Module 05 — Nested Statements and Strings

ITP 120 — Java Programming I

Northern Virginia Community College | Randy Michak

🎯 Learning Objectives

- Write and trace nested `if` statements across multiple levels
- Combine nested logic with `&&`, `||`, and `!` for cleaner decisions
- Declare, concatenate, and explain why Strings are immutable
- Call common String methods: `length()`, `charAt()`, `substring()`, `indexOf()`, `contains()`, and more
- Compare Strings correctly using `equals()` — not `==`
- Use `StringBuilder` to build strings efficiently in loops
- Process strings character by character with the `Character` class
- Apply everything in a realistic password validator and email format checker

📚 Key Terms

nested if · **decision tree** · **immutable** · **String pool** · **reference type** · **zero-based index** · **substring** · **StringBuilder** · **Character class** · **lexicographic order**

1. Nested if Statements — A Quick Recap

You saw basic `if` / `else if` / `else` chains in Module 4. A **nested if** is an `if` statement living *inside* another `if` statement. The inner block only runs when the outer condition was already true.

> 🧠 **Think of It This Way**
>
> Nesting is like a series of locked doors. You only reach door #2 if door #1 already opened. The inner `if` block only executes when the outer condition evaluated to `true`. There is no shortcut to door #2.

1.1 The Basic Shape

```
if (outerCondition) {
    // We are inside the outer if
    if (innerCondition) {
        System.out.println("Both are true");
    } else {
        System.out.println("Only outer is true");
    }
} else {
    // Outer is false -- inner block never runs
    System.out.println("Outer is false");
}
```

1.2 Multi-Level Nesting — Number Classifier

```
int num = 42;

if (num != 0) {
    if (num > 0) {
        if (num % 2 == 0) {
            System.out.println("Positive and even");
        } else {
            System.out.println("Positive and odd");
        }
    } else {
        System.out.println("Negative number");
    }
} else {
    System.out.println("Zero");
}
```

```
Output: Positive and even
```

1.3 Nesting vs. Logical Operators

Sometimes a nested `if` can be collapsed into one compound condition. Compare:

```
// Nested -- more verbose, same result
if (age >= 18) {
    if (hasID) {
        System.out.println("Allowed entry");
    }
}

// Cleaner with &&
if (age >= 18 && hasID) {
    System.out.println("Allowed entry");
}
```

Use nesting when the inner logic is complex enough that flattening it would create an unreadable condition. Use `&&` and `||` when you can express the whole check on one clean line.

> 💡 **Indentation Is Not Optional**
>
> Java ignores whitespace, but your teammates — and your future self — do not. Use consistent indentation (2 or 4 spaces per level). If your code is stair-stepping off the right edge of the screen, it is time to refactor.

2. Complex Decision Logic

Real programs make layered decisions. A grade calculator needs a score. A shipping calculator needs the weight *and* the destination zone. This is where nested ifs really pay off.

2.1 Grade Calculator

```
import java.util.Scanner;

public class GradeCalc {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter score (0-100): ");
        int score = input.nextInt();

        String grade;
        if (score >= 90) {
            grade = "A";
        } else if (score >= 80) {
            grade = "B";
        } else if (score >= 70) {
            grade = "C";
        } else if (score >= 60) {
            grade = "D";
        } else {
            grade = "F";
        }

        System.out.println("Grade: " + grade);
    }
}
```

```
Enter score (0-100): 85 Grade: B
```

As soon as one branch matches, Java skips the rest. That is the *else-if ladder* — the cleanest approach for mutually exclusive numeric ranges.

2.2 Shipping Cost by Weight and Zone

Here the cost depends on two independent factors, so two-level nesting makes sense:

```
String zone = "East";
double weightLbs = 8.5;
double cost;

if (zone.equals("East")) {
    if (weightLbs <= 5) {
        cost = 5.99;
    } else if (weightLbs <= 20) {
        cost = 9.99;
    } else {
        cost = 19.99;
    }
} else if (zone.equals("West")) {
    if (weightLbs <= 5) {
        cost = 7.99;
    } else if (weightLbs <= 20) {
        cost = 12.99;
    } else {
        cost = 24.99;
    }
} else {
    cost = 39.99;  // International flat rate
}

System.out.printf("Shipping cost: $%.2f%n", cost);
```

```
Shipping cost: $9.99
```

⚠️ The Dangling Else Problem

Java matches each `else` with the *nearest* unmatched `if`. Always use curly braces `{ }` around every block, even single-line bodies. Skipping braces is the number-one source of hard-to-find logic bugs in nested code.

🔧 Try It — Tax Bracket Calculator

Write a program that reads annual income and prints the marginal tax rate:

- $0 – $11,000 → 10%
- $11,001 – $44,725 → 12%
- $44,726 – $95,375 → 22%
- Above $95,375 → 24%

Then add a nested check: if the rate is 22%+ *and* `hasDependents` is true, print "You may qualify for additional deductions."

3. String Basics

You have been using Strings since Module 1. Now it is time to understand what they actually are under the hood — because that knowledge explains some surprising behavior you are about to encounter.

3.1 String Is a Class, Not a Primitive

Java has eight primitive types: `int`, `double`, `boolean`, `char`, and four others. `String` is **not** on that list. It is a full Java class, so String variables hold *references* to objects in memory — not the data itself. That distinction matters a lot when we get to comparisons.

```
// Most common: string literal
String name = "Alice";

// Explicit object creation (rare, usually unnecessary)
String greeting = new String("Hello");

// Empty string -- NOT null, has length 0
String empty = "";

// Concatenation with +
String full = greeting + ", " + name + "!";
System.out.println(full);
```

```
Hello, Alice!
```

3.2 Strings Are Immutable

Once a String object is created, its content **cannot change**. When you write `s = s + "!"`, Java creates a brand-new String with the added character and points your variable at it. The old object floats in memory until the garbage collector removes it.

```
String s = "Hello";
s = s + " World";   // New object created; s now points to it
System.out.println(s);
```

```
Hello World
```

For most code, immutability does not matter. But inside a loop that builds a string piece by piece, creating hundreds of throwaway objects tanks performance. That is why `StringBuilder` exists — covered in Section 7.

> 🧠 **Think of It This Way**
>
> A String is like a printed photograph. You can look at it, copy it, and label it differently — but you cannot change the image on the print. To "edit" it, you print a new photo. `StringBuilder` is the digital negative: modify it in place, then print the final version.

4. String Methods

The `String` class has over 60 methods. Here are the ones you need right now:

Method	Returns	What It Does
`length()`	`int`	Number of characters in the string
`charAt(i)`	`char`	Character at zero-based index *i*
`toUpperCase()`	`String`	All letters converted to uppercase

Method	Returns	What It Does
`toLowerCase()`	`String`	All letters converted to lowercase
`trim()`	`String`	Removes leading and trailing whitespace
`substring(begin)`	`String`	Characters from *begin* through the end
`substring(begin, end)`	`String`	From *begin* up to but NOT including *end*
`indexOf(str)`	`int`	First position of *str*; -1 if not found
`contains(str)`	`boolean`	`true` if *str* appears anywhere
`startsWith(str)`	`boolean`	`true` if string begins with *str*
`endsWith(str)`	`boolean`	`true` if string ends with *str*
`replace(old, new)`	`String`	Replaces every occurrence of *old* with *new*
`isEmpty()`	`boolean`	`true` if the length is 0

4.1 length() and charAt()

```
String word = "Java";
System.out.println(word.length());   // 4
System.out.println(word.charAt(0));  // J
System.out.println(word.charAt(3));  // a
// word.charAt(4) throws StringIndexOutOfBoundsException!
```

Indexes always start at **0**. The last valid index is always `length() - 1`. That rule applies everywhere in Java.

4.2 toUpperCase(), toLowerCase(), trim()

```
String messy = "  Hello, World!  ";
System.out.println(messy.toUpperCase());         // "  HELLO, WORLD!  "
System.out.println(messy.toLowerCase());         // "  hello, world!  "
System.out.println(messy.trim());                // "Hello, World!"
System.out.println(messy.trim().toLowerCase());  // "hello, world!"
```

4.3 substring()

```
String url = "https://nova.edu";
System.out.println(url.substring(8));       // "nova.edu"
System.out.println(url.substring(0, 5));    // "https"
System.out.println(url.substring(8, 12));   // "nova"
```

> 💡 **Off-by-One Is Real**
>
> `substring(0, 5)` gives you positions 0, 1, 2, 3, 4 — five characters. The end index is *exclusive*. Think: "start here, stop *before* this index." Same convention in Python, JavaScript, and most other languages.

4.4 indexOf(), contains(), startsWith(), endsWith()

```
String email = "student@nova.edu";
System.out.println(email.indexOf("@"));       // 7
System.out.println(email.indexOf("."));       // 12  (first dot)
System.out.println(email.indexOf("xyz"));     // -1  (not found)
System.out.println(email.contains("nova"));   // true
System.out.println(email.startsWith("stu"));  // true
System.out.println(email.endsWith(".edu"));   // true
System.out.println(email.endsWith(".com"));   // false
```

4.5 replace() and isEmpty()

```
String s = "I love cats. Cats are great.";
System.out.println(s.replace("cats", "dogs"));
// "I love dogs. Cats are great."  (case-sensitive -- "Cats" unchanged)

String blank  = "";
String spaces = "   ";
System.out.println(blank.isEmpty());          // true
System.out.println(spaces.isEmpty());         // false
System.out.println(spaces.trim().isEmpty());  // true
```

🔧 Try It — String Method Practice

Given `String s = " Northern Virginia ";`, write code that:

1. Removes whitespace and prints the trimmed result
2. Prints the length of the trimmed string
3. Prints the character at index 9 of the trimmed string
4. Prints the first 8 characters of the trimmed string
5. Prints the trimmed string in all uppercase

5. String Comparison

This is where a lot of beginners make a mistake that costs hours of debugging. Pay close attention.

5.1 Why == Fails on Strings

`==` on objects checks whether two variables point to the *same object in memory* — not whether they hold the same text. For string literals, Java sometimes reuses the same object (the string pool), so `==` might appear to work. But it is unreliable — when it fails silently, you get a bug that is nearly impossible to track down.

```
String a = "hello";
String b = "hello";
String c = new String("hello");

System.out.println(a == b);          // true  (same pool object -- gets lucky)
System.out.println(a == c);          // false (c is a separate heap object)
System.out.println(a.equals(c));     // true  (same characters)
System.out.println(a.equals(b));     // true
```

> ⚠️ **The Golden Rule of String Comparison**
>
> Always use `.equals()` to compare String content. Never use `==` on Strings. Your instructor will dock points on every assignment where you use `==` to compare strings. Consider yourself warned.

5.2 equals() and equalsIgnoreCase()

```
String input = "YES";

if (input.equals("yes")) {
    System.out.println("Exact match");           // does NOT print
}
if (input.equalsIgnoreCase("yes")) {
    System.out.println("Case-insensitive match"); // prints this
}
```

Use `equalsIgnoreCase()` any time user input is involved. Users type "YES", "Yes", and "yes" interchangeably — always meet them where they are.

5.3 compareTo() — Alphabetical Order

`compareTo()` determines lexicographic order. It returns 0 if equal, negative if the calling string comes first alphabetically, positive if it comes after.

```
String s1 = "apple";
String s2 = "banana";

System.out.println(s1.compareTo(s2));      // negative (apple < banana)
System.out.println(s2.compareTo(s1));      // positive (banana > apple)
System.out.println(s1.compareTo("apple")); // 0

if (s1.compareTo(s2) < 0) {
    System.out.println(s1 + " comes before " + s2);
}
```

```
apple comes before banana
```

> 💡 **The String Pool (Brief Version)**
>
> When you write `String a = "hello"`, Java stores "hello" in a special cache called the *string pool*. The next time you write `String b = "hello"`, Java reuses that same object — that is why `a == b` is sometimes `true`. But `new String("hello")` always creates a fresh object outside the pool. Takeaway: never rely on `==`. Always use `equals()`.

6. String Parsing and Searching

Combining `indexOf()` and `substring()` lets you extract meaningful pieces from a larger string. This is the backbone of most real string-processing tasks.

6.1 Extracting Domain from an Email

```
String email = "student@nova.edu";

int atPos   = email.indexOf("@");           // 7
String domain = email.substring(atPos + 1); // "nova.edu"

int dotPos  = domain.indexOf(".");          // 4
String tld  = domain.substring(dotPos + 1); // "edu"

System.out.println("Domain : " + domain);
System.out.println("TLD    : " + tld);
```

```
Domain : nova.edu TLD : edu
```

6.2 Counting a Character

```
String sentence = "Hello from NOVA";
int vowelCount = 0;

for (int i = 0; i < sentence.length(); i++) {
    char c = sentence.charAt(i);
    String lower = String.valueOf(c).toLowerCase();
    if (lower.equals("a") || lower.equals("e") || lower.equals("i")
            || lower.equals("o") || lower.equals("u")) {
        vowelCount++;
    }
}
System.out.println("Vowels: " + vowelCount);
```

```
Vowels: 5
```

6.3 Finding Multiple Occurrences with indexOf(start)

`indexOf(str, fromIndex)` lets you search starting at a specific position — useful for finding every occurrence of a character:

```
String text = "banana";
int pos = 0;
int count = 0;

while ((pos = text.indexOf("a", pos)) != -1) {
    System.out.println("Found 'a' at index " + pos);
    count++;
    pos++;  // move past this occurrence
}
System.out.println("Total 'a' count: " + count);
```

```
Found 'a' at index 1 Found 'a' at index 3 Found 'a' at index 5 Total 'a' count: 3
```

🔧 Try It — Parse User Input

Ask the user to enter a full name in "Firstname Lastname" format. Use `indexOf(" ")` and `substring()` to split it into first and last name. Print them on separate lines. Bonus: handle the case where the user enters no space.

7. StringBuilder

You know that Strings are immutable. That means every time you use `+` to add something to a String, Java creates a new object. For a handful of concatenations, that is fine. For a loop that runs a thousand times, it is wasteful.

Enter `StringBuilder` — a mutable sequence of characters you can modify in place. When you are done building, call `toString()` to get a regular String.

7.1 Why StringBuilder Matters

```
// Slow approach -- creates 1000 temporary String objects
String result = "";
for (int i = 0; i < 1000; i++) {
    result += i + " ";
}

// Fast approach -- modifies one object in place
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 1000; i++) {
    sb.append(i);
    sb.append(" ");
}
String result2 = sb.toString();
```

For small strings, you will never notice the difference. For large strings built in loops — like generating HTML or processing log files — `StringBuilder` can be orders of magnitude faster.

7.2 Key StringBuilder Methods

Method	What It Does
`append(value)`	Adds *value* at the end (accepts any type)
`insert(i, value)`	Inserts *value* at index *i*
`delete(start, end)`	Removes characters from *start* to *end* (exclusive)
`replace(start, end, str)`	Replaces characters from *start* to *end* with *str*
`reverse()`	Reverses the entire sequence
`length()`	Number of characters currently in the builder
`toString()`	Converts to an immutable String

7.3 StringBuilder Examples

```
StringBuilder sb = new StringBuilder("Hello");

sb.append(", World");
System.out.println(sb);          // "Hello, World"

sb.insert(5, " there");
System.out.println(sb);          // "Hello there, World"

sb.delete(5, 11);
System.out.println(sb);          // "Hello, World"

sb.reverse();
System.out.println(sb);          // "dlroW ,olleH"

// Convert back to String when done
String finished = sb.reverse().toString();
System.out.println(finished);    // "Hello, World"
```

7.4 Building a Table of Squares

```
StringBuilder table = new StringBuilder();
table.append("n\tn^2\n");
table.append("---\t---\n");

for (int n = 1; n <= 5; n++) {
    table.append(n);
    table.append("\t");
    table.append(n * n);
    table.append("\n");
}

System.out.print(table.toString());
```

```
n n^2 --- --- 1 1 2 4 3 9 4 16 5 25
```

> **💡 Use StringBuilder in Loops**
>
> Any time you are building a String inside a loop, use `StringBuilder`. Outside loops, `+` concatenation is fine — the Java compiler often optimizes it to a `StringBuilder` behind the scenes anyway.

8. The Character Class

The `Character` class wraps a single `char` and provides static utility methods for testing and converting characters. You call these as `Character.methodName(c)` where `c` is a `char`.

Method	Returns	Example
`Character.isLetter(c)`	`boolean`	`isLetter('A')` → true
`Character.isDigit(c)`	`boolean`	`isDigit('7')` → true
`Character.isUpperCase(c)`	`boolean`	`isUpperCase('Z')` → true
`Character.isLowerCase(c)`	`boolean`	`isLowerCase('z')` → true
`Character.isWhitespace(c)`	`boolean`	`isWhitespace(' ')` → true
`Character.isLetterOrDigit(c)`	`boolean`	`isLetterOrDigit('@')` → false
`Character.toUpperCase(c)`	`char`	`toUpperCase('a')` → 'A'
`Character.toLowerCase(c)`	`char`	`toLowerCase('A')` → 'a'

8.1 Processing a String Character by Character

```
String input = "Hello123!";
int letters = 0, digits = 0, symbols = 0;

for (int i = 0; i < input.length(); i++) {
    char c = input.charAt(i);
    if (Character.isLetter(c)) {
        letters++;
    } else if (Character.isDigit(c)) {
        digits++;
    } else {
        symbols++;
    }
}

System.out.println("Letters: " + letters);   // 5
System.out.println("Digits : " + digits);    // 3
System.out.println("Symbols: " + symbols);   // 1
```

8.2 Reversing a String and Converting Case

```
String word = "Java";
StringBuilder reversed = new StringBuilder();

for (int i = word.length() - 1; i >= 0; i--) {
    char c = word.charAt(i);
    if (Character.isUpperCase(c)) {
        reversed.append(Character.toLowerCase(c));
    } else {
        reversed.append(Character.toUpperCase(c));
    }
}

System.out.println(reversed.toString());   // "AVAj"
```

> 💡 **char vs. String — Know the Difference**
>
> A `char` holds a single character and uses *single quotes*: `'A'`. A `String` holds zero or more characters and uses *double quotes*: `"A"`. They look similar but they are completely different types. You cannot call String methods on a `char` — use `Character` class methods instead.

> 🔧 **Try It — Character Analysis**
>
> Write a program that reads a word from the user and prints:
>
> - The number of uppercase letters
> - The number of lowercase letters
> - The number of digits
> - Whether the word starts with an uppercase letter

9. Practical Examples

This section puts everything together. All three examples pull from nested ifs, String methods, StringBuilder, and the Character class simultaneously.

9.1 Password Validator

A real password check needs to verify multiple requirements independently. Nested ifs are perfect for building up detailed feedback.

```
import java.util.Scanner;

public class PasswordValidator {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter password: ");
        String password = sc.nextLine();

        boolean valid = true;
        StringBuilder feedback = new StringBuilder();

        // Check length
        if (password.length() < 8) {
            valid = false;
            feedback.append("- Must be at least 8 characters\n");
        }

        // Scan for uppercase and digit
        boolean hasUpper = false;
        boolean hasDigit = false;

        for (int i = 0; i < password.length(); i++) {
            char c = password.charAt(i);
            if (Character.isUpperCase(c)) {
                hasUpper = true;
            }
            if (Character.isDigit(c)) {
                hasDigit = true;
            }
        }

        if (!hasUpper) {
            valid = false;
            feedback.append("- Must contain at least one uppercase letter\n");
        }
        if (!hasDigit) {
            valid = false;
            feedback.append("- Must contain at least one digit\n");
        }

        // Report result
        if (valid) {
```

```
            System.out.println("Password accepted!");
        } else {
            System.out.println("Password rejected. Issues:\n" + feedback);
        }
    }
}
```

```
Enter password: hello Password rejected. Issues: - Must be at least 8
characters - Must contain at least one uppercase letter - Must contain at least
one digit
```

9.2 Email Format Checker

A simple check for basic email validity using String methods:

```
public static boolean isValidEmail(String email) {
    // Must have exactly one @
    int atPos = email.indexOf("@");
    if (atPos <= 0) {
        return false;  // No @ or starts with @
    }
    if (email.indexOf("@", atPos + 1) != -1) {
        return false;  // More than one @
    }

    // Must have a dot after the @
    String domain = email.substring(atPos + 1);
    if (!domain.contains(".")) {
        return false;
    }

    // Domain can't start or end with a dot
    if (domain.startsWith(".") || domain.endsWith(".")) {
        return false;
    }

    return true;
}

// Test it:
System.out.println(isValidEmail("student@nova.edu"));  // true
System.out.println(isValidEmail("bademail.com"));      // false
System.out.println(isValidEmail("also@bad@email.com")); // false
```

```
true false false
```

9.3 Input Cleanup Utility

User input is almost always messy. This utility normalizes a name entered by the user:

```
import java.util.Scanner;

public class CleanInput {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter your full name: ");
        String raw = sc.nextLine();

        // Clean it up
        String cleaned = raw.trim();

        // Find the space separating first and last
        int spacePos = cleaned.indexOf(" ");

        if (spacePos == -1) {
            // Only one word provided
            String capitalized = Character.toUpperCase(cleaned.charAt(0))
                               + cleaned.substring(1).toLowerCase();
            System.out.println("Hello, " + capitalized + "!");
        } else {
            String first = cleaned.substring(0, spacePos);
            String last  = cleaned.substring(spacePos + 1);

            // Capitalize first letter of each name
            first = Character.toUpperCase(first.charAt(0))
                  + first.substring(1).toLowerCase();
            last  = Character.toUpperCase(last.charAt(0))
                  + last.substring(1).toLowerCase();

            System.out.println("Hello, " + first + " " + last + "!");
        }
    }
}
```

```
Enter your full name: alice SMITH Hello, Alice Smith!
```

> 💡 **Always Trim User Input**
>
> Users accidentally add leading or trailing spaces all the time. Make it a habit to call `.trim()` on anything that comes from `Scanner` or a text field before doing any comparisons or processing. It will save you from hours of "why doesn't this match?!" bugs.

Vocabulary Review

Term	Definition
nested if	An `if` statement placed inside another `if` or `else` block; the inner block only executes when the outer condition is true.
decision tree	A branching structure of conditions that leads to different outcomes; implemented in Java with nested or chained `if` statements.
immutable	Cannot be changed after creation. Java Strings are immutable — any "modification" creates a new object.
String pool	A special area in memory where Java caches String literal objects to avoid creating duplicate instances.
reference type	A variable that stores a memory address (reference) to an object, rather than the value directly. `String` is a reference type.
zero-based index	Counting that starts at 0 instead of 1. The first character of a String is at index 0.
substring	A portion of a String extracted using the `substring()` method.
StringBuilder	A mutable sequence of characters used to efficiently build Strings, especially in loops.
Character class	A Java wrapper class that provides static methods for testing and converting individual `char` values.

Term	Definition
lexicographic order	Ordering based on Unicode character values — essentially dictionary order; returned by `compareTo()`.

📋 Module 05 Summary

- **Nested if:** Inner `if` blocks only run when the outer condition is already true. Use curly braces always. Indent consistently.
- **Decision trees:** The else-if ladder handles mutually exclusive ranges. True nesting handles multi-dimensional conditions (zone + weight).
- **String is a class:** Not a primitive. Variables hold references, not values directly.
- **Immutable:** Strings cannot change. Every "modification" creates a new object.
- **String methods:** `length()`, `charAt()`, `substring()`, `indexOf()`, `contains()`, `replace()`, `trim()`, `toUpperCase()`, `toLowerCase()`, `isEmpty()`.
- **String comparison:** Always `.equals()`, never `==`. Use `equalsIgnoreCase()` for user input. Use `compareTo()` for alphabetical ordering.
- **StringBuilder:** Mutable string builder. Use in loops. Call `toString()` when done.
- **Character class:** Static methods for testing and converting individual characters: `isLetter()`, `isDigit()`, `isUpperCase()`, `toUpperCase()`, etc.

Module 05 Quiz

Choose the best answer for each question. Record your answers on a separate sheet.

1. What is the output of the following code?

```
int x = 10;
if (x > 5) {
    if (x > 8) {
        System.out.println("A");
    } else {
        System.out.println("B");
    }
} else {
    System.out.println("C");
}
```

A) B

B) C

C) A

D) A and B

2. Which statement about Strings in Java is true?

A) String is a primitive type like `int` or `double`

B) String variables can be compared with `==` safely

C) String objects are immutable — their content cannot be changed after creation

D) Calling `toUpperCase()` modifies the original String object

3. What does this code print?

```
String s = "Programming";
System.out.println(s.substring(0, 4));
```

A) "Prog"

B) "Progr"

C) "rogr"

D) "ammi"

4. A student writes: `if (name == "Alice")` to check if a String matches. What is the problem?

A) `==` is not a valid Java operator

B) `==` compares object references, not String content — it may return `false` even when the text is the same

C) String literals like `"Alice"` cannot be used in conditions

D) There is no problem; `==` always works correctly on Strings

5. What is the value of `pos` after: `int pos = "hello world".indexOf("world");`

A) 5

B) 6

C) -1

D) 4

6. Which of the following correctly creates a `StringBuilder`, appends "Hello" and " World", and converts it to a String?

A) `String s = new StringBuilder("Hello").add(" World").build();`

B) `StringBuilder sb = new StringBuilder(); sb.append("Hello"); sb.append(" World"); String s = sb.toString();`

C) `StringBuilder sb = "Hello" + " World"; String s = sb;`

D) `String s = StringBuilder("Hello", " World");`

7. What does `Character.isDigit('9')` return?

A) `"9"`

B) `9`

C) `false`

D) `true`

8. What is printed by the following code?

```
String s = "  Java  ";
System.out.println(s.trim().length());
```

A) 8

B) 6

C) 4

D) 7

9. When should you use `StringBuilder` instead of String concatenation with `+`?

A) Whenever you are working with numbers

B) Only when the string is longer than 100 characters

C) When building a string incrementally inside a loop, to avoid creating many temporary objects

D) Never — `+` is always preferred because it is simpler

10. What does `"apple".compareTo("banana")` return?

A) `true`

B) A negative integer, because "apple" comes before "banana" alphabetically

C) A positive integer, because "apple" comes before "banana" alphabetically

D) `0`, because both are valid English words

✅ Answer Key

Question	Answer	Explanation
1	**C**	Multi-level nested if statements evaluate conditions from outside in.
2	**C**	Strings in Java are immutable — once created, they cannot be changed.
3	**A**	`substring(0, 4)` returns characters at indices 0, 1, 2, 3 (end index is exclusive).
4	**B**	`==` compares object references, not String content. Use `.equals()` instead.
5	**B**	`indexOf()` returns the zero-based position of the first occurrence.
6	**B**	StringBuilder uses `append()` to build strings and `toString()` to convert.
7	**D**	`Character.isDigit()` returns true if the character is a digit.
8	**C**	`trim()` removes leading and trailing whitespace, then `length()` counts remaining characters.
9	**C**	String concatenation with `+` creates a new String object each time.
10	**B**	`charAt()` returns the character at a specified index position.