In the previous chapter, we explored the fundamentals of Java, including variables, data types, and basic operations. Programs written with only these concepts execute sequentially, one line after another. This chapter introduces **decision structures**, which empower your programs to make choices and follow different paths based on specific conditions. This control over the flow of execution is a fundamental concept in programming.

What you will learn

In this chapter, you will learn how to:

- Understand and use the `if`, `if-else`, and `if-else-if` statements to control program flow.
- Form conditions using **Relational and Logical Operators**.
- Implement complex logic with **nested `if` statements**.
- Use boolean variables as **flags** to track program states.
- Compare primitive `char` types and non-primitive `String` objects.
- Use the **conditional (ternary) operator** for concise `if-else` logic.
- Create multi-branch decisions with the **`switch` statement**.
- Format output precisely using `System.out.printf` and `String.format`. # 3.1 The `if` Statement, Boolean Expressions, and Relational Operators

The `if` statement is the most basic decision structure. It uses a **Boolean expression** to conditionally execute a block of code. If the expression evaluates to `true`, the code block runs; otherwise, it is skipped.

```
int score = 85;

if (score > 59) {
    System.out.println("Congratulations, you passed!");
}
```

Using Relational Operators to Form Conditions

To create these conditions, `if` statements rely on **Boolean expressions**. A Boolean expression is a statement that evaluates to either `true` or `false`. These expressions are often formed using **relational operators** to compare values.

Operator	Meaning	Example	Result (if x=10, y=5)
`>`	Greater than	`x > y`	`true`
`<`	Less than	`x < y`	`false`
`>=`	Greater than or equal to	`x >= 10`	`true`
`<=`	Less than or equal to	`y <= 5`	`true`
`==`	Equal to	`x == y`	`false`
`!=`	Not equal to	`x != y`	`true`

> **Keep exploring!** Relational operators are one type of **Operators in Java** (alongside Arithmetic, Assignment, Unary, Logical, Ternary, a). This article features a full list of here's a full list of Java operators.

Boolean expressions

Boolean expressions use Relational Operators to evaluate whether a statement is true or false. Here are some examples:

- `is 4 < 3?` **false**
- `is 25 > 5 ?` **true**

Program execution is determined by the result of Boolean expressions. ## Programming Style and the `if` Statement

A block of code is enclosed in a set of curly braces `{ }`. While Java allows you to omit the curly braces if there is only a single statement to be executed, it is highly recommended to **always use them**.

This is valid, but not recommended:

```
if (score > 59)
    System.out.println("You passed!");
```

This is the recommended style:

```
if (score > 59) {
    System.out.println("You passed!");
}
```

Using braces consistently improves readability, helps you understand code scope, and is required if you need to execute multiple statements conditionally.

> **a worthwhile aside**: I prefer to use Egyptian Brackets (also called K&R style), which are seen in the above example. This is the style used by the Brian Kernighan and Dennis Ritchie in their legendary textbook The C Programming Language in 1978.
>
> The alternative is where each brace gets its own line. Read more about Notable Indentation Styles. See the Allman (also called BSD) Style.

Boolean Flags

A **flag** is a `boolean` variable used to signal whether a condition exists. This is a common and important practice that allows you to check the current state of an operation.

For example, we can set a `passingScore` flag to `true` only if a student passes. Later in the program, we can check this flag to perform another action.

```java
boolean passingScore = false;
double score = 72.5;

if (score > 59.9) {
    passingScore = true;
}

// Some other code might be here...

if (passingScore) {
    System.out.println("This student is eligible for the next course.");
}
```

This technique is useful in many scenarios, such as tracking whether a user is logged in. An application can show limited content when a `userLoggedIn` flag is `false` and reveal more options once it becomes `true`.

Comparing Characters

Because `char` is a primitive type, its value is stored directly in memory as a Unicode number. This means you can use relational operators to compare characters based on their numeric order.

- 'B' is greater than 'A' because its Unicode value is higher.
- 'a' is greater than 'A' because uppercase letters have lower Unicode values than lowercase letters.

  Reference note: The Wikipedia article, List of Unicode Characters, lists the numerical (decimal) values of each Unicode character.

  The capital letter 'A' has a decimal value of 65, while 'B' has a value of 66, and so forth. The lowercase letter 'a' has a decimal value of 97, while 'b' has a value of 98.

```java
char letter = 'B';

if (letter > 'A') {
    System.out.println("The letter is after A in the alphabet.");
}

// Example from quiz3.md
char grade = 'C';
if (grade != 'A') { // This correctly checks if grade is NOT 'A'
    System.out.println("This is not an A grade.");
}
```

This is a correct way to determine if the `char` variable `grade` is not equal to 'A'.

One takeaway from this section is that characters are compared using relational

operators because `char` is a primitive data type, like `int` or `double`. Strings are Java objects, or non-primitive data types, so we use a String method (`.equals()` or `.equalsIgnoreCase()`) to compare Strings.

Check Your Understanding

1. What is the primary purpose of an `if` statement in a program?
2. A Boolean expression, which is required by an `if` statement, must evaluate to what two possible values?
3. Explain the difference between the `=` and `==` operators. Why is using `=` inside an `if` condition a common error?
4. Why is it highly recommended to always use curly braces `{}` with `if` statements, even for a single line of code?
5. What is a boolean "flag"? Describe a real-world scenario (other than user login) where you might use a flag to control a program's logic.

Practice Problems

1. Write a Java `if` statement that checks if the value of an `int` variable `temperature` is greater than 98.6. If it is, print "You have a fever."
2. Given `char initial = 'D';`, write an `if` statement that prints "This is one of the first four letters of the alphabet" if `initial` is 'A', 'B', 'C', or 'D'. (Hint: you can use `||` or multiple `if` statements, but think about which is more efficient). # 3.2 The `if-else` and `if-else-if` Statements

The `if-else` Statement

The `if-else` statement provides an alternative path of execution. If the boolean expression is `true`, the `if` block executes. If it is `false`, the `else` block executes. The two blocks are mutually exclusive; one or the other will run, but never both.

```
double score = 55.0;

if (score > 59.9) {
    System.out.println("You passed the class.");
} else {
    System.out.println("You did not pass the class.");
}
```

Nested `if` Statements

An `if` statement can be placed inside another `if` or `else` block. This is called a **nested `if` statement** and is used for complex decisions where one condition depends on another.

```
// Program determines if an applicant meets two conditions to qualify for a loan.
double salary = 50000;
```

```java
int yearsOnJob = 3;

if (salary >= 40000) {
    if (yearsOnJob >= 2) {
        System.out.println("You qualify for the loan.");
    } else {
        System.out.println("You must have been on your current job for at least two years.")
    }
} else {
    System.out.println("You must earn at least $40,000 per year.");
}
```

While useful, deeply nested `if` statements can become hard to read. Often, they can be simplified using logical operators.

The `if-else-if` Statement

The `if-else-if` statement is perfect for testing a series of mutually exclusive conditions. The program evaluates each condition in order and executes the block for the **first one that is `true`**. Once a condition is met, the rest of the statement is skipped.

This structure is more efficient than a series of independent `if` statements, because in that case, every `if` statement would be checked, regardless of the outcome of the others.

```java
int testScore = 78;
char grade;

if (testScore < 60) {
    grade = 'F';
} else if (testScore < 70) {
    grade = 'D';
} else if (testScore < 80) {
    grade = 'C';
} else if (testScore < 90) {
    grade = 'B';
} else {
    grade = 'A';
}
System.out.println("Your grade is " + grade); // Prints: Your grade is C
```

An important point to remember when learning if-else-if statements is that every condition except the final `else` statement includes a Boolean expression. `else` is the catch-all if no condition has been met through the series of `if` and `else if` statements.

Check Your Understanding

1. In an `if-else` statement, when is the code block following the `else` executed?
2. What is the key difference in execution flow between a series of separate `if` statements and a single `if-else-if` statement?
3. When would a nested `if` statement be more appropriate than an `if-else-if` statement? Provide an example scenario.

Practice Problems

1. Write an `if-else` statement that checks if a `double` variable named `accountBalance` is less than 0. If it is, print "Account overdrawn." Otherwise, print "Account in good standing."
2. Convert the following nested `if` statement into a single `if` statement using a logical operator.

```
int age = 25;
boolean hasLicense = true;
if (age >= 16) {
    if (hasLicense) {
        System.out.println("You are eligible to drive.");
    }
}
```

3. A program needs to assign a shipping cost based on a package's weight. Write an `if-else-if` statement that assigns a value to a `double shippingCost` variable based on the `double weight` variable according to these rules:
   - Weight under 2 lbs: cost is $5.50
   - Weight from 2 lbs up to 10 lbs: cost is $12.00
   - Weight 10 lbs or more: cost is $18.25 # 3.3 Logical Operators

Logical operators (`&&`, `||`, `!`) are used to combine or modify Boolean expressions.

- `&&` (**AND**): Returns `true` only if both expressions are true.
- `||` (**OR**): Returns `true` if at least one expression is true.
- `!` (**NOT**): Reverses the logical state of an expression. It's a **unary operator** because it works on a single operand.

Here's a table with an example of each:

Operator	Meaning	Example	Result (if p=true,q=false)
&&	**Logical AND**	p && q	`false`
||	**Logical OR**	p || q	`true`
!	**Logical NOT**	!p	`false`

Logical operators can simplify nested `if` statements. The loan qualifier example could be rewritten as:

```
double salary = 50000;
int yearsOnJob = 3;

if (salary >= 40000 && yearsOnJob >= 2) {
    System.out.println("You qualify for the loan.");
} else {
    System.out.println("You do not qualify for the loan.");
}
```

This is useful for checking numeric ranges. For example, to test if `temp` is between 0 and 100 inclusive, you would use the `&&` operator.

```
int temp = 75;
if (temp >= 0 && temp <= 100) {
    System.out.println("Temperature is within the valid range.");
}
```

Similarly, you can combine multiple conditions, such as testing if `x` is within a range OR if `y` has a certain value.

```
int x = 550;
int y = 2000;

// Test if x is between 500 and 650 (exclusive) OR y is not equal to 1000.
if ((x > 500 && x < 650) || (y != 1000)) {
    System.out.println("The condition is true.");
}
```

Check Your Understanding

1. Name the three logical operators and describe what each one does.
2. For the `&&` (AND) operator to return `true`, what must be true about its operands? What about for the `||` (OR) operator?
3. The `!` operator is a unary operator. What does this mean, and how does it differ from operators like `&&` and `+`? ## Practice Problems
4. Write a boolean expression that would be `true` if an `int` variable `x` is between, but not including, 500 and 650.
5. What is the output of the following code snippet? Explain your reasoning.

```
boolean isMember = false;
int age = 67;
if (isMember || age > 65) {
    System.out.println("Discount applies.");
} else {
```

```
    System.out.println("No discount.");
}
```

3.4 Comparing `String` Objects

A common pitfall is using the `==` operator to compare `String` objects. Because strings are objects, the `==` operator compares their memory addresses, not their contents. While it may sometimes appear to work, it is unreliable.

To compare the actual text content of two strings, you **must use the `.equals()` method**.

```
String str1 = "Hello";
String str2 = "Hello";
String str3 = "Goodbye";

// Correct way to compare strings for equality
if (str1.equals(str2)) { // This is true
    System.out.println("str1 and str2 are equal.");
}

if (str1.equals(str3)) { // This is false
    System.out.println("str1 and str3 are equal.");
} else {
    System.out.println("str1 and str3 are NOT equal.");
}
```

If you want to compare strings while ignoring differences in capitalization, use the `.equalsIgnoreCase()` method.

```
String username1 = "Admin";
String username2 = "admin";

if (username1.equalsIgnoreCase(username2)) { // This is true
    System.out.println("Usernames are the same, ignoring case.");
}
```

Check Your Understanding

1. Why is it incorrect to use the `==` operator to compare the contents of two `String` objects? What does `==` actually compare when used with objects?
2. Describe a situation where using the `.equalsIgnoreCase()` method would be more appropriate than the `.equals()` method. ## Practice Problems
3. Given `String pass1 = "secret";` and a `Scanner` object to read user input into a `String pass2`, write an `if-else` statement that correctly

checks if the user's input matches `pass1` and prints "Access granted" or "Access denied" accordingly. # 3.5 The Conditional (Ternary) Operator

The **conditional operator**, also known as the **ternary operator**, is a compact way to write a simple `if-else` statement. It is called "ternary" because it takes three operands.

Syntax:

```
booleanExpression ? valueIfTrue : valueIfFalse;
```

Consider this `if-else` block:

```
int hours = 3;
int billableHours;

if (hours < 5) {
    billableHours = 5;
} else {
    billableHours = hours;
}
```

Using the ternary operator, this can be written in a single line:

```
int hours = 3;
int billableHours = (hours < 5) ? 5 : hours; // billableHours will be 5
```

This operator is useful for simple conditional assignments, setting default values, or conditional printing.

Check Your Understanding

1. Why is the conditional operator referred to as the "ternary" operator?
2. The ternary operator can make code more concise. Can you think of a situation where using a standard `if-else` statement would be more readable and thus a better choice? ## Practice Problems
3. Rewrite the following `if-else` statement using the conditional (ternary) operator.

```
int points = 95;
String result;
if (points > 90) {
    result = "Excellent";
} else {
    result = "Good";
}
```

3.6 The `switch` Statement

The `switch` statement provides another way to create a multi-branch decision structure. It is ideal when you need to check a single variable or expression against a series of discrete values (`case`).

Java has evolved its `switch` syntax. **Modern Java (version 14+) introduces a more concise "arrow syntax" (`->`) that eliminates the need for `break` statements and allows the `switch` to be used as an expression that returns a value**.

Modern `switch` Example

The modern syntax is cleaner and less error-prone. Here, the `switch` expression determines the value of `monthName` and assigns it directly.

```
int month = 2;
String monthName = switch (month) {
    case 1 -> "January";
    case 2 -> "February";
    case 3 -> "March";
    // ... other months
    default -> "Invalid month";
};
System.out.println(monthName); // Prints: February
```

The arrow syntax ensures that only the code for the matching case is executed, preventing "fall-through" errors common with the classic syntax. If you need to execute multiple statements for a case, you can use curly braces and the `yield` keyword to return a value.

```
int choice = 1;
String result = switch (choice) {
    case 1 -> {
        System.out.println("Performing action one...");
        yield "Completed action 1";
    }
    case 2 -> "Completed action 2";
    default -> "Invalid choice";
};
```

Classic `switch` Syntax

The classic syntax is still valid but is more verbose and requires a `break` statement at the end of each `case` block to prevent execution from "falling through" to the next case.

```
int month = 2;
String monthName;
```

```java
switch (month) {
    case 1:
        monthName = "January";
        break;
    case 2:
        monthName = "February";
        break;
    case 3:
        monthName = "March";
        break;
    // ... other months
    default:
        monthName = "Invalid month";
        break;
}
System.out.println(monthName);
```

Check Your Understanding

1. What is the purpose of the `break` statement in a classic `switch` statement? What problem does the modern arrow (`->`) syntax solve?
2. When would a `switch` statement be a better choice than a long `if-else-if` statement?
3. In modern Java, what is the purpose of the `yield` keyword within a `switch` expression?

Practice Problems

1. Write a modern `switch` statement that takes a `char` variable `grade` and prints "Excellent" for 'A', "Good" for 'B', "Average" for 'C', and "Needs Improvement" for any other grade. # 3.7 Displaying Formatted Output

Often, you need more control over how output is displayed, such as setting the number of decimal places for a floating-point number or aligning text in columns. Java provides two powerful methods for this: `System.out.printf` and `String.format`.

`System.out.printf`

The `printf` method (short for "print formatted") allows you to format and print data in one step. It takes a **format string** containing text and **format specifiers**, followed by a list of arguments that correspond to those specifiers.

Common format specifiers include:

- `%d`: for integers (`int`, `long`)

- `%f`: for floating-point numbers (`double`, `float`)
- `%s`: for strings
- `%c`: for characters

```
String name = "Alice";
int age = 30;
double salary = 75000.5;

System.out.printf("Name: %s, Age: %d, Salary: %,.2f\n", name, age, salary);
```

Output: `Name: Alice, Age: 30, Salary: 75,000.50`

In `%,.2f`:

- The comma (`,`) is a flag to add thousand separators.
- The `.2` specifies a precision of two decimal places.

You can also specify a **minimum field width** to align output in columns. By default, output is right-aligned. Use a minus sign (`-`) flag for left alignment.

```
System.out.printf("%-10s %10.2f\n", "Item A:", 12.5);
System.out.printf("%-10s %10.2f\n", "Item B:", 250.75);
```

Output:

```
Item A:        12.50
Item B:       250.75
```

`String.format()`

The `String.format()` method works exactly like `printf` but instead of printing the output to the console, it **returns the formatted string as a `String` object**. You can then store this string in a variable for later use.

```
double price = 19.99;
int quantity = 3;
double total = price * quantity;

String output = String.format("Total cost: $%.2f", total);

// You can now use the 'output' string elsewhere
System.out.println(output); // Prints: Total cost: $59.97
```

This is useful when you need to build a formatted string to be used in a GUI, saved to a file, or sent over a network.

Check Your Understanding

1. What is the key difference between `System.out.printf()` and `String.format()`?

2. In the format specifier `%,10.2f`, explain what the comma (`,`), the `10`, and the `.2` each do. ## Practice Problems

3. You need to generate a log message that will be saved to a file. The message should be "User: [username], ID: [id]". The username is a `String` and the id is an `int`. Use `String.format()` to create this message and store it in a `String` variable called `logEntry`.