Chapter 5

Boolean Expressions and Conditional Execution

Arithmetic expressions evaluate to numeric values; a Boolean expression, sometimes called a *predicate*, evaluates to `false` or `true`. Boolean expressions are essential for building conditional and iterative statements. The simplest Boolean expressions are `true` and `false`. More interesting Boolean expressions can be built by comparing program elements (variables and literals), checking for equality or inequality.

5.1 Relational Operators

Relational operators are used to compare two expressions. They evaluate to `boolean` values, `true` or `false`. Table 5.1 lists the relational operators available in Java.

Operator	Meaning
==	is equal to
>=	is greater than or equal to
<=	is less than or equal to
<	is less than
>	is greater than
!=	is not equal to

Table 5.1: Relational operators

Boolean expressions can be entered into the interpreter:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> int x = 10;
> 10 < 20
true
> x < 20
true
```

```
> 10 > 20
false
> x > 20
false
> x == 10
true
> x == 20
false
> x != 10
false
> x != 20
true
> x <= 10
true
> x >= 10
true
> x <= 11
true
> x <= 9
false
```

(Be careful not to confuse the > Interactions pane prompt on the left of each expression with the > greater than operator that may be used in an expression.)

An expression like `10 < 20` is legal but of little use, since the expression `true` is equivalent, simpler, and less likely to confuse human readers. Boolean expressions are extremely useful when their truth values depend on the state of the program (for example, the value of a variable).

The relational operators are binary operators, and all have a lower precedence than any of the arithmetic operators; therefore, the expression

```
x + 2 < y / 10
```

is evaluated as if parentheses were placed as so:

```
(x + 2) < (y / 10)
```

Simple Boolean expressions, each involving one relational operator, can be combined into more complex Boolean expressions using the logical operators `&&` (and), `||` (or), and `!` (not). A combination of two or more Boolean expressions is called a *compound Boolean expression*. Table 5.2 shows how these logical operators work. Both `&&` and `||` are binary operators; that is, they require two operands, both of which must be Boolean expressions. Logical *not* (`!`) is a unary operator; it requires a single Boolean operand immediately to its right. Note that `||` is an *inclusive or*; that is, if either one or both of its operands is true, then the `||` expression is true.

Suppose you want to determine if an integer variable `x` is in the range 1...10. In order for `x` to be within this range the following two conditions simultaneously must be true: `x >= 1` and `x <= 10`. The only numbers that meet both of these criteria are 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10.

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> x = 5;
```

e_1	e_2	e_1 && e_2 (and)	e_1 || e_2 (or)	! e_1 (not)
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Table 5.2: Logical operators. e_1 and e_2 and logical expressions.

```
> x >= 1 && x <= 10
true
> x = 15;
> x >= 1 && x <= 10
false
> x = 0;
> x >= 1 && x <= 10
false
```

! has higher precedence than any binary operator. && has higher precedence than ||. && and || have lower precedence than any other binary operator except assignment. This means the expression

```
x <= y && x <= z
```

is evaluated as if parentheses were added as shown here:

```
(x <= y) && (x <= z)
```

Some programmers prefer to use the parentheses as shown here even though they are not required. The parentheses improve the readability of complex expressions, and the compiled code is no less efficient.

Suppose we want to see if a variable, x, is less than two other variables, y and z. Informally we may think, “Is x less than both y and z?” and express this thought as:

Interactions
```
Welcome to DrJava.  Working directory is /Users/rick/java
> int x = 10, y = 20, z = 30;
> x < y && z
Error: And expression type
```

Logical **and** (&&) has lower precedence than less than (<), so the expression is evaluated as

```
(x < y) && z
```

Recall the operands of && must be boolean expressions, but here z is an integer. Regrouping the parentheses would not help:

```
x < (y && z)
```

Now neither operands of the `&&` are Boolean expressions! The correct way to express what we intend is:

Interactions
```
Welcome to DrJava.  Working directory is /Users/rick/java
> int x = 10, y = 20, z = 30;
> (x < y) && (x < z)
true
```

Thus the correct way to think about the comparison is to treat it as two independent clauses: "Is `x` less than `y`, and is `x` less than `z`?"

5.2 The `if` Statement

A program that behaves exactly the same way every time it is run is not very interesting. A chess program that makes the same moves every game it plays soon would be defeated easily. Useful software alters its behavior under varying environmental conditions, such as user input.

In Java, the fundamental statement for varying program behavior is the conditional statement. The simplest conditional statement uses `if` to optionally execute code. The simple `if` statement has the following general form:

```
if ( condition )
        body
```

where

- The reserved word `if` identifies an `if` statement.
- *condition* is a Boolean expression that determines whether or not the body will be executed. The condition must be enclosed within parentheses.
- *body* is either a single Java statement or a block of statements (called a *compound statement*) enclosed within curly braces. The curly braces are optional if the body consists of a single statement. Many developers always use curly braces as a matter of style, even when they are not required. If the body consists of only one statement and curly braces are not used, then the semicolon that terminates the statement in the body also terminates the `if` statement. If curly braces are used to delimit the body, a semicolon is not required after the close curly brace.

Figure 5.1 shows how program execution flows through a simple `if` statement.

The following interactive session illustrates the use of the simple `if` statement:

Interactions
```
Welcome to DrJava.  Working directory is /Users/rick/java
> int x = 20, y = 40;
> y
40
> if (x > 10)
     y = 50;
> y
```

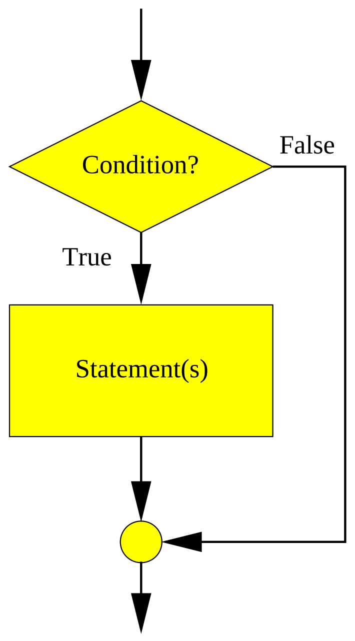

Figure 5.1: Execution flow in an `if` statement

```
50
> if (x > 100)
     y = 2;
> y
50
```

Notice that `y` is initially 40, as expected. The first `if` statement reassigns `y` to be 50, but the second `if` statement does not reassign `y` to 2. In the first case the Boolean expression `x > 10` is true, but in the second case `x > 100` is false. In both cases the reassignment of `y` is conditional on the value of `x`.

As shown above, a single `if` statement spans two lines. This is required neither in the DrJava interpreter nor by the Java language, but when we write Java programs it will improve the readability of the code. Also, we indented the body of the `if` on the second line of the statement. Again, neither the interpreter nor the Java language dictates this, but indentation is a standard convention to make the code more readable to humans. The compiler and interpreter are just as satisfied with

Interactions

```
> if (x < 100) y = 2;
> y
2
```

but to the human reader the indented body indicates that the code is optionally executed.

If more than one statement is to be included in the body, curly braces are required, as in

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> int x = 20, y = 40, z = 60;
> x
20
```

```
> y
40
> z
60
> if (x == 20) {
      x = 10;
      y = 5;
      z = 2;
  }
> x
10
> y
5
> z
2
```

If the curly braces are omitted, the first statement is considered the body and the other statements are not part of the `if` statement. Remember, the indentation is for the convenience of the human reader not the compiler or interpreter. It is a good habit to enclose the body within curly braces even when it consists of a single statement. It is easy to introduce a logic error into a program if additional statements are added to the body later and the programmer forgets to add the then required curly braces.

Consider the following `if` statement that attempts to optionally assign `y`:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> int x = 20, y = 40;
> if (x < 10);
>     y = 2;
> y
2
```

It is important *not* to put a semicolon at the end of the line with the `if`. Here the semicolon terminates the `if` statement, but the indentation implies that the second line is intended to be part of the `if` statement. The compiler interprets the badly formatted `if` statement as if it were written as

Interactions

```
> if (x < 10)
      ;
> x = 2;
```

The semicolon by itself represents an *empty statement* which is legal in Java and sometimes even useful. Here it means the `if` statement has an empty body. The assignment

```
y = 2;
```

is always executed since it is not part of the `if` statement body. The assignment statement is simply an independent statement following the `if` statement.

`TimeConverter2` (5.1) is a variation of `TimeConverter` (4.3) that returns a string in digital timer format (as in 03:41:07):

```java
public class TimeConverter2 {
    public String convert(int seconds) {
        //  First compute hours, minutes, and seconds
        int hours = seconds/3600;
        seconds = seconds % 3600;
        int minutes = seconds/60;
        seconds = seconds % 60;
        //  Next, format the output as 00:00:00
        String result = "";
        if (hours < 10) {  //  Print leading zero, if necessary
            result = result + "0";
        }
        result = result + hours + ":";
        if (minutes < 10) {  //  Print leading zero, if necessary
            result = result + "0";
        }
        result = result + minutes + ":";
        if (seconds < 10) {  //  Print leading zero, if necessary
            result = result + "0";
        }
        result = result + seconds;
        return result;
    }
}
```

Listing 5.1: `TimeConverter2`—converts seconds to hours:minutes:seconds

In `TimeConverter2` (5.1) the `convert()` method works as follows:

- The first part that computes the values of the `hours`, `minutes`, and `seconds` variables remains the same as `TimeConverter` (4.3).
- As before, the second part assembles a string to return using the values computed from the first part. In this version, however, the format is as displayed on a digital timer: 00:00:00. It is easy to build a string consisting hours, minutes, and seconds separated by colons (:)—the string concatenation operator (+) works nicely. The more interesting problem is how to express 2 hours, 47 minutes, and 5 seconds as 02:47:05, instead of 2:47:5. We need to add an extra zero in front of numbers that are less than ten. For the case of hours:

  ```java
  if (hours < 10) {  //  Print leading zero, if necessary
      result = result + "0";
  }
  result = result + hours + ":";
  ```

  we optionally concatenate a zero on the front of hours value. The same concept is applied to minutes and seconds.

As shown in this Interactions session, the method works as desired:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> t = new TimeConverter();
> t.convert(10000)
"02:46:40"
> t.convert(10025)
"02:47:05"
> t.convert(0)
"00:00:00"
> t.convert(1)
"00:00:01"
> t.convert(60)
"00:01:00"
```

5.3 The `if/else` Statement

The `if` statement allows the optional execution of a section of code. Either the code is executed, or it is not. Another common situation is the need to execute of one section of code *or* another; that is, do *this* code *here* or do *that* code *there* depending on a given condition.

The `if` statement has an optional `else` clause. The general form of an `if`/`else` statement is

```
if ( condition )
        if body
else
        else body
```

Figure 5.2 shows the `if`/`else` flowchart.

The `else` clause contains an alternate body that is executed if the condition is false. Consider the following sequence of interactions:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> int x = 20, y = 40, z = 60;
> if (x == 20) {
      y = z;
  } else {
      z = y;
  }
> y
60
> z
60
> y = 40;
> if (x != 20) {
```

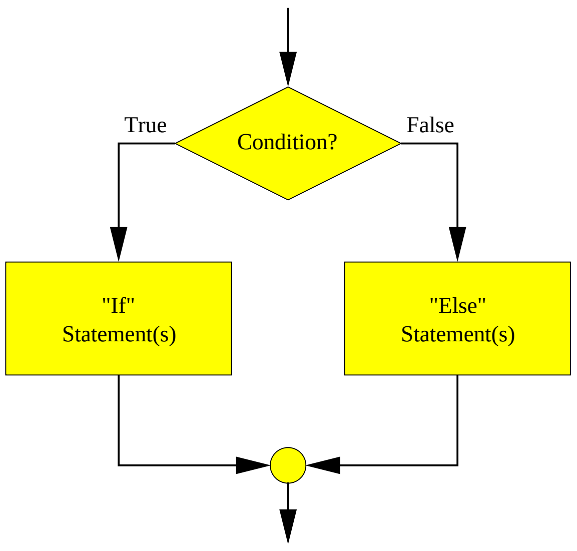

Figure 5.2: Execution flow in an `if/else` statement

```
        y = z;
    } else {
        z = y;
    }
> y
40
> z
40
>
```

The first `if/else` statement assigns `z`'s value to `y`, while the second one assigns `y`'s value to `z`.

The plain `if` statement is really a simplified form of the `if/else` statement. The plain `if` statement

```
if (x < 4) {
    y = 0;
}
```

could be written as

```
if (x < 4) {
    y = 0;
} else {}
```

The plain `if` form is simpler, though, and empty `else` bodies as shown here should not be used.

5.4 Summary

- The relational operators (`==`, `!=`, `<`, `>`, `<=`, and `>=`) evaluate to Boolean values.
- `!` is the unary *not* operator.

- Boolean expressions can be combined via `&&` and `||`.
- The `if` statement can be used to optionally execute statements.
- A block groups a series of statements within a pair of curly braces (`{}`).
- The `if` statement has an optional `else` clause to require the selection between two alternate paths of execution.
- The `if`/`else` statements can be nested to achieve arbitrary complexity.

5.5 Exercises

1. What is a predicate?

2. Can the two symbols that comprise the $\leq$ operator in Java (`<=`) be written reversed (`=<`) and still represent $\leq$?

3. Can a space appear between the two symbols that comprise Java's $\leq$ operator (as in `< =`)?

4. Given that integer variables `x`, `y`, and `z` have the values 1, 2, and 3 respectively, determine the value of each of the following expressions:

a. `x == 2`	b. `x != 2`	c. `x <= 2`	d. `x >= 2`	e.
f. `x < 2`	g. `x > 2`	h. `x < 2 || x == 2`	i. `!(x < y)`	j. `!(x <= y)`
k.	l.	m.	n.	o.
p.	q.	r.	s.	t.

5. Rewrite the following Boolean expressions in simpler form:

a. `!(x == 2)`	b. `x < 2 || x == 2`	c. `!(x < y)`	d. `!(x <= y)`	e.
f.	g.	h.	i.	j.
k.	l.	m.	n.	o.

6. What Java conditional expression correctly represents the mathematical expression $10 \leq \texttt{x} \leq 100$?

7. Rewrite the following code fragment to achieve the same behavior without using an `else`:

```
if (x == y) {
    z = 0;
} else {
    z = 1;
}
```

8. The following code fragment works but is awkward. Rewrite it to achieve the same behavior without using the empty `if` body:

```
if (x == y) {}
else {
    z = 0;
}
```

9.