# Control Flow Statements

The statements inside your source files are generally executed from top to bottom, in the order that they appear. *Control flow statements*, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to *conditionally* execute particular blocks of code. This section describes the decision-making statements (if-then, if-then-else, switch), the looping statements (for, while, do-while), and the branching statements (break, continue, return) supported by the Java programming language.\

# The if-then and if-then-else Statements

## The if-then Statement

The if-then statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code *only if* a particular test evaluates to true. For example, the Bicycle class could allow the brakes to decrease the bicycle's speed *only if* the bicycle is already in motion. One possible implementation of the applyBrakes method could be as follows:

void applyBrakes() {

// the "if" clause: bicycle must be moving

if (isMoving){

// the "then" clause: decrease current speed

currentSpeed--;

}

}

If this test evaluates to false (meaning that the bicycle is not in motion), control jumps to the end of the if-then statement.

In addition, the opening and closing braces are optional, provided that the "then" clause contains only one statement:

void applyBrakes() {

// same as above, but without braces

if (isMoving)

currentSpeed--;

}

Deciding when to omit the braces is a matter of personal taste. Omitting them can make the code more brittle. If a second statement is later added to the "then" clause, a common mistake would be forgetting to add the newly required braces. The compiler cannot catch this sort of error; you'll just get the wrong results.

## The if-then-else Statement

The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false. You could use an if-then-else statement in the applyBrakes method to take some action if the brakes are applied when the bicycle is not in motion. In this case, the action is to simply print an error message stating that the bicycle has already stopped.

void applyBrakes() {

if (isMoving) {

currentSpeed--;

} else {

System.err.println("The bicycle has already stopped!");

}

}

The following program, [IfElseDemo](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/IfElseDemo.java" \t "https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_blank), assigns a grade based on the value of a test score: an A for a score of 90% or above, a B for a score of 80% or above, and so on.

class IfElseDemo {

public static void main(String[] args) {

int testscore = 76;

char grade;

if (testscore >= 90) {

grade = 'A';

} else if (testscore >= 80) {

grade = 'B';

} else if (testscore >= 70) {

grade = 'C';

} else if (testscore >= 60) {

grade = 'D';

} else {

grade = 'F';

}

System.out.println("Grade = " + grade);

}

}

The output from the program is:

Grade = C

You may have noticed that the value of testscore can satisfy more than one expression in the compound statement: 76 >= 70 and 76 >= 60. However, once a condition is satisfied, the appropriate statements are executed (grade = 'C';) and the remaining conditions are not evaluated.

# The switch Statement

Unlike if-then and if-then-else statements, the switch statement can have a number of possible execution paths. A switch works with the byte, short, char, and int primitive data types. It also works with *enumerated types* (discussed in [Enum Types](https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html" \t "https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_top)), the [String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html" \t "https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_blank) class, and a few special classes that wrap certain primitive types: [Character](https://docs.oracle.com/javase/8/docs/api/java/lang/Character.html" \t "https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_blank), [Byte](https://docs.oracle.com/javase/8/docs/api/java/lang/Byte.html" \t "https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_blank), [Short](https://docs.oracle.com/javase/8/docs/api/java/lang/Short.html" \t "https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_blank), and [Integer](https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html" \t "https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_blank) (discussed in [Numbers and Strings](https://docs.oracle.com/javase/tutorial/java/data/index.html" \t "https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_top)).

The following code example, [SwitchDemo](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/SwitchDemo.java" \t "https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_blank), declares an int named month whose value represents a month. The code displays the name of the month, based on the value of month, using the switch statement.

public class SwitchDemo {

public static void main(String[] args) {

int month = 8;

String monthString;

switch (month) {

case 1: monthString = "January";

break;

case 2: monthString = "February";

break;

case 3: monthString = "March";

break;

case 4: monthString = "April";

break;

case 5: monthString = "May";

break;

case 6: monthString = "June";

break;

case 7: monthString = "July";

break;

case 8: monthString = "August";

break;

case 9: monthString = "September";

break;

case 10: monthString = "October";

break;

case 11: monthString = "November";

break;

case 12: monthString = "December";

break;

default: monthString = "Invalid month";

break;

}

System.out.println(monthString);

}

}

In this case, August is printed to standard output.

The body of a switch statement is known as a *switch block*. A statement in the switch block can be labeled with one or more case or default labels. The switch statement evaluates its expression, then executes all statements that follow the matching case label.

You could also display the name of the month with if-then-else statements:

int month = 8;

if (month == 1) {

System.out.println("January");

} else if (month == 2) {

System.out.println("February");

}

... // and so on

Deciding whether to use if-then-else statements or a switch statement is based on readability and the expression that the statement is testing. An if-then-else statement can test expressions based on ranges of values or conditions, whereas a switch statement tests expressions based only on a single integer, enumerated value, or String object.

Another point of interest is the break statement. Each break statement terminates the enclosing switch statement. Control flow continues with the first statement following the switch block. The break statements are necessary because without them, statements in switch blocks fall through: All statements after the matching case label are executed in sequence, regardless of the expression of subsequent case labels, until a break statement is encountered. The program [SwitchDemoFallThrough](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/SwitchDemoFallThrough.java" \t "https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_blank) shows statements in a switch block that fall through. The program displays the month corresponding to the integer month and the months that follow in the year:

public class SwitchDemoFallThrough {

public static void main(String[] args) {

java.util.ArrayList<String> futureMonths =

new java.util.ArrayList<String>();

int month = 8;

switch (month) {

case 1: futureMonths.add("January");

case 2: futureMonths.add("February");

case 3: futureMonths.add("March");

case 4: futureMonths.add("April");

case 5: futureMonths.add("May");

case 6: futureMonths.add("June");

case 7: futureMonths.add("July");

case 8: futureMonths.add("August");

case 9: futureMonths.add("September");

case 10: futureMonths.add("October");

case 11: futureMonths.add("November");

case 12: futureMonths.add("December");

break;

default: break;

}

if (futureMonths.isEmpty()) {

System.out.println("Invalid month number");

} else {

for (String monthName : futureMonths) {

System.out.println(monthName);

}

}

}

}

This is the output from the code:

August

September

October

November

December

Technically, the final break is not required because flow falls out of the switch statement. Using a break is recommended so that modifying the code is easier and less error prone. The default section handles all values that are not explicitly handled by one of the case sections.

The following code example, [SwitchDemo2](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/SwitchDemo2.java" \t "https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_blank), shows how a statement can have multiple case labels. The code example calculates the number of days in a particular month:

class SwitchDemo2 {

public static void main(String[] args) {

int month = 2;

int year = 2000;

int numDays = 0;

switch (month) {

case 1: case 3: case 5:

case 7: case 8: case 10:

case 12:

numDays = 31;

break;

case 4: case 6:

case 9: case 11:

numDays = 30;

break;

case 2:

if (((year % 4 == 0) &&

!(year % 100 == 0))

|| (year % 400 == 0))

numDays = 29;

else

numDays = 28;

break;

default:

System.out.println("Invalid month.");

break;

}

System.out.println("Number of Days = "

+ numDays);

}

}

This is the output from the code:

Number of Days = 29

## Using Strings in switch Statements

In Java SE 7 and later, you can use a String object in the switch statement's expression. The following code example, [StringSwitchDemo](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/StringSwitchDemo.java" \t "https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_blank), displays the number of the month based on the value of the String named month:

public class StringSwitchDemo {

public static int getMonthNumber(String month) {

int monthNumber = 0;

if (month == null) {

return monthNumber;

}

switch (month.toLowerCase()) {

case "january":

monthNumber = 1;

break;

case "february":

monthNumber = 2;

break;

case "march":

monthNumber = 3;

break;

case "april":

monthNumber = 4;

break;

case "may":

monthNumber = 5;

break;

case "june":

monthNumber = 6;

break;

case "july":

monthNumber = 7;

break;

case "august":

monthNumber = 8;

break;

case "september":

monthNumber = 9;

break;

case "october":

monthNumber = 10;

break;

case "november":

monthNumber = 11;

break;

case "december":

monthNumber = 12;

break;

default:

monthNumber = 0;

break;

}

return monthNumber;

}

public static void main(String[] args) {

String month = "August";

int returnedMonthNumber =

StringSwitchDemo.getMonthNumber(month);

if (returnedMonthNumber == 0) {

System.out.println("Invalid month");

} else {

System.out.println(returnedMonthNumber);

}

}

}

The output from this code is 8.

The String in the switch expression is compared with the expressions associated with each case label as if the [String.equals](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html" \l "equals-java.lang.Object-" \t "https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_blank) method were being used. In order for the StringSwitchDemo example to accept any month regardless of case, month is converted to lowercase (with the [toLowerCase](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html" \l "toLowerCase--" \t "https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_blank) method), and all the strings associated with the case labels are in lowercase.

**Note**: This example checks if the expression in the switch statement is null. Ensure that the expression in any switch statement is not null to prevent a NullPointerException from being thrown.

# The while and do-while Statements

The while statement continually executes a block of statements while a particular condition is true. Its syntax can be expressed as:

while (expression) {

statement(s)

}

The while statement evaluates *expression*, which must return a boolean value. If the expression evaluates to true, the while statement executes the statement(s) in the while block. The while statement continues testing the expression and executing its block until the expression evaluates to false. Using the while statement to print the values from 1 through 10 can be accomplished as in the following [WhileDemo](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/WhileDemo.java" \t "https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_blank) program:

class WhileDemo {

public static void main(String[] args){

int count = 1;

while (count < 11) {

System.out.println("Count is: " + count);

count++;

}

}

}

You can implement an infinite loop using the while statement as follows:

while (true){

// your code goes here

}

The Java programming language also provides a do-while statement, which can be expressed as follows:

do {

statement(s)

} while (expression);

The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once, as shown in the following [DoWhileDemo](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/DoWhileDemo.java" \t "https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_blank) program:

class DoWhileDemo {

public static void main(String[] args){

int count = 1;

do {

System.out.println("Count is: " + count);

count++;

} while (count < 11);

}

}

# Branching Statements

## The break Statement

The break statement has two forms: labeled and unlabeled. You saw the unlabeled form in the previous discussion of the switch statement. You can also use an unlabeled break to terminate a for, while, or do-while loop, as shown in the following [BreakDemo](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/BreakDemo.java" \t "https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_blank) program:

class BreakDemo {

public static void main(String[] args) {

int[] arrayOfInts =

{ 32, 87, 3, 589,

12, 1076, 2000,

8, 622, 127 };

int searchfor = 12;

int i;

boolean foundIt = false;

for (i = 0; i < arrayOfInts.length; i++) {

if (arrayOfInts[i] == searchfor) {

foundIt = true;

**break;**

}

}

if (foundIt) {

System.out.println("Found " + searchfor + " at index " + i);

} else {

System.out.println(searchfor + " not in the array");

}

}

}

This program searches for the number 12 in an array. The break statement, shown in boldface, terminates the for loop when that value is found. Control flow then transfers to the statement after the for loop. This program's output is:

Found 12 at index 4

An unlabeled break statement terminates the innermost switch, for, while, or do-while statement, but a labeled break terminates an outer statement. The following program, [BreakWithLabelDemo](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/BreakWithLabelDemo.java" \t "https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_blank), is similar to the previous program, but uses nested for loops to search for a value in a two-dimensional array. When the value is found, a labeled break terminates the outer for loop (labeled "search"):

class BreakWithLabelDemo {

public static void main(String[] args) {

int[][] arrayOfInts = {

{ 32, 87, 3, 589 },

{ 12, 1076, 2000, 8 },

{ 622, 127, 77, 955 }

};

int searchfor = 12;

int i;

int j = 0;

boolean foundIt = false;

search:

for (i = 0; i < arrayOfInts.length; i++) {

for (j = 0; j < arrayOfInts[i].length;

j++) {

if (arrayOfInts[i][j] == searchfor) {

foundIt = true;

break search;

}

}

}

if (foundIt) {

System.out.println("Found " + searchfor + " at " + i + ", " + j);

} else {

System.out.println(searchfor + " not in the array");

}

}

}

This is the output of the program.

Found 12 at 1, 0

The break statement terminates the labeled statement; it does not transfer the flow of control to the label. Control flow is transferred to the statement immediately following the labeled (terminated) statement.

## The continue Statement

The continue statement skips the current iteration of a for, while , or do-while loop. The unlabeled form skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop. The following program, [ContinueDemo](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/ContinueDemo.java" \t "https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_blank) , steps through a String, counting the occurrences of the letter "p". If the current character is not a p, the continue statement skips the rest of the loop and proceeds to the next character. If it *is* a "p", the program increments the letter count.

class ContinueDemo {

public static void main(String[] args) {

String searchMe = "peter piper picked a " + "peck of pickled peppers";

int max = searchMe.length();

int numPs = 0;

for (int i = 0; i < max; i++) {

// interested only in p's

if (searchMe.charAt(i) != 'p')

continue;

// process p's

numPs++;

}

System.out.println("Found " + numPs + " p's in the string.");

}

}

Here is the output of this program:

Found 9 p's in the string.

To see this effect more clearly, try removing the continue statement and recompiling. When you run the program again, the count will be wrong, saying that it found 35 p's instead of 9.

A labeled continue statement skips the current iteration of an outer loop marked with the given label. The following example program, ContinueWithLabelDemo, uses nested loops to search for a substring within another string. Two nested loops are required: one to iterate over the substring and one to iterate over the string being searched. The following program, [ContinueWithLabelDemo](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/ContinueWithLabelDemo.java" \t "https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_blank), uses the labeled form of continue to skip an iteration in the outer loop.

class ContinueWithLabelDemo {

public static void main(String[] args) {

String searchMe = "Look for a substring in me";

String substring = "sub";

boolean foundIt = false;

int max = searchMe.length() -

substring.length();

test:

for (int i = 0; i <= max; i++) {

int n = substring.length();

int j = i;

int k = 0;

while (n-- != 0) {

if (searchMe.charAt(j++) != substring.charAt(k++)) {

continue test;

}

}

foundIt = true;

break test;

}

System.out.println(foundIt ? "Found it" : "Didn't find it");

}

}

Here is the output from this program.

Found it

## The return Statement

The last of the branching statements is the return statement. The return statement exits from the current method, and control flow returns to where the method was invoked. The return statement has two forms: one that returns a value, and one that doesn't. To return a value, simply put the value (or an expression that calculates the value) after the return keyword.

return ++count;

The data type of the returned value must match the type of the method's declared return value. When a method is declared void, use the form of return that doesn't return a value.

return;

The [Classes and Objects](https://docs.oracle.com/javase/tutorial/java/javaOO/methods.html" \t "https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_top) lesson will cover everything you need to know about writing methods.