Java Programming Fundamentals Study Guide

This guide covers core concepts in Java programming, including data types, operators, control flow (conditional statements and loops), character and string manipulation, and the basics of methods and variable scope.

I. Data Types and Operators

A. Numeric Data Types and Their Limits

- **boolean:** Represents true or false values.
- **byte:** Maximum value 127.
- **short:** Maximum value 32,767.
- **int:** Maximum value 2,147,483,647.
- **long:** Maximum value > 9 x 10^18.
- **float:** Single-precision floating-point numbers.
- **double:** Double-precision floating-point numbers (default for floating-point literals).

B. Numeric Operators

- **+ (Addition):** Adds two operands.
- **- (Subtraction):** Subtracts the second operand from the first.
- *** (Multiplication):** Multiplies two operands.
- **/ (Division):** Divides the first operand by the second.
- **Integer Division:** If both operands are integers, the result is an integer, truncating any decimal part (e.g., 5 / 2 yields 2).
- **Floating-point Division:** If at least one operand is a floating-point number, the result is a floating-point number (e.g., 5.0 / 2 yields 2.5).
- **% (Remainder/Modulo):** Returns the remainder of a division (e.g., 20 % 3 yields 2, 5 % 2 yields 1).

C. Assignment Operators

- **= (Assignment):** Assigns a value to a variable. (e.g., x = 1;).
- **Augmented Assignment Operators:** Combine an arithmetic operation with assignment.

- += (Addition assignment): i += 8 is equivalent to i = i + 8.
- -= (Subtraction assignment): i -= 8 is equivalent to i = i - 8.
- *= (Multiplication assignment): i *= 8 is equivalent to i = i * 8.
- /= (Division assignment): i /= 8 is equivalent to i = i / 8.
- %= (Remainder assignment): i %= 8 is equivalent to i = i % 8.

D. Increment and Decrement Operators

- **++ (Increment):** Increases the value of a variable by 1.
- **Prefix increment (++var):** Increments the variable *before* using its value in the statement.
- **Postfix increment (var++):** Uses the original value of the variable in the statement *before* incrementing it.
- **-- (Decrement):** Decreases the value of a variable by 1.
- **Prefix decrement (--var):** Decrements the variable *before* using its value in the statement.
- **Postfix decrement (var--):** Uses the original value of the variable in the statement *before* decrementing it.

II. Control Flow Statements

A. Conditional Statements (if, if-else, if-else-if)

- **if (condition) { statements; }**: Executes statements if the condition is true.
- **if (condition) { statements; } else { statements; }**: Executes one block if the condition is true, and another block if it's false.
- **if (condition1) { statements; } else if (condition2) { statements; } else { statements; }**: Evaluates conditions sequentially, executing the block for the first true condition. The final else block executes if no preceding condition is true.
- **Block Statements:** Multiple statements can be grouped using curly braces {}. If a conditional statement has only one statement in its body, braces can be omitted, but this is prone to errors and generally discouraged for readability.
- **else Mapping Rule:** An else clause is always mapped to the last unmatched if statement (the most nested if). Indentation does not affect this rule.

- **Semicolon after if statement:** A semicolon immediately after an if condition (e.g., if (condition);) creates an empty if body, leading to logical errors. The subsequent code will execute regardless of the condition.
- **Boolean Expressions:** Conditions evaluate to a boolean (true/false).
- **Comparison Operators:** >, <, >=, <=, == (equality), != (inequality).
- **Logical Operators:**&& (Logical AND): True if both operands are true.
- || (Logical OR): True if at least one operand is true.
- ! (Logical NOT/Negation): Reverses the boolean value of an operand (unary operator).
- **Precedence:** Mathematical operators are evaluated before relational operators, which are evaluated before logical operators.
- **Conditional Operator (Ternary Operator):** boolean-expression ? expression1 : expression2. If boolean-expression is true, expression1 is returned; otherwise, expression2 is returned.

B. Loop Statements (while, do-while, for)

- **Purpose:** Allow code to be executed repeatedly (repetition statements).
- **Loop Body:** The part of the code executed during each repetition.
- **Conditional Loops (while, do-while):** Rely on a boolean expression to control execution; the number of iterations is not known beforehand (non-deterministic/undeterministic).
- **Counting Loops (for):** Use a counter; the number of iterations is typically known beforehand (deterministic).

1. while Loop

- **Syntax:** while (loop-continuation-condition) { statements; }
- **Execution:** The loop-continuation-condition is evaluated *before* each iteration. If true, the loop body executes. If false, the loop terminates. The loop body may not execute at all if the condition is initially false.

2. do-while Loop

- **Syntax:** do { statements; } while (loop-continuation-condition);

- **Execution:** The loop body executes *at least once*. After the first execution, the loop-continuation-condition is evaluated. If true, the loop continues. If false, the loop terminates.

3. for Loop

- **Syntax:** for (initialization; condition; adjustment) { statements; }
- **Components:initialization:** Executed once at the beginning of the loop (e.g., int i = 0;).
- **condition:** Evaluated before each iteration. If true, the loop body executes. If false, the loop terminates.
- **adjustment (or action-after-each-iteration):** Executed after each iteration of the loop body (e.g., i++).
- **Infinite Loops:** Can occur if the condition is always true or if the adjustment does not change the loop control variable in a way that eventually makes the condition false. Omitting the condition in a for loop implicitly makes it true (e.g., for (;;)).
- **Semicolon after for clause:** A semicolon immediately after the for clause (e.g., for (int i = 0; i < 10; i++);) creates an empty loop body, which is a common mistake and leads to logical errors.

4. Nested Loops

- A loop can be placed inside another loop.
- If an outer loop runs n times and an inner loop runs m times, the statements inside the inner loop will execute n * m times.
- The inner loop completes all its iterations for each single iteration of the outer loop.
- Commonly used for tasks like generating patterns (e.g., stars, numbers) or processing multi-dimensional data (e.g., arrays, palindrome checks).

5. break and continue Keywords

- **break:** Terminates the innermost loop (or switch statement) and transfers control to the statement immediately following the loop.
- **continue:** Skips the rest of the current iteration of the innermost loop and proceeds to the next iteration (re-evaluates the loop condition).

III. Character and String Manipulation

A. char Data Type

- **Comparison:** char values can be compared using relational operators (e.g., ch >= 'A' && ch <= 'Z').
- **ASCII/Unicode Representation:** Characters are represented by numerical Unicode values. This allows arithmetic operations on characters (e.g., ch - 'A' + 10 to convert a hex digit character to an integer value).
- **Casting:** Explicitly convert integers to characters using (char).

B. Character Class (in java.lang package, no import needed)

- Provides static methods for character testing and conversion.
- **isDigit(ch):** Returns true if ch is a digit ('0'-'9').
- **isLetter(ch):** Returns true if ch is a letter ('a'-'z' or 'A'-'Z').
- **isLetterOrDigit(ch):** Returns true if ch is a letter or a digit.
- **isLowerCase(ch):** Returns true if ch is a lowercase letter.
- **isUpperCase(ch):** Returns true if ch is an uppercase letter.
- **toLowerCase(ch):** Converts ch to its lowercase equivalent.
- **toUpperCase(ch):** Converts ch to its uppercase equivalent.
- These conversion methods are useful for standardizing user input (e.g., converting all input to uppercase or lowercase to simplify comparison logic).

C. String Class

- Strings are objects.
- Can be declared directly using double quotes (e.g., String s = "Hello";) or using the new operator (e.g., String s = new String("Hello");).
- **Key Methods:length():** Returns the number of characters in the string.
- **charAt(index):** Returns the character at the specified index (0-based).
- **indexOf(ch) / indexOf(str):** Returns the index of the first occurrence of ch or str.
- **lastIndexOf(ch) / lastIndexOf(str):** Returns the index of the last occurrence of ch or str.

- **equals(s1):** Compares two strings for equality (case-sensitive). Returns true if they are identical.
- **equalsIgnoreCase(s1):** Compares two strings for equality, ignoring case.
- **startsWith(s1):** Returns true if the string starts with s1.
- **endsWith(s1):** Returns true if the string ends with s1.
- **substring(beginIndex):** Returns a substring from beginIndex to the end of the string.
- **substring(beginIndex, endIndex):** Returns a substring from beginIndex up to, but not including, endIndex.
- **toUpperCase():** Converts all characters in the string to uppercase.
- **toLowerCase():** Converts all characters in the string to lowercase.
- **trim():** Returns a new string with leading and trailing whitespace removed.
- **replace(oldChar, newChar):** Returns a new string with all occurrences of oldChar replaced by newChar.
- **compareTo(s1):** Compares two strings lexicographically. Returns 0 if equal, a positive value if the calling string is "greater," and a negative value if it's "less." The value represents the Unicode difference at the first differing character.
- **compareToIgnoreCase(s1):** Same as compareTo, but ignores case.
- **Reading Strings from User Input (using Scanner):next():** Reads input up to the first whitespace character (space, tab, newline).
- **nextLine():** Reads the entire line of input, including spaces, until the Enter key is pressed.
- **Reading a Character from User Input:** Since Scanner doesn't have nextChar(), you typically read a string using nextLine() or next() and then extract the first character using charAt(0).
- **String to Number Conversion:Integer.parseInt(string):** Converts a string of digits to an int.
- **Double.parseDouble(string):** Converts a string representing a double to a double.
- **Number to String Conversion:** Concatenate the number with an empty string (e.g., "" + number).

IV. Math Class (in java.lang package, no import needed)

- Provides mathematical functions.
- **Math.pow(a, b):** Returns a raised to the power of b (a^b).
- **Math.sqrt(value):** Returns the square root of value. Can result in NaN (Not a Number) if the input is negative.
- **Trigonometric Functions:** Math.sin(), Math.cos(), Math.tan(), Math.asin(), Math.acos(), Math.atan().
- **Exponential and Logarithmic Functions:** Math.exp(), Math.log(), Math.log10().
- **Rounding Functions:Math.ceil(value):** Rounds value up to the nearest integer (returns a double).
- **Math.floor(value):** Rounds value down to the nearest integer (returns a double).
- **Math.round(value):** Rounds to the nearest integer. For X.5, it rounds to the nearest *even* integer in Java (e.g., Math.round(2.5) is 2, Math.round(3.5) is 4). Returns a long for double input, int for float input.
- **Math.max(value1, value2):** Returns the greater of two values.
- **Math.min(value1, value2):** Returns the lesser of two values.
- **Math.abs(value):** Returns the absolute (positive) value of an argument.
- **Math.random():** Returns a random double value between 0.0 (inclusive) and 1.0 (exclusive).
- **Generating random numbers in a range:** To generate a random integer between a (inclusive) and b (inclusive): (int)(a + Math.random() * (b - a + 1))

V. Methods and Variable Scope

A. Methods

- Reusable blocks of code that perform a specific task.
- Defining methods helps organize code and promotes reusability.
- **public static int gcd(int n1, int n2)**: Example of a method signature.
- public static: Access modifiers.
- int: Return type.

- gcd: Method name.
- (int n1, int n2): Parameters.
- **Recursive Helper Method:** A common design technique in recursive programming where a second method is defined that receives additional parameters to help with the recursion, typically for an "ever-shrinking substring" or similar problem.

B. Scope of Local Variables

- **Scope (or Lifespan):** The region of a program where a variable can be accessed.
- **Local Variable:** A variable declared inside a block (e.g., a method, a loop, an if statement).
- **Rule:** A local variable's scope starts from its declaration point and continues to the end of the block that contains it. It must be declared before use.
- **Naming Rules:**You **can** declare a local variable with the same name multiple times in *different, non-nesting* blocks within a method. This is because their lifespans do not overlap.
- You **cannot** declare a local variable twice in *nested* blocks. This creates ambiguity for the compiler as the scopes overlap.
- Variables declared in a for loop's initialization section have their scope limited to the entire for loop block (including the loop body). They cannot be accessed outside the loop.

Quiz: Short Answer Questions

Answer each question in 2-3 sentences.

1. Explain the difference between integer division and floating-point division using an example.
2. Describe the purpose of the modulo operator (%) and provide an example of its use.
3. How does the ++ operator behave differently in its prefix form (++var) versus its postfix form (var++)?
4. What is the significance of the "last unmatched if" rule when dealing with if-else statements, especially concerning indentation?
5. Explain why placing a semicolon immediately after an if statement's condition (if (condition);) is a logical error and what effect it has.

6. Differentiate between a conditional loop (like while) and a counting loop (like for) in terms of when their number of iterations is determined.
7. Under what conditions can a for loop lead to an infinite loop? Name two specific scenarios.
8. When would you use Math.ceil() versus Math.floor()? Provide an example of how Java's Math.round() handles X.5 values.
9. Describe how String.equals() differs from String.equalsIgnoreCase() and when each might be preferred.
10. Explain the rule regarding variable declaration in nested blocks and why it is enforced in Java.

Answer Key for Quiz

1. **Integer division** occurs when both operands of the division operator (/) are integers, resulting in an integer quotient by truncating any decimal part (e.g., 7 / 3 yields 2). **Floating-point division** occurs if at least one operand is a floating-point number, producing a precise double or float result (e.g., 7.0 / 3 yields 2.333...).
2. The **modulo operator (%)** returns the remainder of a division operation. It is commonly used to determine if a number is even or odd (number % 2 == 0) or to cycle through a range of numbers, like 20 % 3 which results in 2.
3. In its **prefix form (++var)**, the ++ operator increments the variable's value first, and then the new value is used in the expression. In its **postfix form (var++)**, the original value of the variable is used in the expression first, and then the variable is incremented.
4. The "last unmatched if" rule states that an else clause always binds to the most recently declared if statement that does not already have an else. This is significant because incorrect indentation can imply a different mapping, but Java's compiler will always follow this rule, potentially leading to unexpected program logic.
5. Placing a semicolon immediately after an if statement's condition (if (condition);) creates an **empty statement** as the if's body. This means the actual code block intended to be controlled by the if statement will execute unconditionally, leading to a logical error where the program behaves differently than intended.
6. A **conditional loop** (e.g., while loop) is non-deterministic; its number of iterations depends on a condition that may change during execution, so the total count isn't known beforehand. A **counting loop** (e.g., for loop) is deterministic because it uses

a counter, allowing the number of iterations to be determined or predicted before the loop begins.

7. A for loop can lead to an infinite loop if its **condition is always true** (e.g., for (;;) or a condition that never evaluates to false). Another common scenario is if the **adjustment part of the loop is missing or incorrectly implemented** such that the loop control variable never changes in a way that would cause the condition to become false.
8. You use Math.ceil() to round a number **up** to the nearest whole number (e.g., Math.ceil(3.2) is 4.0), and Math.floor() to round a number **down** to the nearest whole number (e.g., Math.floor(3.8) is 3.0). Java's Math.round() rounds X.5 values to the nearest *even* integer, so Math.round(2.5) is 2 and Math.round(3.5) is 4.
9. String.equals() performs a **case-sensitive** comparison between two strings, returning true only if they are identical character by character, including case. String.equalsIgnoreCase() performs a **case-insensitive** comparison, returning true if the strings are identical disregarding whether characters are uppercase or lowercase. equals() is preferred for exact matches, while equalsIgnoreCase() is useful for flexible user input or data retrieval.
10. Java enforces the rule that you **cannot declare a local variable twice in nested blocks** because it creates **ambiguity** for the compiler. If a variable of the same name exists in both an outer and inner scope, the compiler wouldn't know which variable is being referred to at a given point, leading to a compilation error.

Essay Format Questions

1. Discuss the importance of choosing the correct numeric data type in Java, considering memory usage and value limits. Provide examples where selecting an int instead of a long or a float instead of a double could lead to errors or unexpected behavior.
2. Compare and contrast the while, do-while, and for loop structures in Java. Include a discussion of their typical use cases, how their conditions are evaluated, and when one loop type might be more appropriate than another.
3. Explain the concept of operator precedence and associativity in Java, particularly focusing on arithmetic, relational, and logical operators. Provide a complex expression as an example and trace its evaluation step-by-step according to these rules.

4. Describe how characters and strings are handled in Java, including the utility of the Character and String classes. Detail at least five common String methods and how they facilitate string manipulation for practical programming tasks, such as validating user input or formatting output.
5. Analyze the concept of variable scope in Java methods and loops. Explain the "Java Rule" for declaring local variables in non-nesting versus nesting blocks, providing illustrative code snippets to demonstrate valid and invalid declarations according to this rule.

Glossary of Key Terms

- **Absolute Value:** The non-negative value of a number, regardless of its sign. In Java, found using Math.abs().
- **Ambiguity:** A situation where the compiler cannot determine which of multiple possible meanings or interpretations to use, often due to overlapping variable names or incorrect syntax.
- **Assignment Operator (=):** An operator used to assign a value to a variable.
- **Augmented Assignment Operators (+=, -=, *=, /=, %=):** Shorthand operators that combine an arithmetic operation with assignment (e.g., x += y is equivalent to x = x + y).
- **Binary Operator:** An operator that operates on two operands (e.g., +, -, *, /, %, &&, ||).
- **Block Statement:** A group of zero or more statements enclosed in curly braces {}. Often used with if, else, and loop constructs.
- **Boolean Expression:** An expression that evaluates to either true or false. Used as conditions in if statements and loops.
- **break:** A keyword used to prematurely terminate the innermost loop or switch statement, transferring control to the statement immediately following it.
- **char:** A primitive data type in Java used to store a single Unicode character.
- **Character Class:** A built-in Java class (in java.lang) that provides static methods for character manipulation, such as checking if a character is a digit or converting its case.

- **Conditional Operator (? :):** Java's only ternary operator, it evaluates a boolean expression and returns one of two expressions based on whether the condition is true or false.
- **continue:** A keyword used to skip the remainder of the current iteration of the innermost loop and proceed to the next iteration.
- **Counting Loop:** A type of loop (like the for loop) where the number of iterations is typically known or determined before the loop begins. Also known as a deterministic loop.
- **Deterministic Loop:** See **Counting Loop**.
- **double:** A primitive data type in Java representing a double-precision floating-point number. It is the default type for floating-point literals.
- **do-while Loop:** A loop construct where the loop body is executed at least once, and then the loop-continuation condition is evaluated after each iteration.
- **else Mapping Rule:** The rule stating that an else clause always pairs with the most recent unmatched if statement.
- **for Loop:** A loop construct often used for counting or iterating a known number of times, consisting of initialization, condition, and adjustment components.
- **Floating-point Division:** Division where at least one operand is a floating-point number, resulting in a floating-point (decimal) result.
- **if-else-if Statement:** A multi-way conditional construct that evaluates a series of conditions sequentially, executing the block associated with the first true condition.
- **if-else Statement:** A conditional construct that executes one block of code if a condition is true, and a different block if the condition is false.
- **if Statement:** A conditional construct that executes a block of code only if a specified condition is true.
- **Increment Operator (++):** An operator that increases the value of a variable by one. Can be prefix or postfix.
- **Infinite Loop:** A loop that continues to execute indefinitely because its loop-continuation condition never becomes false.
- **Integer Division:** Division where both operands are integers, resulting in an integer quotient with any fractional part truncated.

- **long:** A primitive data type in Java used to store large integer values, with a greater maximum value than int.
- **Loop Body:** The block of statements that are repeatedly executed within a loop.
- **Loop-Continuation Condition:** The boolean expression that controls the execution of a loop. The loop continues as long as this condition is true.
- **Math Class:** A built-in Java class (in java.lang) that provides static methods for common mathematical operations (e.g., pow, sqrt, random, round).
- **Method:** A reusable block of code that performs a specific task.
- **Modulo Operator (%):** An arithmetic operator that returns the remainder of a division.
- **NaN (Not a Number):** A special floating-point value that represents an undefined or unrepresentable numerical result (e.g., the square root of a negative number).
- **Nested Blocks:** Code blocks (e.g., loops, if statements) that are contained within other code blocks.
- **Nested Loops:** A programming construct where one loop is placed inside another loop. The inner loop completes all its iterations for each iteration of the outer loop.
- **Non-deterministic Loop:** See **Conditional Loop**.
- **Numeric Operators:** Symbols used to perform mathematical calculations (+, -, *, /, %).
- **Operand:** A value or variable on which an operator performs an action.
- **Operator Precedence:** The order in which different operators are evaluated in an expression (e.g., multiplication before addition).
- **Palindrome:** A sequence of characters that reads the same forwards and backward (e.g., "mom," "racecar").
- **Postfix Decrement (var--):** The variable's original value is used in the expression, then it is decremented by one.
- **Postfix Increment (var++):** The variable's original value is used in the expression, then it is incremented by one.
- **Prefix Decrement (--var):** The variable is decremented by one, then its new value is used in the expression.

- **Prefix Increment (++var):** The variable is incremented by one, then its new value is used in the expression.
- **Scanner Class:** A class in Java (java.util.Scanner) used to obtain input from primitive types like int, double, and String.
- **Scope (of a variable):** The region of a program where a variable is accessible and can be referenced.
- **String Class:** A built-in Java class used to represent sequences of characters. Strings are immutable objects.
- **trim():** A String method that returns a new string with leading and trailing whitespace characters removed.
- **Unary Operator:** An operator that operates on a single operand (e.g., !, ++, --).
- **Undeterministic Loop:** See **Conditional Loop**.
- **Unicode:** A universal character encoding standard that assigns a unique number to every character, used by Java's char type.
- **while Loop:** A loop construct where the loop-continuation condition is evaluated before each iteration. If the condition is initially false, the loop body may not execute at all.