Forming Conditions (Chapter 5 Notes)

- Order of a program’s instructions: flow of control

- 4 types of flow control: sequential execution, method calls, selection, and looping.

- Selection gives us a way to test for certain conditions and to select the instructions to execute based on the results of the test. To perform selection, Java provides a number of alternatives: if, if/else, if/else if, the conditional operator (?:), and switch.

- Java provides equality, relational, and logical operators to evaluate and test whether an expression is true or false. It also provides selection statements to transfer control to a different part of the program depending on the result of that test.

- To compare variables of primitive data types, Java provides the equality operators. Both are binary operators, meaning that they take two operands. The operands may be expressions that evaluate to a primitive numeric or Boolean type or an object reference. The result of an expression and its two operands is a Boolean value, that is, true or false.

For instance, if an int variable “age” holds the value 32, then

The expression ( age == 32 ) will evaluate to true, and

The expression ( age != 32 ) will evaluate to false

Equality Operators

|  |  |  |
| --- | --- | --- |
| Equality Operator | Type | Meaning |
| == | binary | Is equal to |
| != | binary | Is not equal to |

- The following expression can be used to eliminate seniors by testing whether the value of the int variable yearsInCollege is not equal to 4:

yearsInCollege != 4

- The following expression can be used in a game program to determine whether the user wants to play again:

playAgain == ‘y’

Assuming the user’s input is stored in the char variable playAgain, then if the user typed ‘y’, the expression evaluates to true. With any other input value, the expression evaluates to false.

- Although the equality operators can be used to compare object references, these operators cannot be used to compare object data.

- A common operation in a program is to test whether a combination of conditions is true or false. For these operations, Java provides the logical operators !, &&, and ||, which correspond to the Boolean logic operators NOT, AND, and OR. These operators take Boolean expressions as operands.

- A Boolean expression can be a combination of variables, operators, and method calls that result in a Boolean value.

- The NOT operator (!) takes one Boolean expression as an operand and inverts the value of that operand. If the operand is true, the result will be false. If the operand is false, the result will be true.

- The AND operator (&&) takes two Boolean expressions as operands; if both operands are true, then the result will be true; otherwise, the result will be false.

- The OR operator (||) takes two Boolean expressions as operands; if both operands are false, then the result will be false; otherwise, the result will be true. The OR operator consists of two vertical bars with no intervening space.

|  |  |  |
| --- | --- | --- |
| Logical operator | Type | Meaning |
| ! | unary | NOT |
| && | binary | AND |
| || | binary | OR |

Order of precedence of the relational, logical , and arithmetic operators.

|  |  |  |
| --- | --- | --- |
| Operator hierarchy | Order of same-statement  evaluation | Operation |
| ( ) | Left to right | Parentheses for explicit grouping |
| ++, - - | Right to left | Shortcut postincrement / postdecrement |
| ++, - - , ! | Right to left | Shortcut preincrement/postdecrement, logical unary NOT |
| \*, / , % | Left to right | Multiplication, division, modulus |
| + , - | Left to right | Addition or String concatenation, subtraction |

|  |  |  |
| --- | --- | --- |
| Operator hierarchy (continued) | Order of same-statement  evaluation | Operation |
| <, <=, >, >= | Left to right | Relational operators: less than, less than or equal to, greater than, greater than or equal to |
| ==, != | Left to right | Equality operators: equal to and not equal to |
| && | Left to right | Logical AND |
| || | Left to right | Logical OR |
| =, +=, - =, \*=, /=, %= | Right to left | Assignment operator and shortcut assignment operators |

Boolean test;

test = ( age > 18 || age < 25 ) ;

System.out.println( test );

If the age is between 18-25, the output is true. Otherwise, it is false.

DeMorgan’s Laws

NOT ( A AND B ) = ( NOT A ) OR ( NOT B )

NOT ( A OR B ) = ( NOT A ) OR ( NOT B )

The simple selection pattern is appropriate when our program needs to perform an operation for one set of data, but not for all other data. For this situation, we use a simple if statement, which has this pattern:

If ( condition )

{

True block

}

Next statement

The true block can contain one or more statements and is executed only if the condition evaluates to true. After the true block executes, the instruction following the if statement is executed. If the condition is false, the true block is skipped, and execution picks up after the if statement. Indenting clarifies the structure of the program.

The second form of an if statement is appropriate when the data falls into two mutually exclusive categories and different instructions should be executed for each category. For these situations, we use an “if/else statement”, which has the following pattern:

If ( condition )

{

True block

}

Else

{

False block

}

Next statement

Diagram

Description automatically generated

If the condition evaluates to true, the true block is executed and the false block is skipped. If the condition evaluates to false, the true block is skipped and the false block is executed. In either situation, the statement following the if statement is executed next.

Notice the indentation of the true and false blocks and that the else and curly braces line up under the if. This coding style makes it easy to see which statements belong to the true block and which belong to the false block. If the indentation is incorrect, a reader of our program may misunderstand which statement will be executed. In any event, the compiler ignores the indentation; the indentation is only designed for humans to understand the logic of the code.

An if/else statement simplifies our processing and avoids unnecessarily testing two conditions when only one of the conditions is true.

The scope of a variable is a region within a program where the variable can be referenced or used. When we declare a variable, its scope extends from the point at which it is declared until the end of the block in which we declared it. A method, such as the main, is a block. The true blocks and false blocks for if statements are also blocks.

We must declare variables or objects outside of selections, only then can we use them within a selection.

The last form of an if statement is appropriate when the data falls into more than two mutually exclusive categories and the appropriate instructions to execute are different for each category. For this situation, Java provides the “if/else if” statement.

The if/else if statement follows this pattern:

if ( condition 1 )

{

True block for condition 1

}

Else if ( condition 2 )

{

True block for condition 2

}

Else if ( condition n )

{

True block for condition n

}

Else

{

False block for all conditions being false

}

The flow of control for this form is shown in figure 5.8

There can be any number of conditions in an if/else if statement. As we can see, once a condition evaluates to true for any value, control moves to the true block for that condition, then skips the remainder of the conditions, continuing execution at any statement that follows the if/else if statement. The final false block (along with the final else) is optional and is executed only when none of the conditions evaluates to true. Note that if the final else is used, then the condition it covers is not coded. The else stands alone on the line.

When we need the results of one if statement’s processing before we can evaluate the next condition, we can write multiple if statements either sequentially or nested within other if statements.

To illustrate sequential if statements, let’s look at the problem of finding the smallest of three numbers. We can use multiple sequential if statements. First we find the smaller of the first two numbers, then we find the smaller of that result and the third number. The pseudocode for this application is:

Read number1

Read number2

Read number3

If number1 is less than number2

Smallest is number1

Else

Smallest is number2

If number3 is less than smallest

Smallest is number3

When we need the results of one “if” statement’s processing before we can evaluate the next conditions, we can write multiple “if” statements either sequentially or nested within other if statements.

Finding the minimum or maximum values

To illustrate sequential if statements, let’s look at the problem of finding the smallest of three numbers. We can use multiple, sequential if statements. First we find the smaller of the first two numbers, then we find the smaller of that result and the third number. The pseudocode for this application is:

Read number1

Read number2

Read number3

If number1 is less than number2

Smallest is number1

Else

Smallest is number2

If number 3 is less than smallest

Smallest is number3

import java.util.Scanner;

public class smallestOfThree {

public static void main(String[] args)

{

Scanner input = new Scanner(System.in);

int num1 = 0;

int num2 = 0;

int num3 = 0;

int smallest = 0;

System.out.print("\nEnter your first integer: ");

num1 = input.nextInt();

System.out.println();

System.out.print("Enter your second integer: ");

num2 = input.nextInt();

System.out.println();

System.out.print("Enter your third integer: ");

num3 = input.nextInt();

if (num1 < num2)

{

smallest = num1;

}

else

{

smallest = num2;

}

if (num3 < smallest)

{

smallest = num3;

}

System.out.println("\n" + smallest + " is the smallest integer of the three.\n");

}

}

If statements can be written as part of a true or false block of another if statement. These are called nested if statements. Typically, we nest if statements when more information is required beyond the results of the first if statement.

One difficulty that arises with nested if statements is specifying which else clause pairs with which if statement, especially if some “if” statements have “else” clauses and others do not. The compiler matches any else clause with the most previous if statement that doesn’t already have an else clause. If this matching is not what we want, we can use curly braces to specify the desired “if else” pairing.

In this following code, we have one if statement nested within another if statement:

If (x == 2)

if(y == x)

System.out.println(“x and y equal 2”);

Else

System.out.println(“x equals 2, but y does not”);

Without curly braces, the entire second if statement comprises the true block of the first condition (x==2), and the else is paired with the second condition (y==x), because this is the most previous if condition that doesn’t have an else.

However, we can force the else clause to be paired with the first condition by using curly braces:

If (x == 2)

{

if(y == x)

System.out.println(“x and y equal 2”);

}

Else

System.out.println(“x equals 2, but y does not”);

With the curly braces added, the if condition (y==x), along with it’s true block, becomes the complete true block of the for the condition (x==2), and the else clause now belongs to the first if condition (x==2).