# Chapter 5

# Conditionals and logic

The programs we've seen in previous chapters do pretty much the same thing every time, regardless of the input. For more complex computations, programs usually react to the inputs, check for certain conditions, and generate appropriate results. This chapter presents the features you need for programs to make decisions: a new data type called `boolean`, operators for expressing logic, and `if` statements.

## 5.1 Relational operators

**Relational operators** are used to check conditions like whether two values are equal, or whether one is greater than the other. The following expressions show how they are used:

```
x == y          // x is equal to y
x != y          // x is not equal to y
x > y           // x is greater than y
x < y           // x is less than y
x >= y          // x is greater than or equal to y
x <= y          // x is less than or equal to y
```

The result of a relational operator is one of two special values, `true` or `false`. These values belong to the data type `boolean`; in fact, they are the only boolean values.

You are probably familiar with these operations, but notice that the Java operators are different from the mathematical symbols like $=$, $\neq$, and $\leq$. A common error is to use a single `=` instead of a double `==`. Remember that `=` is the assignment operator, and `==` is a comparison operator. Also, there is no such thing as the `=<` or `=>` operators.

The two sides of a relational operator have to be compatible. For example, the expression `5 < "6"` is invalid because `5` is an `int` and `"6"` is a `String`. When comparing values of different numeric types, Java applies the same conversion rules we saw previously with the assignment operator. For example, when evaluating the expression `5 < 6.0`, Java automatically converts the `5` to `5.0`.

Most relational operators don't work with strings. But confusingly, `==` and `!=` do work with strings – they just don't do what you expect. We'll explain what they do later; in the meantime, don't use them with strings. Instead, you should use the `equals` method:

```
String fruit1 = "Apple";
String fruit2 = "Orange";
System.out.println(fruit1.equals(fruit2));
```

The result of `fruit1.equals(fruit2)` is the boolean value `false`.

## 5.2 Logical operators

Java has three **logical operators**: `&&`, `||`, and `!`, which respectively stand for *and*, *or*, and *not*. The results of these operators are similar to their meanings in English.

For example, `x > 0 && x < 10` is true when `x` is both greater than zero *and* less than 10. The expression `evenFlag || n % 3 == 0` is true if either condition is true, that is, if `evenFlag` is true *or* the number `n` is divisible by 3. Finally, the `!` operator inverts a boolean expression. So `!evenFlag` is true if `evenFlag` is *not* true.

Logical operators evaluate the second expression only when necessary. For example, `true || anything` is always true, so Java does not need to evaluate the expression `anything`. Likewise, `false && anything` is always false.

Ignoring the second operand, when possible, is called **short circuit** evaluation, by analogy with an electrical circuit. Short circuit evaluation can save time, especially if `anything` takes a long time to evaluate. It can also avoid unnecessary errors, if `anything` might fail.

If you ever have to negate an expression that contains logical operators, and you probably will, **De Morgan's laws** can help:

- `!(A && B)` is the same as `!A || !B`
- `!(A || B)` is the same as `!A && !B`

Negating a logical expression is the same as negating each term and changing the operator. The `!` operator takes precedence over `&&` and `||`, so you don't have to put parentheses around the individual terms `!A` and `!B`.

De Morgan's laws also apply to the relational operators. In this case, negating each term means using the "opposite" relational operator.

- `!(x < 5 && y == 3)` is the same as `x >= 5 || y != 3`
- `!(x >= 1 || y != 7)` is the same as `x < 1 && y == 7`

It may help to read these examples out loud in English. For instance, "If I don't want x to be less than 5, and I don't want y to be 3, then I need x to be greater than or equal to 5, or I need y to be anything but 3."

## 5.3 Conditional statements

To write useful programs, we almost always need to check conditions and react accordingly. **Conditional statements** give us this ability. The simplest conditional statement in Java is the `if` statement:

```
if (x > 0) {
    System.out.println("x is positive");
}
```

The expression in parentheses is called the condition. If it is true, the statements in braces get executed. If the condition is false, execution skips over that block of code. The condition in parentheses can be any boolean expression.

A second form of conditional statement has two possibilities, indicated by `if` and `else`. The possibilities are called **branches**, and the condition determines which one gets executed:

```
if (x % 2 == 0) {
    System.out.println("x is even");
} else {
    System.out.println("x is odd");
}
```

If the remainder when `x` is divided by 2 is zero, we know that `x` is even, and this fragment displays a message to that effect. If the condition is false, the second print statement is executed instead. Since the condition must be true or false, exactly one of the branches will run.

The braces are optional for branches that have only one statement. So we could have written the previous example this way:

```
if (x % 2 == 0)
    System.out.println("x is even");
else
    System.out.println("x is odd");
```

However, it's better to use braces – even when they are optional – to avoid making the mistake of adding statements to an `if` or `else` block and forgetting to add the braces.

```
if (x > 0)
    System.out.println("x is positive");
    System.out.println("x is not zero");
```

This code is misleading because it's not indented correctly. Since there are no braces, only the first `println` is part of the `if` statement. Here is what the compiler actually sees:

```
if (x > 0) {
    System.out.println("x is positive");
}
    System.out.println("x is not zero");
```

As a result, the second `println` runs no matter what. Even experienced programmers make this mistake; search the web for Apple's "goto fail" bug.

## 5.4 Chaining and nesting

Sometimes you want to check related conditions and choose one of several actions. One way to do this is by **chaining** a series of `if` and `else` statements:

```
if (x > 0) {
    System.out.println("x is positive");
} else if (x < 0) {
    System.out.println("x is negative");
} else {
    System.out.println("x is zero");
}
```

These chains can be as long as you want, although they can be difficult to read if they get out of hand. One way to make them easier to read is to use standard indentation, as demonstrated in these examples. If you keep all the statements and braces lined up, you are less likely to make syntax errors.

In addition to chaining, you can also make complex decisions by **nesting** one conditional statement inside another. We could have written the previous example as:

```
if (x == 0) {
    System.out.println("x is zero");
} else {
    if (x > 0) {
        System.out.println("x is positive");
    } else {
        System.out.println("x is negative");
    }
}
```

The outer conditional has two branches. The first branch contains a `print` statement, and the second branch contains another conditional statement, which has two branches of its own. These two branches are also `print` statements, but they could have been conditional statements as well.

These kinds of nested structures are common, but they get difficult to read very quickly. Good indentation is essential to make the structure (or intended structure) apparent to the reader.

## 5.5 Flag variables

To store a `true` or `false` value, you need a `boolean` variable. You can create one like this:

```
boolean flag;
flag = true;
boolean testResult = false;
```

The first line is a variable declaration, the second is an assignment, and the third is both. Since relational operators evaluate to a `boolean` value, you can store the result of a comparison in a variable:

```
boolean evenFlag = (n % 2 == 0);     // true if n is even
boolean positiveFlag = (x > 0);      // true if x is positive
```

The parentheses are unnecessary, but they make the code easier to read. A variable defined in this way is called a **flag**, because it signals or "flags" the presence or absence of a condition.

You can use flag variables as part of a conditional statement later:

```
if (evenFlag) {
    System.out.println("n was even when I checked it");
}
```

Notice that you don't have to write `if (evenFlag == true)`. Since `evenFlag` is a `boolean`, it's already a condition. Likewise, to check if a flag is `false`:

```
if (!evenFlag) {
    System.out.println("n was odd when I checked it");
}
```

## 5.6 The return statement

The `return` statement allows you to terminate a method before you reach the end of it. One reason to use `return` is if you detect an error condition:

```
public static void printLogarithm(double x) {
    if (x <= 0.0) {
        System.err.println("Error: x must be positive.");
        return;
    }
    double result = Math.log(x);
    System.out.println("The log of x is " + result);
}
```

This example defines a method named `printLogarithm` that takes a `double` value (named `x`) as a parameter. It checks whether `x` is less than or equal to zero, in which case it displays an error message and then uses `return` to exit the method. The flow of execution immediately returns to where the method was invoked, and the remaining lines of the method are not executed.

This example uses `System.err`, which is an `OutputStream` normally used for error messages and warnings. Some development environments display output to `System.err` with a different color or in a separate window.

## 5.7 Validating input

Here is a method that uses `printLogarithm` from the previous section:

```
public static void scanDouble() {
    Scanner in = new Scanner(System.in);
    System.out.print("Enter a number: ");
    double x = in.nextDouble();
    printLogarithm(x);
}
```

This example uses `nextDouble`, so the `Scanner` tries to read a `double`. If the user enters a floating-point number, the `Scanner` converts it to a `double`. But if the user types anything else, the `Scanner` throws an `InputMismatchException`.

We can prevent this error by checking the input before parsing it:

```
public static void scanDouble() {
    Scanner in = new Scanner(System.in);
    System.out.print("Enter a number: ");
    if (!in.hasNextDouble()) {
        String word = in.next();
        System.err.println(word + " is not a number");
        return;
    }
    double x = in.nextDouble();
    printLogarithm(x);
}
```

The `Scanner` class provides `hasNextDouble`, which checks whether the next token in the input stream can be interpreted as a `double`. If so, we can call `nextDouble` with no chance of throwing an exception. If not, we display an error message and return. Returning from `main` terminates the program.

## 5.8 Recursive methods

Now that we have conditional statements, we can explore one of the most magical things a program can do: **recursion**. Consider the following example:

```
public static void countdown(int n) {
    if (n == 0) {
        System.out.println("Blastoff!");
    } else {
        System.out.println(n);
        countdown(n - 1);
    }
}
```

The name of the method is `countdown`; it takes a single integer as a parameter. If the parameter is zero, it displays the word "Blastoff". Otherwise, it displays the number and then invokes *itself*, passing `n - 1` as the argument. A method that invokes itself is called **recursive**.

What happens if we invoke `countdown(3)` from `main`?

> The execution of `countdown` begins with `n == 3`, and since `n` is not zero, it displays the value 3, and then invokes itself...
>
> > The execution of `countdown` begins with `n == 2`, and since `n` is not zero, it displays the value 2, and then invokes itself...
> >
> > > The execution of `countdown` begins with `n == 1`, and since `n` is not zero, it displays the value 1, and then invokes itself...
> > >
> > > > The execution of `countdown` begins with `n == 0`, and since `n` is zero, it displays the word "Blastoff!" and then returns.
> > >
> > > The `countdown` that got `n == 1` returns.
> >
> > The `countdown` that got `n == 2` returns.
>
> The `countdown` that got `n == 3` returns.

And then you're back in `main`. So the total output looks like:

```
3
2
1
Blastoff!
```

As a second example, we'll rewrite the methods `newLine` and `threeLine` from Section 4.3.

```
public static void newLine() {
    System.out.println();
}

public static void threeLine() {
    newLine();
    newLine();
    newLine();
}
```

Although these methods work, they would not help if we wanted to display two newlines, or maybe 100. A better alternative would be:

```
public static void nLines(int n) {
    if (n > 0) {
        System.out.println();
        nLines(n - 1);
    }
}
```

This method takes an integer, `n`, as a parameter and displays `n` newlines. The structure is similar to `countdown`. As long as $n$ is greater than zero, it displays a newline and then invokes itself to display $(n - 1)$ additional newlines. The total number of newlines is $1 + (n - 1)$, which is just what we wanted: $n$.

## 5.9 Recursive stack diagrams

In the previous chapter, we used a stack diagram to represent the state of a program during a method invocation. The same kind of diagram can make it easier to interpret a recursive method.

Remember that every time a method gets called, Java creates a new frame that contains the current method's parameters and variables. Figure 5.1 is a stack diagram for `countdown`, called with `n == 3`.

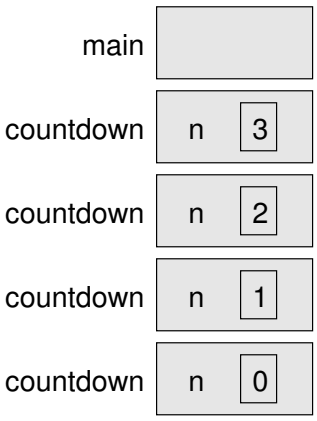


Figure 5.1: Stack diagram for the `countdown` program.

By convention, the stack for `main` is at the top and the stack grows down. The frame for `main` is empty because `main` does not have any variables. (It has the parameter `args`, but since we're not using it, we left it out of the diagram.)

There are four frames for `countdown`, each with a different value for the parameter `n`. The last frame, with `n == 0`, is called the **base case**. It does not make a recursive call, so there are no more frames below it.

If there is no base case in a recursive method, or if the base case is never reached, the stack would grow forever, at least in theory. In practice, the size of the stack is limited; if you exceed the limit, you get a `StackOverflowError`.

For example, here is a recursive method without a base case:

```
public static void forever(String s) {
    System.out.println(s);
    forever(s);
}
```

This method displays the string until the stack overflows, at which point it throws an exception.

## 5.10 Binary numbers

The `countdown` example has three parts: (1) it checks the base case, (2) displays something, and (3) makes a recursive call. What do you think happens if you reverse steps 2 and 3, making the recursive call *before* displaying?

```
public static void countup(int n) {
    if (n == 0) {
        System.out.println("Blastoff!");
    } else {
        countup(n - 1);
        System.out.println(n);
    }
}
```

The stack diagram is the same as before, and the method is still called $n$ times. But now the `System.out.println` happens just before each recursive call returns. As a result, it counts up instead of down:

```
Blastoff!
1
2
3
```

This behavior comes in handy when it is easier to compute results in reverse order. For example, to convert a decimal integer into its **binary** representation, you repeatedly divide the number by two:

```
23 / 2 is 11 remainder 1
11 / 2 is  5 remainder 1
 5 / 2 is  2 remainder 1
 2 / 2 is  1 remainder 0
 1 / 2 is  0 remainder 1
```

Reading these remainders from bottom to top, 23 in binary is 10111. For more background about binary numbers, see http://www.mathsisfun.com/binary-number-system.html.

Here is a recursive method that displays the binary representation of any positive integer:

```
public static void displayBinary(int value) {
    if (value > 0) {
        displayBinary(value / 2);
        System.out.print(value % 2);
    }
}
```

If `value` is zero, `displayBinary` does nothing (that's the base case). If the argument is positive, the method divides it by two and calls `displayBinary` recursively. When the recursive call returns, the method displays one digit of the result and returns (again).

The leftmost digit is at the bottom of the stack, so it gets displayed first. The rightmost digit, at the top of the stack, gets displayed last. After invoking `displayBinary`, we use `println` to complete the output.

```
displayBinary(23);
System.out.println();
// output is 10111
```

Learning to think recursively is an important aspect of learning to think like a computer scientist. Many algorithms can be written concisely with recursive methods that perform computations on the way down, on the way up, or both.

## 5.11 Vocabulary

**boolean:** A data type with only two values, `true` and `false`.

**relational operator:** An operator that compares two values and produces a `boolean` indicating the relationship between them.

**logical operator:** An operator that combines boolean values and produces a boolean value.

**short circuit:** A way of evaluating logical operators that only evaluates the second operand if necessary.

**De Morgan's laws:** Mathematical rules that show how to negate a logical expression.

**conditional statement:** A statement that uses a condition to determine which statements to execute.

**branch:** One of the alternative sets of statements inside a conditional statement.

**chaining:** A way of joining several conditional statements in sequence.

**nesting:** Putting a conditional statement inside one or both branches of another conditional statement.

**flag:** A variable (usually `boolean`) that represents a condition or status.

**recursion:** The process of invoking (and restarting) the same method that is currently executing.

**recursive:** A method that invokes itself, usually with different arguments.

**base case:** A condition that causes a recursive method *not* to make another recursive call.

**binary:** A system that uses only zeros and ones to represent numbers. Also known as "base 2".

## 5.12 Exercises

The code for this chapter is in the `ch05` directory of `ThinkJavaCode`. See page xv for instructions on how to download the repository. Before you start the exercises, we recommend that you compile and run the examples.

If you have not already read Appendix A.6, now might be a good time. It describes the DrJava debugger, which is a useful tool for tracing the flow of execution.

**Exercise 5.1** Logical operators can simplify nested conditional statements. For example, can you rewrite this code using a single `if` statement?

```
if (x > 0) {
    if (x < 10) {
        System.out.println("positive single digit number.");
    }
}
```

**Exercise 5.2** For the following program:

1. Draw a stack diagram that shows the state of the program the *second* time `ping` is invoked.
2. What is the complete output?

```
public static void zoop(String fred, int bob) {
    System.out.println(fred);
    if (bob == 5) {
        ping("not ");
    } else {
        System.out.println("!");
    }
}

public static void main(String[] args) {
    int bizz = 5;
    int buzz = 2;
    zoop("just for", bizz);
    clink(2 * buzz);
}
```

```
public static void clink(int fork) {
    System.out.print("It's ");
    zoop("breakfast ", fork) ;
}

public static void ping(String strangStrung) {
    System.out.println("any " + strangStrung + "more ");
}
```

**Exercise 5.3** Draw a stack diagram that shows the state of the program in Section 5.8 after `main` invokes `nLines` with the parameter `n == 4`, just before the last invocation of `nLines` returns.

**Exercise 5.4** Fermat's Last Theorem says that there are no integers $a$, $b$, and $c$ such that $a^n + b^n = c^n$, except when $n \leq 2$.

Write a method named `checkFermat` that takes four integers as parameters – `a`, `b`, `c` and `n` – and checks to see if Fermat's theorem holds. If $n$ is greater than 2 and $a^n + b^n = c^n$, the program should display "Holy smokes, Fermat was wrong!" Otherwise the program should display "No, that doesn't work."

*Hint:* You may want to use `Math.pow`.

**Exercise 5.5** The purpose of this exercise is to take a problem and break it into smaller problems, and to solve the smaller problems by writing simple methods. Consider the first verse of the song "99 Bottles of Beer":

> 99 bottles of beer on the wall,
> 99 bottles of beer,
> ya' take one down, ya' pass it around,
> 98 bottles of beer on the wall.

Subsequent verses are identical except that the number of bottles gets smaller by one in each verse, until the last verse:

> No bottles of beer on the wall,
> no bottles of beer,
> ya' can't take one down, ya' can't pass it around,
> 'cause there are no more bottles of beer on the wall!

And then the song (finally) ends.

Write a program that displays the entire lyrics of "99 Bottles of Beer". Your program should include a recursive method that does the hard part, but you might want to write additional methods to separate other parts of the program. As you develop your code, test it with a small number of verses, like 3.

**Exercise 5.6** This exercise reviews the flow of execution through a program with multiple methods. Read the following code and answer the questions.

```
public class Buzz {

    public static void baffle(String blimp) {
        System.out.println(blimp);
        zippo("ping", -5);
    }

    public static void zippo(String quince, int flag) {
        if (flag < 0) {
            System.out.println(quince + " zoop");
        } else {
            System.out.println("ik");
            baffle(quince);
            System.out.println("boo-wa-ha-ha");
        }
    }

    public static void main(String[] args) {
        zippo("rattle", 13);
    }

}
```

1. Write the number 1 next to the first line of code in this program that will execute.

2. Write the number 2 next to the second line of code, and so on until the end of the program. If a line is executed more than once, it might end up with more than one number next to it.

3. What is the value of the parameter `blimp` when `baffle` gets invoked?

4. What is the output of this program?

**Exercise 5.7** Now that we have conditional statements, we can get back to the "Guess My Number" game from Exercise 3.4.

You should already have a program that chooses a random number, prompts the user to guess it, and displays the difference between the guess and the chosen number.

Adding a small amount of code at a time, and testing as you go, modify the program so it tells the user whether the guess is too high or too low, and then prompts the user for another guess.

The program should continue until the user gets it right. *Hint:* Use two methods, and make one of them recursive.