2.8 Decision Making: Equality and Relational Operators

A **condition** is an expression that can be `true` or `false`. This section introduces Java’s `if` **selection statement**, which allows a program to make a **decision** based on a condition’s value. For example, the condition “grade is greater than or equal to 60” determines whether a student passed a test. If an `if` statement’s condition is *true*, its body executes. If the condition is *false*, its body does not execute.

Conditions in `if` statements can be formed by using the **equality operators** (== and `!=`) and **relational operators** (>, <, >= and <=) summarized in Fig. 2.14. Both equality operators have the same level of precedence, which is *lower* than that of the relational operators. The equality operators associate from *left to right*. The relational operators all have the same level of precedence and also associate from *left to right*.

Algebraic operator	Java equality or relational operator	Sample Java condition	Meaning of Java condition
Equality operators			
`=`	`==`	`x == y`	`x is equal to y`

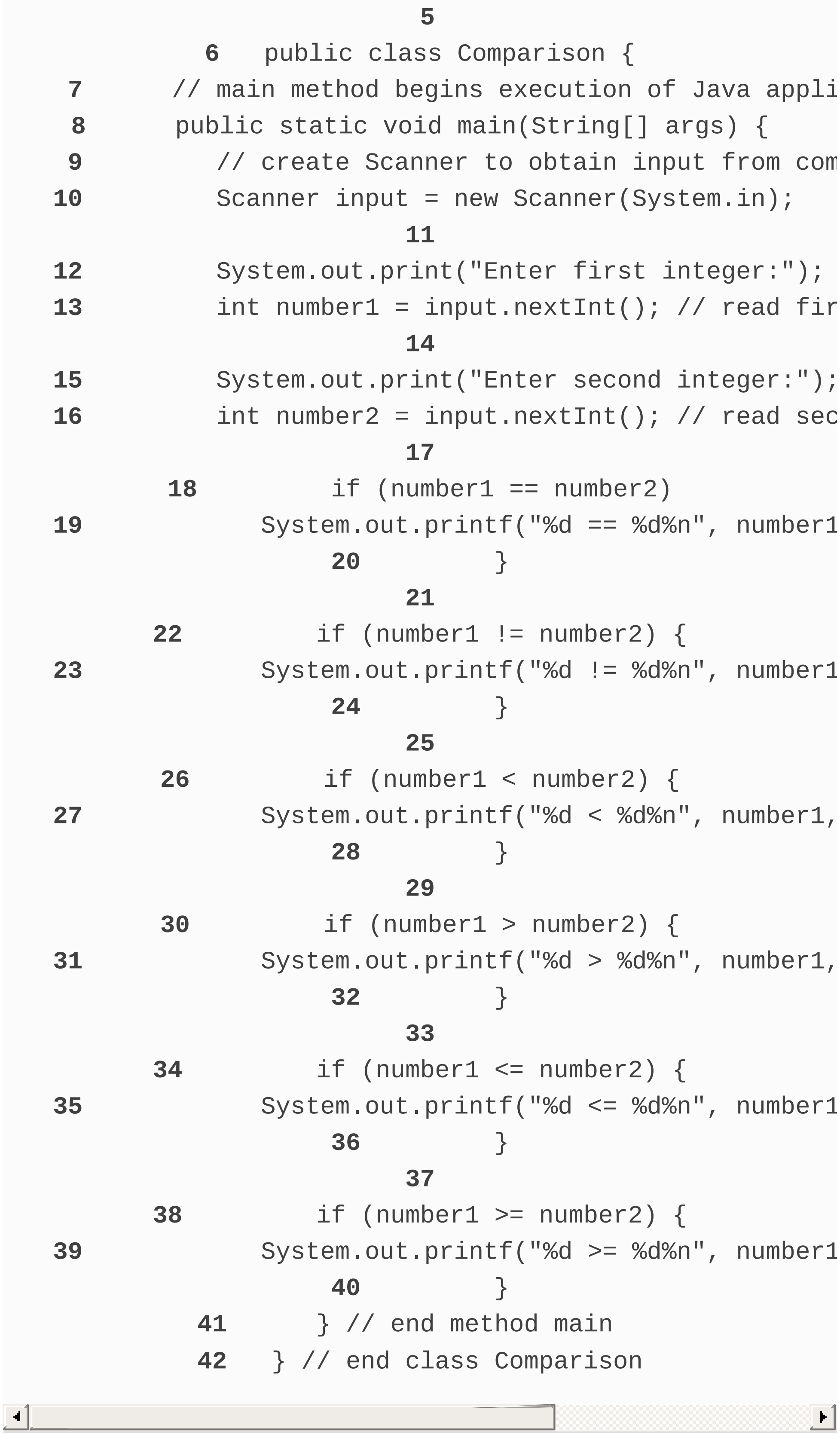
≠	!=	x != y	x is not equal to y
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y

Fig. 2.14

Equality and relational operators.

Figure 2.15 uses six `if` statements to compare two integers input by the user. If the condition in any of these `if` statements is *true*, the statement associated with that if statement executes; otherwise, the statement is skipped. We use a `Scanner` to input the integers from the user and store them in variables `number1` and `number2`. The program *compares* the numbers and displays the results of the comparisons that are true. We show three sample outputs for different values entered by the user.

```
// Fig. 2.15: Comparison.java
// Compare integers using if statements, relatio
// and equality operators.
import java.util.Scanner; // program uses class
```



```

public class Comparison {
   // main method begins execution of Java appli
   public static void main(String[] args) {
      // create Scanner to obtain input from com
      Scanner input = new Scanner(System.in);

      System.out.print("Enter first integer:");
      int number1 = input.nextInt(); // read fir

      System.out.print("Enter second integer:");
      int number2 = input.nextInt(); // read sec

      if (number1 == number2)
         System.out.printf("%d == %d%n", number1
      }

      if (number1 != number2) {
         System.out.printf("%d != %d%n", number1
      }

      if (number1 < number2) {
         System.out.printf("%d < %d%n", number1,
      }

      if (number1 > number2) {
         System.out.printf("%d > %d%n", number1,
      }

      if (number1 <= number2) {
         System.out.printf("%d <= %d%n", number1
      }

      if (number1 >= number2) {
         System.out.printf("%d >= %d%n", number1
      }
   } // end method main
} // end class Comparison
```

```
Enter first integer: 777
Enter second integer: 777
777 == 777
777 <= 777
777 >= 777
```

```
Enter first integer: 1000
Enter second integer: 2000
1000 != 2000
1000 < 2000
1000 <= 2000
```

```
Enter first integer: 2000
Enter second integer: 1000
2000 != 1000
2000 > 1000
2000 >= 1000
```

Fig. 2.15

Compare integers using `if` statements, relational operators and equality operators.

Class `Comparison`’s `main` method (lines 8–41) begins the execution of the program.

Line 10

```
Scanner input = new Scanner(System.in);
```

declares `Scanner` variable `input` and assigns it a `Scanner` that inputs data from the standard input (i.e., the keyboard).

Lines 12–13

```
System.out.print("Enter first integer:"); // prompt
int number1 = input.nextInt(); // read first number f
```

prompt the user to enter the first integer and input the value, respectively. The value is stored in the `int` variable `number1`.

Lines 15–16

```
System.out.print("Enter second integer:"); // prompt
int number2 = input.nextInt(); // read second number
```

prompt the user to enter the second integer and input the value, respectively. The value is stored in the `int` variable `number2`.

Lines 18–20

```
if (number1 == number2) {
System.out.printf("%d == %d%n", number1, number2);
}
```

compare the values of variables `number1` and `number2` to test for equality. `If` the values are equal, the statement in line 19 displays a line of text indicating that the numbers are equal. The if statements starting in lines 22, 26, 30, 34 and 38 compare `number1` and `number2` using the operators `!=`, `<`, `>`, `<=` and `>=`, respectively. If the conditions are `true` in one or more of those `if` statements, the corresponding body statement displays an appropriate line of text.

Each `if` statement in Fig. 2.15 contains a single body statement that's indented. Also notice that we've enclosed each body statement in a pair of braces, `{ }`, creating what's called a **compound statement** or a **block**.

Good Programming Practice 2.11

Indent the statement(s) in the body of an `if` *statement to enhance readability. IDEs typically do this for you, allowing you to specify the indent size.*

Error-Prevention Tip 2.4

You don't need to use braces, `{ }`*, around single-statement bodies, but you* must *include the braces around multiple-*

statement bodies. You’ll see later that forgetting to enclose multiple-statement bodies in braces leads to errors. To avoid errors, as a rule, always enclose an `if` *statement’s body statement(s) in braces.*

Common Programming Error 2.7

Placing a semicolon immediately after the right parenthesis after the condition in an `if` *statement is often a logic error (although not a syntax error). The semicolon causes the body of the* `if` *statement to be empty, so the* `if` *statement performs no action, regardless of whether or not its condition is true. Worse yet, the original body statement of the* `if` *statement* always *executes, often causing the program to produce incorrect results.*

White Space

Note the use of white space in Fig. 2.15. Recall that the compiler normally ignores white space. So, statements may be split over several lines and may be spaced according to your preferences without affecting a program’s meaning. It’s incorrect to split identifiers and strings. Ideally, statements should be kept small, but this is not always possible.

Error-Prevention Tip 2.5

A lengthy statement can be spread over several lines. If a single statement must be split across lines, choose natural breaking points, such as after a comma in a comma-separated list, or after an operator in a lengthy expression. If a statement is split across two or more lines, indent all subsequent lines until the end of the statement.

Operators Discussed So Far

Figure 2.16 shows the operators discussed so far in decreasing order of precedence. All but the assignment operator, `=`, associate from *left to right*. The assignment operator, =, associates from *right to left*. An assignment expression's value is whatever was assigned to the variable on the = operator's left side—for example, the value of the expression `x = 7` is `7`. So an expression like `x = y = 0` is evaluated as if it had been written as `x = (y = 0)`, which first assigns the value `0` to variable `y`, then assigns the result of that assignment, `0`, to `x`.

Operators	**Associativity**	**Type**
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality

=	right to left	assignment

Fig. 2.16

Precedence and associativity of operators discussed.

Good Programming Practice 2.12

When writing expressions containing many operators, refer to the operator precedence chart (Appendix A). Confirm that the operations in the expression are performed in the order you expect. If, in a complex expression, you’re uncertain about the order of evaluation, use parentheses to force the order, exactly as you’d do in algebraic expressions.