList<String> grades= new ArrayList<String>();
grades.add("Pass"); grades.add("Fail");
```

The following code creates a nested `ArrayList`, `nestedArrayList`, every element of which is itself an `ArrayList` of `String` objects:

ArrayList of ArrayList

```
ArrayList<ArrayList<String>> nestedArrayList =
                                   new ArrayList< ArrayList<String>>();
nestedArrayList.add(exams);
nestedArrayList.add(levels);
nestedArrayList.add(grades);
```

Add object of ArrayList to nestedArrayList

The `nestedArrayList` can be compared to a multidimensional array, as shown in figure 5.13.

A nested enhanced `for` loop can be used to iterate through the nested `ArrayList` `nestedArrayList`. Here's the relevant code:

```
for (ArrayList<String> nestedListElement : nestedArrayList)
    for (String element : nestedListElement)
        System.out.println(element);
```

The output of this code is as follows:

```
Java
Oracle
Basic
Advanced
Pass
Fail
```

The enhanced `for` loop is again a breeze to use to iterate through *nested or non-nested* arrays. For example, you can use the following code to iterate through an array of elements and calculate its total:

```
int total = 0;
int primeNums[] = {2, 3, 7, 11};
for (int num : primeNums)
    total += num;
```

What happens when you try to modify the value of the loop variable in an enhanced `for` loop? The result depends on whether you're iterating through a collection of primitive values or objects. If you're iterating through an array of primitive values, manipulation of the loop variable will never change the value of the array being iterated because the primitive values are passed by value to the loop variable in an enhanced `for` loop.

When you iterate through a collection of objects, the value of the collection is passed by reference to the loop variable. Therefore, if the value of the loop variable is manipulated by executing methods on it, the modified value will be reflected in the collection of objects being iterated:

```
StringBuilder myArr[] = {
                new StringBuilder("Java"),
                new StringBuilder("Loop")
                 };

for (StringBuilder val : myArr)
    System.out.println(val);

for (StringBuilder val : myArr)
    val.append("Oracle");

for (StringBuilder val : myArr)
    System.out.println(val);
```

Iterates through array myArr and prints Java and Loop

Appends Oracle to value referred by loop variable val

Iterates through array myArr and prints JavaOracle and LoopOracle

The output of the previous code is:

```
Java
Loop
JavaOracle
LoopOracle
```

Let's modify the previous code. Instead of calling the method `append` on the loop variable `val`, let's assign to it another `StringBuilder` object. In this case, the original elements of the array being iterated will not be affected and will remain the same:

```
StringBuilder myArr[] = {
                new StringBuilder("Java"),
                new StringBuilder("Loop")
                 };

or (StringBuilder val : myArr)
    System.out.println (val);

for (StringBuilder val : myArr)
    val = new StringBuilder("Oracle");

for (StringBuilder val : myArr)
    System.out.println (val);
```

Iterates through array myArr and prints Java and Loop

Assigns new StringBuilder object to reference variable val with value Oracle

Iterates through array myArray and still prints Java and Loop

The output of previous code is:

```
Java
Loop
Java
Loop
```

EXAM TIP Watch out for code that uses an enhanced `for` loop and its loop variable to change the values of elements in the collection that it iterates. This behavior often serves as food for thought for the exam authors.

5.4.1 Limitations of the enhanced for loop

Though a `for`-each loop is a good choice for iterating through collections and arrays, it can't be used in some places.

CAN'T BE USED TO INITIALIZE AN ARRAY AND MODIFY ITS ELEMENTS

Can you use an enhanced `for` loop in place of the regular `for` loop in the following code?

```
int[] myArray = new int[5];                      // Declare array
for (int i=0; i<myArray.length; ++i) {           // count=length of the array
    myArray[i] = i;                              // Initialize array elements
    if ((myArray[i]%2)==0)
        myArray[i] = 20;                         // Modify array elements
}
```

The simple answer is "no." Although you can define a "counter" outside of the enhanced `for` loop and use it to initialize and modify the array elements, this approach defeats the purpose of the `for`-each loop. The existing `for` loop is easier to use in this case.

CAN'T BE USED TO DELETE OR REMOVE THE ELEMENTS OF A COLLECTION

Because the `for` loop hides the *iterator* used to iterate through the elements of a collection, you can't use it to remove or delete the existing collection values because you can't call the `remove` method.

If you assign a `null` value to the loop variable, it won't remove the element from a collection:

```
ArrayList<StringBuilder> myList= new ArrayList<>();
myList.add(new StringBuilder("One"));
myList.add(new StringBuilder("Two"));

for (StringBuilder val : myList)
    System.out.println (val);

for (StringBuilder val : myList)
    val = null;                    // Doesn't remove an object from list; sets value of loop variable to null

for (StringBuilder val : myList)
    System.out.println(val);
```

The output of the previous code is:

```
One
Two
One
Two
```

CAN'T BE USED TO ITERATE OVER MULTIPLE COLLECTIONS OR ARRAYS IN THE SAME LOOP

Though it's perfectly fine for you to iterate through nested collections or arrays using a `for` loop, you can't iterate over multiple collections or arrays in the same `for`-each loop because the `for`-each loop allows for the creation of only one looping variable. Unlike the regular `for` loop, you can't define multiple looping variables in a `for`-each loop.

EXAM TIP Use the `for`-each loop to iterate arrays and collections. Don't use it to initialize, modify, or filter them.

5.4.2 Nested enhanced for loop

First of all, working with a nested collection is not the same as working with a nested loop. A nested loop can also work with unrelated collections.

As discussed in section 5.3.4, loops defined within another loop are called *nested loops*. The loop that defines another loop within itself is called the *outer loop*, and the loop that's defined within another loop is called the *inner loop*. Theoretically, the level of nesting for any of the loops has no limits, including the enhanced `for` loop.

In this section, we'll work with three nested enhanced `for` loops. You can compare a three-level nested loop with a clock that has hour, minute, and second hands. The second hand of the clock completes a full circle each minute. Similarly, the minute hand completes a full circle each hour. This comparison is shown in figure 5.14.

The following is a coding example of the nested, enhanced `for` loop, which I discussed in a previous section:

```
ArrayList<String> exams= new ArrayList<String>();
exams.add("Java"); exams.add("Oracle");

ArrayList<String> levels= new ArrayList<String>();
levels.add("Basic"); levels.add("Advanced");

ArrayList<String> grades= new ArrayList<String>();
grades.add("Pass"); grades.add("Fail");

for (String exam : exams)
    for (String level : levels)
        for (String grade : grades)
            System.out.println(exam+":"+level+":"+grade);
```

First ArrayList

Second ArrayList

Third ArrayList

❶ Outermost loop

❷ Inner nested loop

❸ Innermost nested loop

An inner loop in a nested loop executes for each iteration of its outer loop. The previous example defines three enhanced `for` loops: the outermost loop at ❶, the inner nested loop at ❷, and the innermost loop at ❸. The complete innermost loop at ❸ executes for each iteration of its immediate outer loop defined at ❷. Similarly, the complete inner loop defined at ❷ executes for each iteration of its

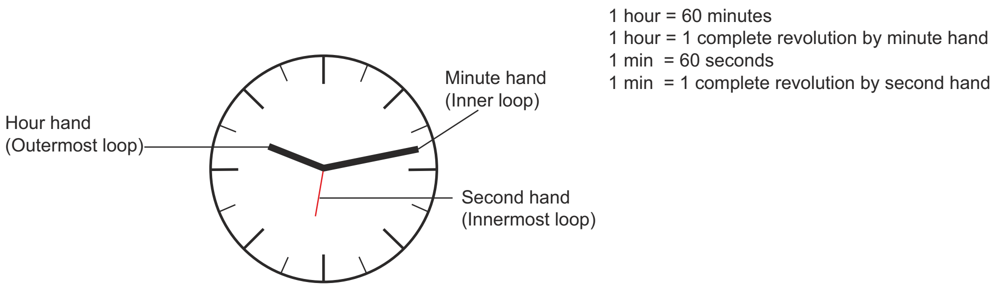
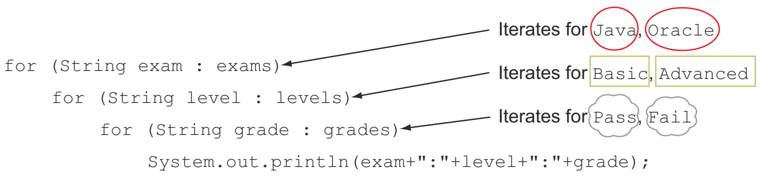

1 hour = 60 minutes
1 hour = 1 complete revolution by minute hand
1 min = 60 seconds
1 min = 1 complete revolution by second hand
Minute hand (Inner loop)
Hour hand (Outermost loop)
Second hand (Innermost loop)

Figure 5.14 Comparison between a clock with three hands and the levels of a nested `for` loop



```
for (String exam : exams)
    for (String level : levels)
        for (String grade : grades)
            System.out.println(exam+":"+level+":"+grade);
```

Figure 5.15 Nested `for` loop with the loop values for which each of these nested loop iterates

immediate outer loop defined at ❶. Figure 5.15 shows the loop values for which all of these loops iterate.

The output of the previous code is as follows:

```
Java:Basic:Pass
Java:Basic:Fail
Java:Advanced:Pass
Java:Advanced:Fail
Oracle:Basic:Pass
Oracle:Basic:Fail
Oracle:Advanced:Pass
Oracle:Advanced:Fail
```

EXAM TIP A nested loop executes all its iterations for each single iteration of its immediate outer loop.

Apart from the `for` loops, the other looping statements on the exam are `while` and `do-while`, which are discussed in the next section.

5.5 The while and do-while loops

[5.1] Create and use while loops

[5.3] Create and use do-while loops

You'll learn about `while` and `do-while` loops in this section. Both of these loops execute a set of statements as long as their condition evaluates to `true`. Both of these loops work in exactly the same manner, except for one difference: the while loops checks its condition before evaluating its loop body, and the do-while loop checks its condition after executing the statements defined in its loop body.

Does this difference in behavior make a difference in their execution? Yes, it does, and in this section, you'll see how.

5.5.1 The while loop

A while loop is used to repeatedly execute a set of statements as long as its condition evaluates to true. This loop checks the condition *before* it starts the execution of the statement.

For example, at famous fast-food chain Superfast Burgers, an employee may be instructed to prepare burgers as long as buns are available. In this example, the availability of buns is the while condition and the preparation of burgers is the while's loop body. We can represent this in code as follows:

```
boolean bunsAvailable = true;
while (bunsAvailable) {
    ... prepare burger ...
    if (noMoreBuns)
        bunsAvailable = false;
}
```

The previous example is for demonstration purposes only, because the loop body isn't completely defined. The condition used in the while loop to check whether or not to execute the loop body again should evaluate to false at some point in time; otherwise, the loop will execute indefinitely. The value of this loop variable may be changed by the while loop or by another method if it's an instance or a static variable.

The while loop accepts arguments of type boolean or Boolean. Because the Boolean wrapper class isn't covered in the OCA Java SE 7 Programmer I exam, I won't cover it any further. In the previous code, the loop body checks whether more buns are available. If none are available, it sets the value of the variable bunsAvailable to false. The loop body doesn't execute for the next iteration because bunsAvailable evaluates to false.

The execution of the previous while loop is shown in figure 5.16 as a simple flowchart to help you understand the concept better.

Now, let's examine another simple example that uses the while loop:

```
int num = 9;
boolean divisibleBy7 = false;
while (!divisibleBy7) {
    System.out.println(num);
    if (num % 7 == 0) divisibleBy7 = true;
    --num;
}
```

The output of this code is as follows:

```
9
8
7
```

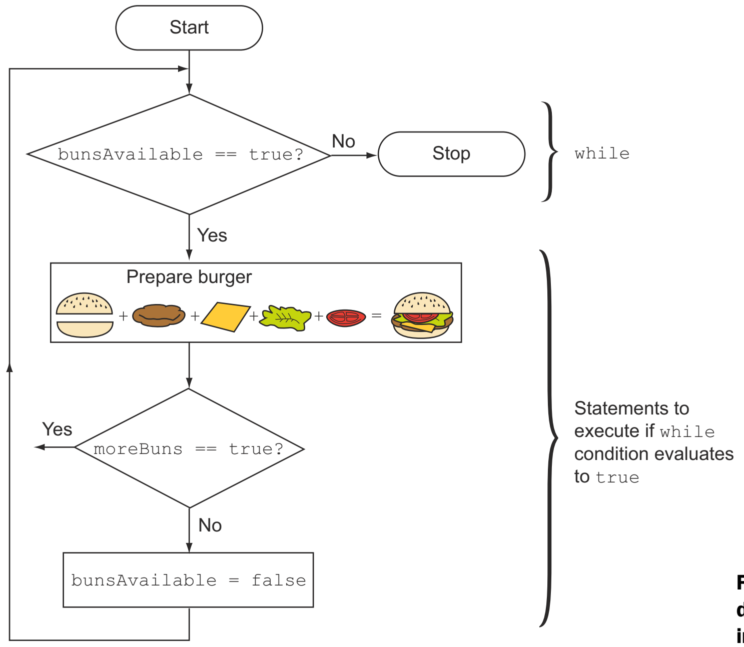


Figure 5.16 A flowchart depicting the flow of code in a while loop

What happens if you change the code as follows (changes in bold):

```
int num = 9;
boolean divisibleBy7 = true;
while (divisibleBy7 == false) {
    System.out.println(num);
    if (num % 7 == 0) divisibleBy7 = true;
    --num;
}
```

The code won't enter the loop because the condition `divisibleBy7==false` isn't `true`.

5.5.2 The do-while loop

A `do-while` loop is used to repeatedly execute a set of statements until the condition that it uses evaluates to `false`. This loop checks the condition *after* it completes the execution of all the statements in its loop body.

You could compare this structure to a software application that displays a menu at startup. Each menu option will execute a set of steps and redisplay the menu. The last menu option is "exit," which exits the application and does not redisplay the menu:

```
boolean exitSelected = false;
do {
    String selectedOption = displayMenuToUser();
    if (selectedOption.equals("exit"))
        exitSelected = true;
    else
        executeCommand(selectedOption);
} while (exitSelected == false);
```

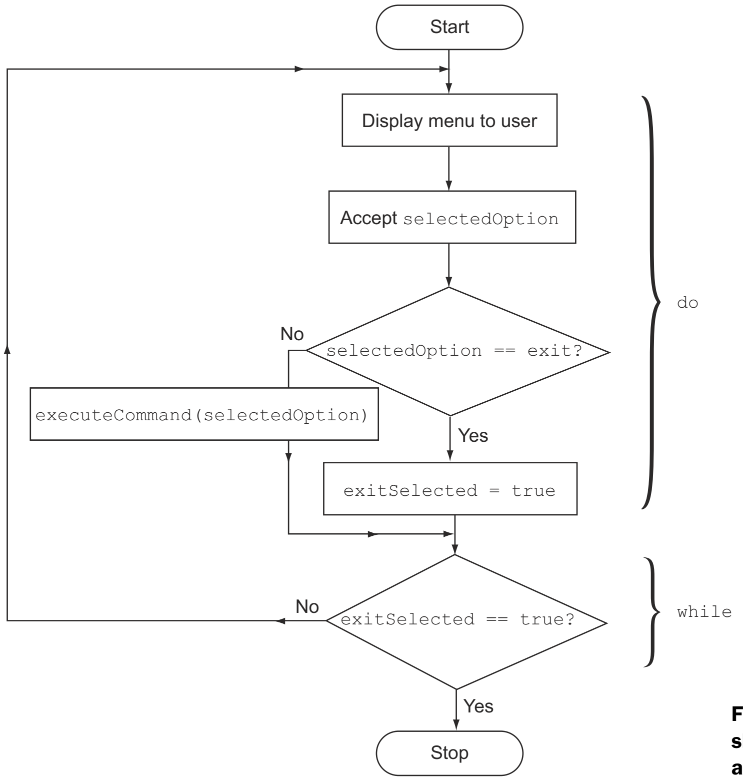


Figure 5.17 Flowchart showing the flow of code in a `do-while` loop

The previous code is represented by a simple flowchart in figure 5.17 that will help you to better understand the code.

The previous example is for demonstration purposes only because the methods used in the `do-while` loop aren't defined. As discussed in the previous section on `while` loops, the condition that's used in the `do-while` loop to check whether or not to execute the loop body again should evaluate to `false` at some point in time, or the loop will execute indefinitely. The value of this loop variable may be changed by the `while` loop or by another method, if it's an instance or `static` variable.

NOTE Don't forget to use a semicolon (`;`) to end the `do-while` loop after specifying its condition. Even some experienced programmers overlook this step!

The `do-while` loop accepts arguments of type `boolean` or `Boolean`. Because the `Boolean` wrapper class isn't covered in the OCA Java SE 7 Programmer I exam, I won't cover it any further.

Let's modify the example used in section 5.5.1 to use the `do-while` loop instead of a `while` loop, as follows:

```
int num = 9;
boolean divisibleBy7 = false;
do {
    System.out.println(num);
    if (num % 7 == 0) divisibleBy7 = true;
    num--;
} while (divisibleBy7 == false);
```

The output of this code is as follows:

```
9
8
7
```

What happens if you change the code as follows (changes in bold):

```
int num = 9;
boolean divisibleBy7 = true;
do {
    System.out.println(num);
    if (num % 7 == 0) divisibleBy7 = true;
    num--;
} while (divisibleBy7 == false);
```

The output of the previous code is as follows:

```
9
```

The `do-while` loop executes once, even though the condition specified in the `do-while` loop evaluates to `false` because the condition is evaluated at the end of execution of the loop body.

5.5.3 While and do-while block, expression, and nesting rules

You can use the curly braces {} with `while` and `do-while` loops to define multiple lines of code to execute for every iteration. Without the use of curly braces, only the first line of code will be considered a part of the `while` or `do-while` loop, as specified in the `if-else` construct in section 5.1.3.

Similarly, the rules that define an appropriate expression to be passed to `while` and `do-while` loops are as for the `if-else` construct in section 5.1.4. Also, the rules for defining nested `while` and `do-while` loops are the same as for an `if-else` construct in section 5.1.5.

5.6 Comparing loop constructs

[5.4] Compare loop constructs

In this section, I'll discuss the differences and similarities between the following looping constructs: `do-while`, `while`, `for`, and enhanced `for`.

5.6.1 Comparing do-while and while loops

Both `do-while` and `while` loops execute a set of statements until their termination condition evaluates to `false`. The only difference between these two statements is that the `do-while` loop executes the code at least once, even if the condition evaluates to

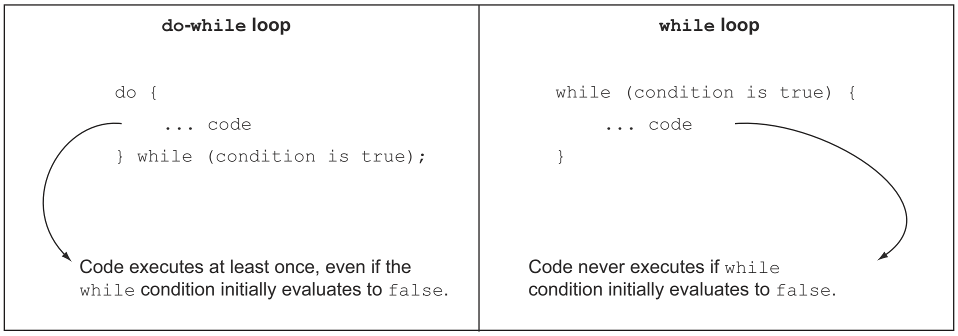

do-while loop
do {
... code
} while (condition is true);
Code executes at least once, even if the while condition initially evaluates to false.
while loop
while (condition is true) {
... code
}
Code never executes if while condition initially evaluates to false.

Figure 5.18 Comparing `do-while` and `while` loops

`false`. The `do-while` loop evaluates the termination condition *after* executing the statements, whereas the `while` loop evaluates the termination condition *before* executing its statements.

The forms taken by these statements are depicted in figure 5.18.

What do you think the output of the following code is?

```
int num=10;
do {
    num++;                    // Value of num incremented to 11 is incremented by 1
} while (++num > 20);         // Evaluate condition ++num > 20
System.out.println (num);
```

The output of the previous code is as follows:

```
12
```

What do you think the output of the following code is?

```
int num=10;
while (++num > 20) {          // Because 11 isn't > 20, num++ doesn't execute.
    num++;
}
System.out.println(num);
```

The output of the previous code is as follows

```
11
```

5.6.2 Comparing for and enhanced for loops

The regular `for` loop, although cumbersome to use, is much more powerful than the enhanced `for` loop (as mentioned in section 5.4.1):

- The enhanced `for` loop can't be used to initialize an array and modify its elements.
- The enhanced `for` loop can't be used to delete the elements of a collection.
- The enhanced `for` loop can't be used to iterate over multiple collections or arrays in the same loop.

5.6.3 Comparing for and while loops

You should *try* to use a `for` loop when you know the number of iterations—for example, when you're iterating through a collection or an array, or when you're executing a loop for a fixed number of times, say to "ping" a server five times.

You should *try* to use a `do-while` or a `while` loop when you don't know the number of iterations beforehand, and when the number of iterations depends on a condition being `true`—for example, when accepting passport renewal applications from applicants until there are no more applicants. In this case, you'd be unaware of the number of applicants who have submitted their applications on a given day.

5.7 Loop statements: break and continue

[5.5] Use break and continue

Imagine that you've defined a loop to iterate through a list of managers, and you're looking for at least one manager whose name starts with the letter *D*. You'd like to exit the loop after you find the first match, but how? You can do this by using the `break` statement in your loop.

Now imagine that you want to iterate through all of the folders on your laptop and scan any files larger than 10 MB for viruses. If all those files are found to be okay, you want to upload them to a server. But what if you'd like to skip the steps of *virus checking* and *file uploading* for file sizes less than 10 MB, yet still proceed with the remaining files on your laptop? You can! You'd use the `continue` statement in your loop.

In this section, I'll discuss the `break` and `continue` statements, which you can use to exit a loop completely or to skip the remaining statements in a loop iteration. At the end of this section, I'll discuss labeled statements.

5.7.1 The break statement

The `break` statement is used to *exit*—or *break out of*—the `for`, `for`-each, `do`, and `do-while` loops, as well as `switch` constructs. Alternatively, the `continue` statement can be used to skip the remaining steps in the current iteration and start with the next loop iteration.

The difference between these statements can be best demonstrated with an example. You could use the following code to browse and print all of the values of a `String` array:

```
String[] programmers = {"Paul", "Shreya", "Selvan", "Harry"};
for (String name : programmers) {
    System.out.println(name);
}
```

The output of the previous code is as follows:

```
Paul
Shreya
Selvan
Harry
```

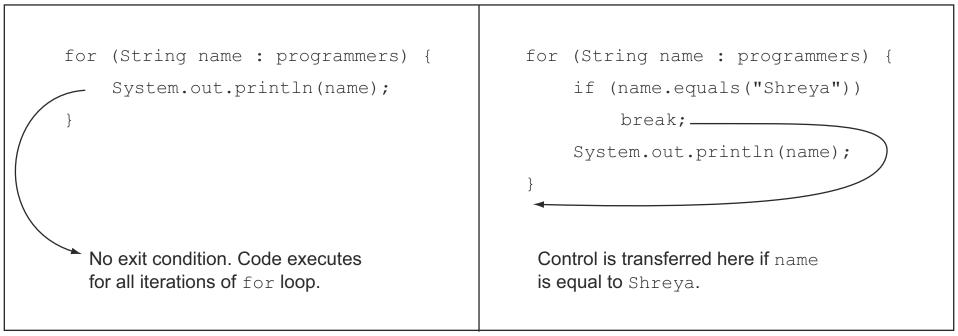

for (String name : programmers) {
System.out.println(name);
}
No exit condition. Code executes for all iterations of for loop.
for (String name : programmers) {
if (name.equals("Shreya"))
break;
System.out.println(name);
}
Control is transferred here if name is equal to Shreya.

Figure 5.19 The flow of control when the `break` statement executes within a loop

Let's modify the previous code to exit the loop when the array value is equal to `Shreya`. Here's the required code:

```
String[] programmers = {"Paul", "Shreya", "Selvan", "Harry"};
for (String name : programmers) {
    if (name.equals("Shreya"))
        break;                                  Break out of the loop
    System.out.println(name);
}
```

The output of the previous code is as follows:

```
Paul
```

As soon as a loop encounters a `break`, it exits the loop. Hence, only the first value of this array—that is, `Paul`—is printed. As mentioned in the section on the `switch` construct, the `break` statement can be defined after every `case` in order for the control to exit the `switch` construct once it finds a matching case.

The previous code snippets are depicted in figure 5.19, which shows the transfer of control upon execution of the `break` statement.

When you use the `break` statement with nested loops, it exits the inner loop. The next Twist in the Tale exercise looks at a small code snippet to see how the control transfers when you use a `break` statement in nested `for` loops (answers in the appendix).

Twist in the Tale 5.4

Let's modify the code used in the previous example as follows. What is the output of this code?

```
String[] programmers = {"Outer", "Inner"};
for (String outer : programmers) {
    for (String inner : programmers) {
        if (inner.equals("Inner"))
            break;
        System.out.print(inner + ":");
    }
}
```

a `Outer:Outer:`

b `Outer:Inner:Outer:Inner:`

c `Outer:`

d `Outer:Inner:`

e `Inner:Inner:`

5.7.2 The continue statement

The `continue` statement is used to skip the remaining steps in the current iteration and start with the next loop iteration. Let's replace the `break` statement in the previous example with `continue` and examine its output:

```
String[] programmers = {"Paul", "Shreya", "Selvan", "Harry"};
for (String name : programmers) {
    if (name.equals("Shreya"))
        continue;                         <—  Skip the remaining
    System.out.println(name);                 loop statements
}
```

The output of the previous code is as follows:

```
Paul
Selvan
Harry
```

As soon as a loop encounters `continue`, it exits the current iteration of the loop. In this example, it skips the printing step for the array value `Shreya`. Unlike the `break` statement, `continue` doesn't exit the loop—it restarts with the next loop iteration, printing the remaining array values (that is, `Selvan` and `Harry`).

When you use the `continue` statement with nested loops, it exits the current iteration of the inner loop.

Figure 5.20 compares how the control transfers out of the loop and to the next iteration when `break` and `continue` statements are used.

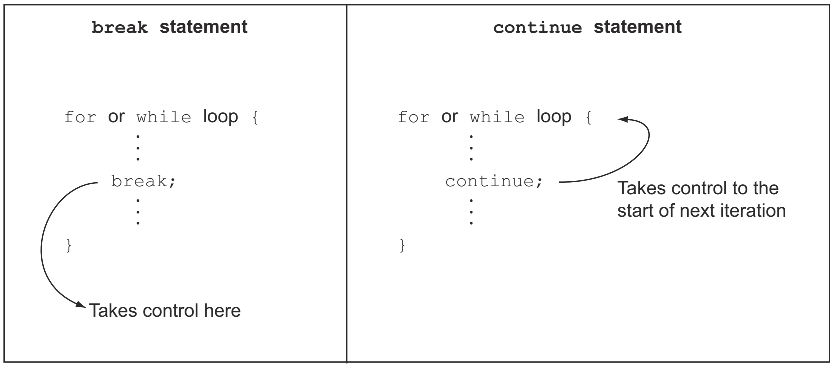


Figure 5.20 Comparing the flow of control when using `break` and `continue` statements in a loop

5.7.3 *Labeled statements*

In Java, you can add labels to the following types of statements:

- A code block defined using `{}`
- All looping statements (`for`, enhanced `for`, `while`, `do-while`)
- Conditional constructs (`if` and `switch` statements)
- Expressions
- Assignments
- `return` statements
- `try` blocks
- `throws` statements

An example of a labeled loop is given here:

```
String[] programmers = {"Outer", "Inner"};
outer:
for (int i = 0; i < programmers.length; i++) {
}
```

You can't add labels to declarations. The following labeled declaration won't compile:

```
outer :
    int[] myArray = {1,2,3};
```

Variable declaration that fails compilation

It's interesting to note that the previous declaration can be defined within a block statement, as follows:

```
outer : {
    int[] myArray = {1,2,3};
}
```

Start definition of block

Variable declaration, compiles

End block

LABELED BREAK STATEMENTS

You can use a labeled `break` statement to exit an outer loop. Here's an example:

```
String[] programmers = {"Outer", "Inner"};
outer:
for (String outer : programmers) {
    for (String inner : programmers) {
        if (inner.equals("Inner"))
            break outer;
        System.out.print(inner + ":");
    }
}
```

Exits the outer loop, marked with label outer

The output of the previous code is:

```
Outer:
```

When this code executes `break outer;`, control transfers to the line of text that marks the end of this block. It doesn't transfer control to the label `outer`.

LABELED CONTINUE STATEMENTS

You can use a labeled `continue` statement to skip an iteration of the outer loop. Here's an example:

```
String[] programmers = {"Paul", "Shreya", "Selvan", "Harry"};
outer:
for (String name1 : programmers) {
    for (String name : programmers) {
        if (name.equals("Shreya"))
            continue outer;
        System.out.println(name);
    }
}
```

Skips remaining code for current iteration of outer loop and starts with its next iteration

The output of the previous code is:

```
Paul
Paul
Paul
Paul
```

5.8 Summary

We started this chapter with the selection statements `if` and `switch`. We covered the different flavors of the `if` construct. Then we looked at the `switch` construct, which accepts a limited set of argument types including `byte`, `char`, `short`, `int`, and `String`. The humble `if-else` construct can define virtually any set of simple or complicated conditions.

We also saw how you can execute your code using all types of loops: `for`, `for`-each, `do`, and `do-while`. The `for`, `do`, and `do-while` loops have been around since the Java language was first introduced, whereas the enhanced `for` loop (the `for`-each loop) was added to the language as of Java version 5.0. I recommend that you use the `for`-each loop to iterate through arrays and collections.

At the end of this chapter, we looked at the `break` and `continue` statements. You use the `break` statement to *exit*—or *break out of*—a `for`, `for`-each, `do`, `do-while`, or `switch` construct. You use the `continue` statement to skip the remaining steps in the current iteration and start with the next loop iteration.

5.9 Review notes

`if` and `if-else` constructs:

- The `if` statement enables you to execute a set of statements in your code based on the result of a condition, which should evaluate to a `boolean` or `Boolean` value.
- The multiple flavors of an `if` statement are `if`, `if-else`, and `if-else-if-else`.
- The `if` construct doesn't use the keyword *then* to define code to execute when an `if` condition evaluates to `true`. The *then* part of the `if` construct follows the `if` condition.
- An `if` construct may or may not define its `else` part.
- The `else` part of an `if` construct can't exist without the definition of its *then* part.
- It's easy to get confused with the common *if-else* syntax used in other programming languages. The `if-elsif` and `if-elseif` statements aren't used in Java to define `if-else-if-else` constructs. The correct keywords are `if` and `else`.

- You can execute a single statement or a block of statements for corresponding `true` and `false` conditions. A pair of braces marks a block of statements: {}.
- If an `if` construct doesn't use {} to define a block of code to execute for its *then* or `else` part, only the first line of code is part of the `if` construct.
- An assignment of a `boolean` variable can also be passed as an argument to the `if` construct. It's valid because the resultant value is `boolean`, which is accepted by `if` constructs.
- Theoretically, nested `if` and `if-else` constructs have no limits on their levels. When using nested `if-else` constructs, be careful about matching the `else` part with the right `if` part.

`switch` statements:

- A `switch` statement is used to compare the value of a variable with multiple predefined values.
- A `switch` statement accepts arguments of type `char`, `byte`, `short`, `int`, and `String`. It also accepts arguments of wrapper classes: `Character`, `Byte`, `Short`, `Integer`, and `Enum`. These wrapper classes aren't on the OCA Java SE 7 Programmer I exam.
- A `switch` statement can be compared with multiple related `if-else-if-else` constructs.
- You can pass an expression as an argument to a `switch` statement, as long as the type of the expression is one of the acceptable data types.
- The `case` value should be a compile-time constant, assignable to the argument passed to the `switch` statement.
- The `case` value can't be the literal value `null`.
- The `case` value can define expressions that use literal values; that is, they can be evaluated at compile time, as in `7+2`.
- One code block can be defined to execute for multiple `case` values in a `switch` statement.
- A `break` statement is used to exit a `switch` construct once a matching case is found and the required code statements have executed.
- In absence of the `break` statement, control will *fall through* all the remaining case values in a `switch` statement until the first `break` statement is found, evaluating the code for the `case` statements in order.

`for` loops:

- A `for` loop is usually used to execute a set of statements a fixed number of times.
- A `for` loop defines three types of statements separated by semicolons (;): initialization statements, termination condition, and update clause.
- The definition of any of the three `for` statements—initialization statements, termination condition, and update clause—is optional. For example, `for (;;);`

and `for (;;){}` are valid code for defining a `for` loop. Also, defining any one of these statements is also valid code.

- An initialization block executes only once. A `for` loop can declare and initialize multiple variables in its initialization block, but the variables that it declares should be of the same type.
- The termination condition is evaluated once, for all the iterations, before the statements defined within the body of the loop are executed.
- The `for` loop terminates when the termination condition evaluates to `false`.
- The update block is usually used to increment or decrement the value of the variables that are defined in the initialization block. It can also execute multiple other statements, including method calls.
- Nested `for` loops have no limits on levels.
- Nested `for` loops are frequently used to work with multidimensional arrays.

Enhanced `for` loops:

- The enhanced `for` loop is also called the `for`-each loop.
- The enhanced `for` loop offers some benefits over the regular `for` loop, but it's not as flexible as the regular `for` loop.
- The enhanced `for` loop offers simple syntax to iterate through a collection of values—an array, `ArrayList`, or other classes from Java's Collection framework that store a collection of values.
- The enhanced `for` loop can't be used to initialize an array and modify its elements.
- The enhanced `for` loop can't be used to delete the elements of a collection.
- The enhanced `for` loop can't be used to iterate over multiple collections or arrays in the same loop.
- Nested enhanced `for` loops have no limits on levels.

`while` and `do-while` loops:

- A `while` loop is used to keep executing a set of statements until the condition that it uses evaluates to `false`. This loop checks the condition *before* it starts the execution of the statement.
- A `do-while` loop is used to keep executing a set of statements until the condition that it uses evaluates to `false`. This loop checks the condition *after* it completes the execution of all the statements in its loop body.
- The levels of nested `do-while` or `while` loops have no limitations.
- Both `do-while` and `while` loops can define either a single line of code or a code block to execute. The latter is defined by using curly braces, `{}`.

Comparing loop constructs:

- Both the `do-while` and `while` loops execute a set of statements until the termination condition evaluates to `false`. The only difference between these two statements is that the do-while loop executes the code at least once, even if the condition evaluates to false.

- The regular for loop, though cumbersome to use, is much more powerful than the enhanced for loop.
- The enhanced for loop can't be used to initialize an array and modify its elements. The enhanced for loop can't be used to delete or remove the elements of a collection.
- The enhanced for loop can't be used to iterate over multiple collections or arrays in the same loop.
- You should try to use a for loop when you know the number of iterations—for example, iterating through a collection or an array, or executing a loop for a fixed number of times, say to "ping" a server five times.
- You should try to use a do-while or a while loop when you don't know the number of iterations beforehand and the number of iterations depends on a condition being true—for example, accepting passport renewal applications until all applicants have been attended to.

Loop statements (break and continue):

- The break statement is used to *exit*—or *break out of*—the for, for-each, do, and do-while loops and the switch construct.
- The continue statement is used to skip the remaining steps in the current iteration and start with the next loop iteration. The continue statement works with the for, for-each, do, and do-while loops and the switch construct.
- When you use the break statement with nested loops, it exits the inner loop.
- When you use the continue statement with nested loops, it exits the current iteration of the inner loop.

Labeled statements:

- You can add labels to a code block defined using braces, {}, all looping statements (for, enhanced for loop, while, do-while), conditional constructs (if and switch statements), expressions and assignments, return statements, try blocks, and throws statements.
- You can't add labels to declarations of variables.
- You can use a labeled break statement to exit an outer loop.
- You can use a labeled continue statement to skip the iteration of the outer loop.

5.10 Sample exam questions

Q5-1. What's the output of the following code?

```
class Loop2 {
    public static void main(String[] args) {
        int i = 10;
        do
```

```
            while (i < 15)
                i = i + 20;
        while (i < 2);
        System.out.println(i);
    }
}
```

- **a** 10
- **b** 30
- **c** 31
- **d** 32

Q5-2. What's the output of the following code?

```
class Loop2 {
    public static void main(String[] args) {
        int i = 10;
        do
            while (i++ < 15)
                i = i + 20;
        while (i < 2);
        System.out.println(i);
    }
}
```

- **a** 10
- **b** 30
- **c** 31
- **d** 32

Q5-3. Which of the following statements is true?

- **a** The enhanced `for` loop can't be used within a regular `for` loop.
- **b** The enhanced `for` loop can't be used within a `while` loop.
- **c** The enhanced `for` loop can be used within a `do-while` loop.
- **d** The enhanced `for` loop can't be used within a `switch` construct.
- **e** All of the above statements are false.

Q5-4. What's the output of the following code?

```
int a =  10;
if (a++ > 10) {
    System.out.println("true");
}
{
    System.out.println("false");
}
System.out.println("ABC");
```

- **a**
```
true
false
ABC
```

b
```
false
ABC
```

c
```
true
ABC
```

d Compilation error

Q5-5. Given the following code, which of the following lines of code can individually replace the `//INSERT CODE HERE` line so that the code compiles successfully?

```
class EJavaGuru {
    public static int getVal() {
        return 100;
    }
    public static void main(String args[]) {
        int num = 10;
        final int num2 = 20;
        switch (num) {
            // INSERT CODE HERE
            break;
            default: System.out.println("default");
        }
    }
}
```

a `case 10*3: System.out.println(2);`

b `case num: System.out.println(3);`

c `case 10/3: System.out.println(4);`

d `case num2: System.out.println(5);`

Q5-6. What's the output of the following code?

```
class EJavaGuru {
    public static void main(String args[]) {
        int num = 20;
        final int num2;
        num2 = 20;

        switch (num) {
            default: System.out.println("default");
            case num2: System.out.println(4);
            break;
        }
    }
}
```

a
```
default
```

b
```
default
4
```

c
```
4
```

d Compilation error

Q5-7. What's the output of the following code?

```
class EJavaGuru {
    public static void main(String args[]) {
        int num = 120;

        switch (num) {
            default: System.out.println("default");
            case 0: System.out.println("case1");
            case 10*2-20: System.out.println("case2");
            break;
        }
    }
}
```

- **a**
  ```
  default
  case1
  case2
  ```
- **b**
  ```
  case1
  case2
  ```
- **c** `case2`
- **d** Compilation error
- **e** Runtime exception

Q5-8. What's the output of the following code?

```
class EJavaGuru3 {
    public static void main(String args[]) {
        byte foo = 120;
        switch (foo) {
            default: System.out.println("ejavaguru"); break;
            case 2: System.out.println("e"); break;
            case 120: System.out.println("ejava");
            case 121: System.out.println("enum");
            case 127: System.out.println("guru"); break;
        }
    }
}
```

- **a**
  ```
  ejava
  enum
  guru
  ```
- **b** `ejava`
- **c**
  ```
  ejavaguru
  e
  ```
- **d**
  ```
  ejava
  enum
  guru
  ejavaguru
  ```

Q5-9. What's the output of the following code?

```
class EJavaGuru4 {
    public static void main(String args[]) {
```

```
        boolean myVal = false;
        if (myVal=true)
        for (int i = 0; i < 2; i++) System.out.println(i);
        else System.out.println("else");
    }
}
```

- **a** `else`
- **b** `0`
  `1`
  `2`
- **c** `0`
  `1`
- **d** Compilation error

Q5-10. What's the output of the following code?

```
class EJavaGuru5 {
    public static void main(String args[]) {
        int i = 0;
        for (; i < 2; i=i+5) {
            if (i < 5) continue;
            System.out.println(i);
        }
        System.out.println(i);
    }
}
```

- **a** Compilation error
- **b** `0`
  `5`
- **c** `0`
  `5`
  `10`
- **d** `10`
- **e** `0`
  `1`
  `5`
- **f** `5`

5.11 Answers to sample exam questions

Q5-1. What's the output of the following code?

```
class Loop2 {
    public static void main(String[] args) {
        int i = 10;
        do
            while (i < 15)
                i = i + 20;
        while (i < 2);
        System.out.println(i);
    }
}
```

- **a** 10
- **b** **30**
- **c** 31
- **d** 32

Answer: b

Explanation: The condition specified in the do-while loop evaluates to false (because 10<2 evaluates to false). But the control enters the do-while loop because the do-while loop executes at least once—its condition is checked at the end of the loop. The while evaluates to true for the first iteration and adds 20 to i, making it 30. The while loop doesn't execute for the second time. Hence, the value of the variable i at the end of the execution of the previous code is 30.

Q5-2. What's the output of the following code?

```
class Loop2 {
    public static void main(String[] args) {
        int i = 10;
        do
            while (i++ < 15)
                i = i + 20;
        while (i < 2);
        System.out.println(i);
    }
}
```

- **a** 10
- **b** 30
- **c** 31
- **d** **32**

Answer: d

Explanation: If you've attempted to answer question 5-1, it's likely that you would select the same answer for this question, too. I've deliberately used the same question text and variable names (with a small difference) because you may encounter a similar pattern in the OCA Java SE 7 Programmer I exam. This question includes one difference: unlike question 5-1, it uses a postfix unary operator in the while condition.

The condition specified in the do-while loop evaluates to false (because 10<2 evaluates to false). But the control enters the do-while loop because the do-while loop executes at least once—its condition is checked at the end of the loop. This question prints outs 32, not 30, because the condition specified in the while loop (which has an increment operator) executes twice.

In this question, the while loop condition executes twice. For the first evaluation, i++ < 15 (that is, 10<15) returns true and increments the value of variable i by 1 (due to the postfix increment operator). The loop body modifies the value of i to 31. The second condition evaluates `i++<15` (that is, `31<15`) to `false`. But due to the postfix increment operator value of `i`, it increments to `32`. The final value is printed as `32`.

Q5-3. Which of the following statements is true?

- **a** The enhanced `for` loop can't be used within a regular `for` loop.
- **b** The enhanced `for` loop can't be used within a `while` loop.
- **c** **The enhanced `for` loop can be used within a `do-while` loop.**
- **d** The enhanced `for` loop can't be used within a `switch` construct.
- **e** All of the above statements are false.

Answer: c

Explanation: The enhanced `for` loop can be used within all types of looping and conditional constructs. Notice the use of "can" and "can't" in the answer options. It's important to take note of these subtle differences.

Q5-4. What's the output of the following code?

```
int a =  10;
if (a++ > 10) {
    System.out.println("true");
}
{
    System.out.println("false");
}
System.out.println("ABC");
```

- **a**
  ```
  true
  false
  ABC
  ```
- **b**
  ```
  false
  ABC
  ```
- **c**
  ```
  true
  ABC
  ```
- **d** Compilation error

Answer: b

Explanation: First of all, the code has no compilation errors. This question has a trick—the following code snippet isn't part of the `if` construct:

```
{
    System.out.println("false");
}
```

Hence, the value `false` will print no matter what, regardless of whether the condition in the `if` construct evaluates to `true` or `false`.

Because the opening and closing braces for this code snippet are placed right after the `if` construct, it leads us to believe that this code snippet is the `else` part of the `if` construct. Also, note that an `if` construct uses the keyword `else` to define the `else` part. This keyword is missing in this question.

The `if` condition (that is, `a++ > 10`) evaluates to `false` because the postfix increment operator (`a++`) increments the value of the variable `a` immediately after its earlier value is used. `10` isn't greater than `10` so this condition evaluates to `false`.

Q5-5. Given the following code, which of the following lines of code can individually replace the `//INSERT CODE HERE` line so that the code compiles successfully?

```
class EJavaGuru {
    public static int getVal() {
        return 100;
    }
    public static void main(String args[]) {
        int num = 10;
        final int num2 = 20;
        switch (num) {
            // INSERT CODE HERE
            break;
            default: System.out.println("default");
        }
    }
}
```

a **`case 10*3: System.out.println(2);`**

b `case num: System.out.println(3);`

c **`case 10/3: System.out.println(4);`**

d **`case num2: System.out.println(5);`**

Answer: a, c, d

Explanation: Option (a) is correct. Compile-time constants, including expressions, are permissible in the `case` labels.

Option (b) is incorrect. The `case` labels should be compile-time constants. A non-`final` variable isn't a compile-time constant because it can be reassigned a value during the course of a class's execution. Although the previous class doesn't assign a value to it, the compiler still treats it as a changeable variable.

Option (c) is correct. The value specified in the `case` labels should be assignable to the variable used in the `switch` construct. You may think that `10/3` will return a decimal number, which can't be assigned to the variable `num`, but this operation discards the decimal part and compares `3` with the variable `num`.

Option (d) is correct. The variable `num2` is defined as a `final` variable and assigned a value on the same line of code, with its declaration. Hence, it's considered to be a compile-time constant.

Q5-6. What's the output of the following code?

```
class EJavaGuru {
    public static void main(String args[]) {
        int num = 20;
        final int num2;
        num2 = 20;
        switch (num) {
            default: System.out.println("default");
            case num2: System.out.println(4);
            break;
        }
    }
}
```

- **a** `default`
- **b** `default`
  `4`
- **c** `4`
- **d** **Compilation error**

Answer: d

Explanation: The code will fail to compile. The `case` labels require compile-time constant values, and the variable `num2` doesn't qualify as such. Although the variable `num2` is defined as a `final` variable, it isn't assigned a value with its declaration. The code assigns a literal value `20` to this variable after its declaration, but it isn't considered to be a compile-time constant by the Java compiler.

Q5-7. What's the output of the following code?

```
class EJavaGuru {
    public static void main(String args[]) {
        int num = 120;

        switch (num) {
            default: System.out.println("default");
            case 0: System.out.println("case1");
            case 10*2-20: System.out.println("case2");
            break;
        }
    }
}
```

- **a** `default`
  `case1`
  `case2`
- **b** `case1`
  `case2`
- **c** `case2`
- **d** **Compilation error**
- **e** Runtime exception

Answer: d

Explanation: The expressions used for both case labels—that is, `0` and `10*2-20`—evaluate to the constant value `0`. Because you can't define duplicate case labels for the `switch` statement, the code will fail to compile with an error message that states that the code defines a duplicate case label.

Q5-8. What's the output of the following code?

```
class EJavaGuru3 {
    public static void main(String args[]) {
        byte foo = 120;
        switch (foo) {
            default: System.out.println("ejavaguru"); break;
            case 2: System.out.println("e"); break;
            case 120: System.out.println("ejava");
            case 121: System.out.println("enum");
            case 127: System.out.println("guru"); break;
        }
    }
}
```

a
```
ejava
enum
guru
```

b
```
ejava
```

c
```
ejavaguru
e
```

d
```
ejava
enum
guru
ejavaguru
```

Answer: a

Explanation: For a `switch` case construct, control enters the `case` labels when a matching `case` is found. The control then falls through the remaining `case` labels until it's terminated by a `break` statement. The control exits the `switch` construct when it encounters a `break` statement or it reaches the end of the `switch` construct.

In this example, a matching label is found for `case` label `120`. The control executes the statement for this case label and prints `ejava` to the console. Because a `break` statement doesn't terminate the case label, the control falls through to `case` label `121`. The control executes the statement for this `case` label and prints `enum` to the console. Because a `break` statement also doesn't terminate this case label, the control falls through to `case` label `127`. The control executes the statement for this `case` label and prints `guru` to the console. This case label is terminated by a `break` statement, so the control exits the `switch` construct.

Q5-9. What's the output of the following code?

```
class EJavaGuru4 {
    public static void main(String args[]) {
        boolean myVal = false;
        if (myVal=true)
        for (int i = 0; i < 2; i++) System.out.println(i);
        else System.out.println("else");
    }
}
```

a `else`

b
```
0
1
2
```

c
```
0
1
```

d Compilation error

Answer: c

Explanation: First of all, the expression used in the `if` construct isn't comparing the value of the variable `myVal` with the literal value `true`—it's assigning the literal value `true` to it. The assignment operator (`=`) assigns the literal value. The comparison operator (`==`) is used to compare values. Because the resulting value is a `boolean` value, the compiler doesn't complain about the assignment in the `if` construct.

The code is deliberately poorly indented because you may encounter similarly poor indentation in the OCA Java SE 7 Programmer I exam. The `for` loop is part of the `if` construct, which prints `0` and `1`. The `else` part doesn't execute because the `if` condition evaluates to `true`. The code has no compilation errors.

Q5-10. What's the output of the following code?

```
class EJavaGuru5 {
    public static void main(String args[]) {
        int i = 0;
        for (; i < 2; i=i+5) {
            if (i < 5) continue;
            System.out.println(i);
        }
        System.out.println(i);
    }
}
```

a Compilation error

b
```
0
5
```

c
```
0
5
10
```

d `10`

e
```
0
1
5
```

f `5`

Answer: f

Explanation: First of all, the following line of code has no compilation errors:

```
for (; i < 2; i=i+5) {
```

Using the initialization block is optional in a `for` loop. In this case, using a semicolon (`;`) terminates it.

For the first `for` iteration, the variable `i` has a value of `0`. Because this value is less than `2`, the following `if` construct evaluates to `true` and the `continue` statement executes:

```
if (i < 5) continue;
```

Because the `continue` statement ignores all of the remaining statements in a `for` loop iteration, the control doesn't print the value of the variable `i`, which leads the control to move on to the next `for` iteration. In the next `for` iteration, the value of the variable `i` is `5`. The `for` loop condition evaluates to `false` and the control moves out of the `for` loop. After the `for` loop, the code prints out the value of the variable `i`, which increments once using the code `i=i+5`.