**Logical Operators:**

Logical operators are used to combine or manipulate Boolean values (true or false). There are three main logical operators: AND (**&&**), OR (**||**), and NOT (**!**).

1. **AND (&&)**:
   * The AND operator returns true only if both operands are true; otherwise, it returns false.
   * It short-circuits; if the left operand evaluates to false, the right operand is not evaluated.

boolean a = true;

boolean b = false;

boolean result = a && b; // result will be false because b is false

**OR (||)**:

* The OR operator returns true if at least one of the operands is true; it returns false if both operands are false.
* Like the AND operator, it also short-circuits; if the left operand evaluates to true, the right operand is not evaluated.

boolean a = true;

boolean b = false;

boolean result = a || b; // result will be true because a is true

**NOT (!)**:

* The NOT operator reverses the logical state of its operand; if a condition is true, the NOT operator makes it false, and vice versa.

Example:

boolean a = true;

boolean result = !a; // result will be false

### Decision Trees for Problem Solving:

Decision trees are hierarchical structures that help in decision-making by visualizing the possible outcomes based on a series of conditions or decisions.

**Example Scenario**: Let's consider a decision tree for deciding whether to go for a walk based on weather conditions and temperature.

Weather

/ \

Sunny Rainy

| |

Temperature > 20°C

| |

Yes No Yes No

\ / \ /

Go No No Stay Home

**Explanation**:

* At the top level, we check the weather condition. If it's sunny, we check the temperature.
* If it's sunny and the temperature is greater than 20°C, we decide to go for a walk; otherwise, we stay home.
* If it's rainy, we always stay home.

public class DecisionTree {

public static void main(String[] args) {

String weather = "Sunny";

int temperature = 25;

if (weather.equals("Sunny")) {

if (temperature > 20) {

System.out.println("Go for a walk");

} else {

System.out.println("Stay home");

}

} else {

System.out.println("Stay home");

}

}

}

In this example:

* We first check if the weather is sunny.
* If it is, we check the temperature. If it's above 20°C, we decide to go for a walk; otherwise, we stay home.
* If the weather is not sunny, we always stay home.

Functions and Modular Design Fundamentals of Function Design Modular Programming Concepts Writing Pseudocode for Modular Functions in java

Functions and modular design are essential concepts in programming, allowing for code organization, reusability, and maintainability. Modular programming involves breaking down a program into smaller, manageable modules or functions, each responsible for performing a specific task. Pseudocode is a high-level description of an algorithm that uses natural language or informal notation to outline the logic of the algorithm without being tied to any specific programming language syntax.

Let's explore these concepts in more detail and write pseudocode for modular functions in Java:

**Fundamentals of Function Design:**

Functions in programming encapsulate a block of code that performs a specific task. They typically have inputs (parameters) and outputs (return values) and can be called from other parts of the program.

Fundamental principles of function design include:

* **Single Responsibility Principle (SRP)**: Each function should have a single, well-defined purpose or responsibility.
* **Modularity**: Functions should be modular, meaning they can be reused and composed to build larger systems.
* **Input and Output**: Functions may take inputs (parameters) and produce outputs (return values).

**Modular Programming Concepts:**

Modular programming involves designing a program by breaking it into independent, interchangeable modules or functions. This approach offers several benefits, including:

* **Code Reusability**: Modules can be reused in different parts of the program or in other programs.
* **Encapsulation**: Modules hide implementation details, exposing only necessary interfaces.
* **Ease of Maintenance**: Modules can be modified or replaced without affecting other parts of the program.

**Writing Pseudocode for Modular Functions in Java:**

Let's write pseudocode for a modular function that calculates the factorial of a number:

Function: factorial(n)

Input: n (integer)

Output: result (integer)

Pseudocode:

1. Start

2. If n is 0 or 1, return 1

3. Otherwise, initialize result to 1

4. Loop from i = 2 to n:

a. Multiply result by i

5. Return result

6. End

Now, let's translate this pseudocode into a Java function:

public class Factorial {

public static int factorial(int n) {

if (n == 0 || n == 1) {

return 1;

}

int result = 1;

for (int i = 2; i <= n; i++) {

result \*= i;

}

return result;

}

public static void main(String[] args) {

int n = 5;

System.out.println("Factorial of " + n + " is: " + factorial(n));

}

}

This Java function calculates the factorial of a given number **n** using a modular approach, following the pseudocode structure. It demonstrates the principles of function design and modular programming.