UNIT 3

Boolean Expressions and if Statements

IN THIS UNIT

Summary: Conditional statements allow programs to flow in one or more directions based on the outcome of a Boolean expression. Relational operators and logical operators allow the programmer to construct those conditional statements and act on their result accordingly. The ability to use conditionals is an important concept to master and will be used often in your programs.

KEY IDEA

Key Ideas

- ✪ Relational and logical operations are used when decisions occur.
- ✪ Conditional statements allow for branching in different directions.
- ✪ An `if` statement is used for one-way selection.
- ✪ An `if-else` statement is used for two-way selection.
- ✪ A nested `if` statement is used for multi-way selection.
- ✪ Short-circuit evaluation can be used to speed up the evaluation of compound conditionals.
- ✪ Curly braces are used to denote a block of code.
- ✪ Indentation within a conditional helps the reader know which piece of code gets executed when the condition is evaluated as true and which gets executed when the condition is false.

Introduction

Up to this point, program execution has been sequential. The first statement is executed, then the second statement, then the third statement, and so on. One of the essential features of programs is their ability to execute certain parts of code based on conditions. In this unit you will learn the various types of conditional statements, along with the relational and logical operators that are used.

Relational Operators

Double Equals Versus Single Equals

Java compares numbers the same way that we do in math class, using **relational operators** (like greater than or less than). A **condition** is a comparison of two values using a relational operator. To decide if two numbers are equal in Java, we compare them using the **double equals**, ==. This is different from the **single equals**, =, which is called an **assignment operator** and is used for assigning values to a variable.

The not Operator: !

To compare if two numbers are **not equal** in Java, we use the **not operator**. It uses the exclamation point, !, and represents the opposite of something. It is also referred to as a **negation** operator. Some Java developers call it a "bang" operator.

Table of Relational Operators

The following table shows the symbols for comparing numbers using relational operators.

Comparison	Java Symbol
Greater than	>
Less than	<
Greater than or equal to	>=
Less than or equal to	<=
Equal to	==
Not equal to	!=

Examples

Demonstrating relational operators within conditional statements:

```
int a = 6;
System.out.println(a == 10);  // false is printed;  6 is not equal to 10
System.out.println(a < 10);   // true is printed;   6 is less than 10
System.out.println(a >= 10);  // false is printed; 6 is not greater than 10
System.out.println(a != 10);  // true is printed;   6 is not equal to 10
```

The ! as a Toggle Switch

The `not` operator, !, can be used in a clever way to reverse the value of a boolean.

Example

Using the `not` operator to toggle player turns for a two-player game:

```
boolean playerOneTurn = true;          // playerOneTurn is true
playerOneTurn = !playerOneTurn;        // playerOneTurn is false
playerOneTurn = !playerOneTurn;        // playerOneTurn is true
```

Logical Operations

Compound Conditionals: Using AND and OR

AND and **OR** are **logical operators**. In Java, the AND operator is typed using two ampersands (&&) while the OR operator uses two vertical bars or pipes (||). These two logical operators are used to form **compound conditional statements.**

Logical Operator	Java Symbol	Compound Conditional	Result with Explanation
AND	&&	*condition1 && condition2*	True only if both conditions are true. False if either condition is false.
OR	||	*condition1 || condition2*	True if either condition is true. False only if both conditions are false.

The Truth Table

A **truth table** describes how AND and OR work.

First Operand (A)	Second Operand (B)	A && B	A || B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Example 1

Suppose you are writing a game that allows the user to play as long as their score is less than the winning score and they still have time left on the clock. You would want to allow them to play the game as long as *both* of these two conditions are true.

```
boolean continuePlaying = (score < winningScore && timeLeft > 0);
```

Example 2

Computing the results of compound conditionals:

```
int a = 3, b = 4, c = 5;
System.out.println(a <= 3 && b != 4);               // false is printed
System.out.println(b % 2 == 1 || c / 2 == 1);       // false is printed
System.out.println(a > 2 && (b > 5 || c < 6));      // true is printed
```

Explanation: `(3 <= 3) && (4 != 4)` → `true && false` → `false`
Explanation: `(4 % 2 == 1) || (5 / 2 == 1)` → `false || false` → `false`
Explanation: `(3 > 2) && (4 > 5 || 5 < 6)` → `true && (false || true)`
→ `true && true` → `true`

Fun Fact: *The boolean variable is named in honor of George Boole, who founded the field of algebraic logic. His work is at the heart of all computing.*

A truth table lists every possible combination of operand values and the result of the operation for each combination. It is also used to determine if two logical expressions are equivalent.

Example 1

Determine if `!(a && b)` is equivalent to `!a || !b`

Let's set up a truth table for the first expression `!(a && b)`.

a	b	a && b	!(a && b)
True	True	True	False
True	False	False	True
False	True	False	True
False	False	False	True

And set up a truth table for the second expression `!a || !b`.

a	b	!a	!b	!a || !b
True	True	False	False	False
True	False	False	True	True
False	True	True	False	True
False	False	True	True	True

Since the last columns in both tables are the same, we can say that the two expressions are equivalent. They both evaluate to the same value in all cases. This is a proof of the identity known as De Morgan's Law.

Short-Circuit Evaluation

Java uses **short-circuit evaluation** to speed up the evaluation of compound conditionals. As soon as the result of the final evaluation is known, then the result is returned and the remaining evaluations are not even performed.

Find the result of: `(1 < 3 || (a >= c - b % a + (b + a / 7) - (a % b +c)))`

In a *split second,* you can determine that the result is true *because 1 is less than 3.* End of story. The rest is never even evaluated. Short-circuit evaluation is useful when the second half of a compound conditional might cause an error. This is tested on the AP Computer Science A Exam.

Example

Demonstrate how short-circuit evaluation can prevent a division by zero error.

```
int count = 0;
int total = 0;
boolean result = (count != 0 && total/count > 0);   // result is false
```

If the count is equal to zero, the second half of the compound conditional is never evaluated, thereby avoiding a division by zero error.

De Morgan's Law

On the AP Computer Science A Exam, you must be able to evaluate complex compound conditionals. **De Morgan's Law** can help you decipher ones that fit certain criteria.

Compound Conditional	Applying De Morgan's Law
`!(a && b)`	`!a || !b`
`!(a || b)`	`!a && !b`

An easy way to remember how to apply De Morgan's Law is to think of the distributive property from algebra, but with a twist. Distribute the `!` to both of the conditionals and also change the logical operator. Note the use of the parentheses and remember that the law can be applied both forward and backward.

Example

Computing the results of complex logical expressions using De Morgan's Law:

```
int a = 2, b = 3;
boolean result1 = !(b == 3 && a < 1);       // result1 is true
boolean result2 = !(a != 2 || b <= 4);      // result2 is false
```

Explanation for `result1`: De Morgan's Law says that you should *distribute* the negation operator. Therefore, the negation of `b == 3` is `b != 3` and the negation of `a < 1` is `a >= 1`:

```
!(b == 3 && a < 1) ➔ (b != 3) || (a >= 1) ➔ (3 != 3) || (2 >= 1)
                                          ➔ false || true ➔ true
```

Explanation for `result2`: The negation of `a != 2` is `a == 2` and the negation of `b <= 4` is `b > 4`:

```
!(a != 2 || b <= 4) ➔ (a == 2) && (b > 4) ➔ (2 == 2) && (3 > 4)
                                          ➔ true && false ➔ false
```

The Negation of a Relational Operator

The negation of *greater than* (>) is *less than or equal to* (<=) and vice versa.
The negation of *less than* (<) is *greater than or equal to* (>=) and vice versa.
The negation of *equal to* (==) is *not equal to* (!=) and vice versa.

Precedence of Java Operators

We already know the precedence of the mathematical operators, but what is the precedence among the logical operators and what if a statement has multiple operators? Luckily Java has well-defined rules for their order.

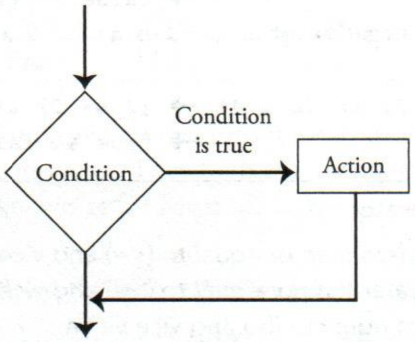
Operation	Symbol	Precedence
Postfix	`++, --`	1
Unary	`+, -,`	2
logical NOT	`!`	2
Multiplication, Division, Modulus	`*, /, %`	3
Addition, Subtraction	`+, -`	4
Relational Operators	`>, <, >=, <=`	6
Equality	`==, !=`	7
logical AND	`&&`	11
logical OR	`||`	12
Assignment	`=, +=, -=, *=, /=, %=`	14

Conditional Statements

If you want your program to branch out into different directions based on the input from the user or the value of a variable, then you will want to have a **conditional** statement in your program. A conditional statement will interrupt the sequential execution of your program. The conditional statement affects the flow of control by executing different statements based on the value of a Boolean expression.

The `if` Statement

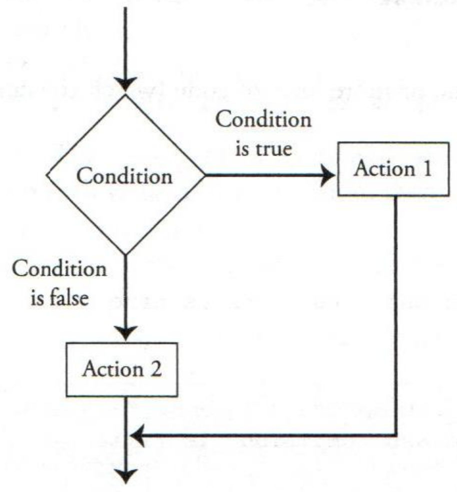
The **if** statement is a conditional statement. In its simplest form, it allows the program to execute a specific set of instructions if some certain condition is met. If the condition is not met, then the program skips over those instructions. This is also known as a one-way selection.



The `if-else` Statement

The **if-else** statement allows the program to execute a specific set of instructions if some certain condition is met and a different set of instructions if the condition is not met.

This is also known as a two-way selection. I will sometimes refer to the code that is executed when the condition is true as the *if clause* and the code that follows the else as the *else clause.*



The syntax for `if` and `if-else` statements can be tricky. Pay attention to the use of the curly braces and indentation. The indentation helps the reader (not the compiler) understand what statements are to be executed if the condition is true.

Example 1
The `if` statement for one or more lines of code:

```
if (condition)
{
   // one or more instructions to be performed when condition is true
}
```

Example 2
The `if-else` statement for one or more lines of code for each result:

```
if (condition)
{
   // instructions to be performed when condition is true
}
else
{
   // instructions to be performed when condition is false
}
```

Example 3
The `if-else` statement using a compound conditional:

```
if (condition1 && condition2)
{
   // condition1 and condition2 are both true
}
else
{
   // either condition1 or condition2 is false (or both are false)
}
```

Nested `if` Statements

Instead of using a compound conditional in an `if-else` statement, a nested `if` can be used. This is known as a multi-way selection.

Example

Nested `if-else` statements for one or more lines of code (watch the curly braces!):

```
if (condition1)
{
   // condition1 is true
   if (condition2)
   {
      // condition1 is true and condition2 is true
   }
   else
   {
      // condition1 is true and condition2 is false
   }
}
else
{
   // condition1 is false
   if (condition3)
   {
      // condition1 is false and condition3 is true
   }
   else
   {
      // condition1 is false and condition3 is false
   }
}
```

if-else-if ladder

In an `if-else-if` ladder, conditions are evaluated from the top. Once a condition is found to be true, execution continues to the statement after the ladder. You can have as many conditions that you need in your ladder.

Example

```
if (condition1)
{
   // one or more instructions to be performed when condition1 is true
}
else if (condition2)
{
   //  one or more instructions to be performed when condition2 is true
}
else if (condition3)
{
   //  one or more instructions to be performed when condition3 is true
}
else
{
   //  one or more instructions to be performed when condition3 is false
}
```

Curly Braces

When you want to execute only one line of code when something is true, you don't actually *need* the curly braces. Java will execute the first line of code after the `if` statement if the result is true.

```
if (condition)
   // single instruction to be performed when condition is true
```

The same goes for an `if-else` statement. If you want to execute only one line of code for each value of the condition, then no curly braces are required.

```
if (condition)
   // single instruction to be performed when condition is true
else
   // single instruction to be performed when condition is false
```

ERROR

Can You Spot the Error in This Program?

```
if (condition);
{
   // instructions to be performed when condition is true
}
```

Answer: Never put a semicolon on the same line as the condition. It ends the `if` statement right there. In general, *never put a semicolon before a curly brace.*

The Dangling `else`

If you don't use curly braces properly within `if-else` statements, you may get a **dangling else**. An `else` attaches itself to the nearest if statement and not necessarily to the one that you may have intended. Use curly braces to avoid a dangling `else` and control which instructions you want executed. The dangling `else` does not cause a compile-time error, but rather it causes a **logic error**. The program doesn't execute in the way that you expected.

Example

The second `else` statement attaches itself to the nearest preceding `if` statement:

```
if (condition1)
   // instruction performed when condition1 is true
   if (condition2)
      // instruction performed when condition1 is true and condition2 is true
   else
      // instruction performed when condition1 is true and condition2 is false
```

Scope of a Variable

The word **scope** refers to the code that knows that a variable exists. Local variables, like the ones we have been declaring, are only known within the block of code in which they are defined. This can be confusing for beginning programmers. Another way to say it is: a variable that is declared inside a pair of curly braces is only known inside that set of curly braces.

Curly Braces and Blocks of Code

Curly braces come in pairs and the code they surround is called a **block of code**.

```
{
   // A group of instructions inside a pair of curly braces
   // is called a "block of code". Variables declared inside a block
   // of code are only known inside that block of code.
}
```

› Rapid Review

Logic

- The relational operators in Java are `>`, `>=`, `<`, `<=`, `==`, and `!=`.
- A condition is an expression that evaluates to either true or false.
- The logical operator AND is coded using two ampersands, &&.
- The && is true only when both conditions are true.
- The logical operator OR is coded using two vertical bars, ||.
- The || is true when either condition or both are true.
- Programmers use logical operators to write compound conditionals.
- The NOT operator (negation operator) is coded using the exclamation point, !.
- The ! can be used to flip-flop a boolean.
- De Morgan's Law states: !(A && B) = !A || !B and also !(A || B) = !A && !B.
- When evaluating conditionals, the computer uses short-circuit evaluation.

Conditional Statements

- The `if` and the `if-else` are called conditional statements since the flow of the program changes based upon the evaluation of a condition.
- Curly braces come in pairs and the code they contain is called a block of code.
- Curly braces need to be used when more than one instruction is to be executed for `if` and `if-else` statements.
- The scope of a variable refers to the code that knows that the variable exists.
- Variables declared within a conditional are only known within that conditional.

› Review Questions

Basic Level

1. Consider the following code segment.

```
int num1 = 9;
int num2 = 5;
if (num1 > num2)
{
    System.out.print((num1 + num2) % num2);
}
else
{
    System.out.print((num1 - num2) % num2);
}
```

What is printed as a result of executing the code segment?

(A) `0`
(B) `2`
(C) `4`
(D) `9`
(E) `14`

2. Consider the following code segment.

```
int value = initValue;
if (value > 10)
   if (value > 15)
      value = 0;
else
   value = 1;
System.out.println("value = " + value);
```

Under which of the conditions below will this code segment print `value = 1`?

(A) `initValue = 8;`
(B) `initValue = 12;`
(C) `initValue = 20;`
(D) Never. `value = 0` will always be printed.
(E) Never. Code will not compile.

3. Assume that `a`, `b`, and `c` have been declared and initialized with `int` values. The expression

```
!(a > b || b <= c)
```

is equivalent to which of the following?

(A) `a > b && b <= c`
(B) `a <= b || b > c`
(C) `a <= b && b > c`
(D) `a < b || b >= c`
(E) `a < b && b >= c`

4. Which conditional statement is equivalent to the expression shown below?

```
if ((temp > 80) && !(80 < temp))
```

(A) `if (temp == 80)`
(B) `if (temp != 80)`
(C) `if ((temp >= 80) || (temp > 80))`
(D) `false`
(E) `true`

5. A high school class places students in a course based on their average from a previous course. Students that have an average 92 or above are placed in an honors level course. Students that have an average below 74 are placed in a remedial level course. All other students are placed in a regular level course. Which of the following statements correctly places the student based on their average?

I.
```
if (average >= 92)
   level = "honors";
if (average < 74)
   level = "remedial";
else
   level = "regular";
```

II.
```
if (average >= 92)
   level = "honors";
else if (average >= 74)
   level = "regular";
else
   level = "remedial";
```

III.
```
if (average >= 92)
   level = "honors";
else if ((average >= 74) && (average < 92))
   level = "regular";
else
   level = "remedial";
```

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) II and III only

Advanced Level

6. Assume that `x`, `y`, and `z` have been declared as follows:

```
boolean x = true;
boolean y = false;
boolean z = true;
```

Which of the following expressions evaluates to true?

(A) `x && y && z`
(B) `x && y || z && y`
(C) `!(x && y) || z`
(D) `!(x || y) && z`
(E) `x && y || !z`

7. Consider the following code segment.

```
int amount = initValue;
if (amount >= 0)
   if (amount <= 10)
      System.out.println("**");
else
   System.out.println("%%");
```

Under what condition will this code segment print %%?

(A) initValue = −5
(B) initValue = 0
(C) initValue = 5
(D) initValue = 10
(E) initValue = 15

8. Which segment of code will correctly print the two Strings `name1` and `name2` in alphabetical order?

I.
```
if (name1 < name2)
   System.out.println(name1 + " " + name2);
else
   System.out.println(name2 + " " + name1);
```

II.
```
if (name1.compareTo(name2))
    System.out.println(name1 + " " + name2);
else
    System.out.println(name2 + " " + name1);
```

III.
```
if (name1.compareTo(name2) < 0)
    System.out.println(name1 + " " + name2);
else
    System.out.println(name2 + " " + name1);
```

(A) I only
(B) II only
(C) III only
(D) II and III only
(E) I, II, and III