Unit 1

Features of Java

- Java is a versatile and robust programming language widely used in software development.
- Its features make it one of the most popular languages for building secure, platform-independent, and efficient applications.

Java's key features

- **Simple:**
  - Java is designed to be easy to learn and use.
  - Its syntax is clear and straightforward, reducing the complexity of writing and understanding code.
  - By eliminating complex features like explicit pointers and operator overloading, Java simplifies development.

- **Object-Oriented**
  - Java is a fully object-oriented language, emphasizing the use of objects and classes.
  - Core principles such as inheritance, encapsulation, polymorphism, and abstraction allow developers to create modular, reusable, and maintainable code.
- **Platform-Independent**
  - Java follows the "write once, run anywhere" (WORA) philosophy.
  - Programs written in Java are compiled into bytecode, which can run on any platform with a Java Virtual Machine (JVM).
  - This feature ensures portability across different operating systems.

- **Secure**
  - Java provides a secure environment for application development through its runtime environment and features such as bytecode verification, automatic memory management, and sandboxing.
  - Additionally, Java’s Security APIs enable secure communication, encryption, and authentication.
- **Robust**
  - Java is designed for reliability.
  - Features like garbage collection, exception handling, and type-checking at compile-time and runtime contribute to the language's robustness.
  - Java minimizes errors and prevents application crashes effectively.

- **Multithreaded**
  - Java supports multithreading, allowing developers to write programs that perform multiple tasks simultaneously.
  - This feature is crucial for building high-performance applications such as games, animations, and real-time systems.
- **High Performance**
  - While Java is not as fast as languages like C or C++, it achieves high performance through Just-In-Time (JIT) compilation.
  - JIT compiles bytecode into native machine code at runtime, optimizing execution speed.

- **Distributed**
  - Java facilitates the development of distributed applications through features like Remote Method Invocation (RMI) and support for networking.
  - Java's APIs enable seamless communication between systems over a network.
- **Dynamic**
  - Java is a dynamic language that adapts to evolving environments.
  - It supports dynamic loading of classes, making it flexible and efficient.
  - Developers can add new code and functionalities without affecting existing systems.

- **Interpreted and Compiled**
  - Java combines the benefits of both interpreted and compiled languages.
  - Source code is compiled into bytecode, which the JVM interprets at runtime.
  - This approach ensures efficiency and portability.
- **Architecture-Neutral**
  - Java’s architecture-neutrality comes from its bytecode, which is designed to be executed identically on any hardware or software platform.
  - This characteristic ensures consistent behavior across systems.

- **Portable**
  - Java eliminates hardware and operating system dependencies by providing a standard runtime environment.
  - Its portability extends beyond platform independence, including consistent data type sizes across all platforms.
- **Automatic Memory Management**
  - Java's garbage collection automatically manages memory, freeing developers from manual memory allocation and deallocation.
  - This feature reduces memory leaks and simplifies development.

- **Rich Standard Library**
  - Java boasts a comprehensive standard library that supports a wide range of functionalities, including data structures, algorithms, networking, database connectivity, and user interface development.
  - This reduces the need for third-party libraries and accelerates development.
- **Scalability**
  - Java applications are scalable, making the language suitable for building applications ranging from small-scale projects to enterprise-level systems.
  - Features like multithreading and robust memory management contribute to this scalability.

- **Community Support**
  - Java has a vast and active community of developers, providing extensive resources, tutorials, and forums for support.
  - This ecosystem ensures continuous evolution and assistance for developers.

Input and Output Statements in Java

- Input and output (I/O) in Java are essential operations that enable interaction between a program and the user or other systems.
- Java provides extensive support for I/O operations through its standard libraries.

Overview of Input and Output in Java

- Java uses streams to perform I/O operations.
- A stream is a sequence of data that can be read from or written to a source.
- Java provides the following types of streams:
  - **Input Stream:** Used to read data from a source (e.g., a file, keyboard).
  - **Output Stream:** Used to write data to a destination (e.g., a file, console).

Basic Input and Output Statements

- **Input:** Java provides several ways to handle input:
  - **Using Scanner Class:** The Scanner class, found in the java.util package, is commonly used for reading input from the console. It supports various data types.

```
import java.util.Scanner;
public class InputExample {
        public static void main(String[] args) {
                Scanner scanner = new Scanner(System.in);
                System.out.print("Enter your name: ");
                String name = scanner.nextLine();
                System.out.print("Enter your age: ");
                int age = scanner.nextInt();
                System.out.println("Hello, " + name + "! You are " + age + " years old.");
        }
}
```

- **Using BufferedReader Class:** The BufferedReader class, part of the java.io package, reads text from an input stream efficiently. It is commonly paired with InputStreamReader.
- Example:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
public class BufferedReaderExample {
    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));
        System.out.print("Enter a number: ");
        int number = Integer.parseInt(reader.readLine());
        System.out.println("You entered: " + number);
    }
}
```

Output

- Java provides the System.out stream for console output.
- The most commonly used methods are:
  - print(): Outputs data without a newline.
  - println(): Outputs data followed by a newline.
  - printf(): Formats output based on a specified format string.

- **Example:**

```
public class OutputExample {
    public static void main(String[] args) {
        System.out.print("This is print.");
        System.out.println("This is println.");
        System.out.printf("This is printf: %d",
        123);
    }
}
```

File Input and Output

- Java provides classes in the java.io and java.nio packages for file I/O operations.
  - **File Input:** Using FileReaderThe FileReader class reads data from files.
  - **Example:**

```
import java.io.FileReader;
import java.io.IOException;
public class FileReaderExample {
        public static void main(String[] args) {
                try (FileReader reader = new FileReader("example.txt")) {
                        int character;
                        while ((character = reader.read()) != -1) {
                                System.out.print((char) character);
                        }
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
```

File Output

- Using FileWriterThe FileWriter class writes data to files.
- **Example:**

```
import java.io.FileWriter;
import java.io.IOException;
public class FileWriterExample {
	public static void main(String[] args) {
		try (FileWriter writer = new FileWriter("example.txt")) {
			writer.write("Hello, File!");
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

Advanced I/O Using Streams

- **Byte Streams:** Byte streams handle raw binary data and are represented by InputStream and OutputStream.
- **Example:**

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class ByteStreamExample {
        public static void main(String[] args) {
                try (FileInputStream in = new FileInputStream("input.txt");
                FileOutputStream out = new FileOutputStream("output.txt")){
                        int data;
                        while ((data = in.read()) != -1) {
                                out.write(data);
                         }
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
```

Character Streams

- Character streams handle textual data and are represented by Reader and Writer classes.
- Example:

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
public class CharacterStreamExample {
        public static void main(String[] args) {
                try (FileReader reader = new FileReader("input.txt"); FileWriter writer =
                new FileWriter("output.txt")) {
                        int character;
                        while ((character = reader.read()) != -1) {
                                writer.write(character);
                        }
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
```

Error Handling in I/O

- Java requires developers to handle exceptions during I/O operations. Common exceptions include:
  - IOException: General I/O errors.
  - FileNotFoundException: File not found during read/write operations.
  - EOFException: End of file reached unexpectedly.
- Example:

```
try {
    // I/O operation
} catch (IOException e) {
    System.err.println("An error occurred: " + e.getMessage());
}
```

Command-Line Arguments

- Command-line arguments are strings of information provided to the main method of a Java program.
- The main method signature in Java is:
  - public static void main(String[] args)
    - The args parameter is an array of String objects, where each element represents a single argument passed to the program.
    - The number of arguments is determined by the user at runtime.

How to Provide Command-Line Arguments?

- Command-line arguments are provided when running the Java program from the terminal or command prompt.
- The syntax is:
  - java ProgramName arg1 arg2 arg3
    - ProgramName: The name of the compiled Java class.
    - arg1, arg2, arg3: The arguments to be passed, separated by spaces.

Accessing Command-Line Arguments

- The command-line arguments are stored in the args array and can be accessed using their indices.
- Example:

```
public class CommandLineExample {
    public static void main(String[] args) {
        System.out.println("Number of arguments: " + args.length);
        for (int i = 0; i < args.length; i++) {
            System.out.println("Argument " + i + ": " + args[i]);
        }
    }
}
```

Accessing Command-Line Arguments

- The command-line arguments are stored in the args array and can be accessed using their indices.
- **Example:**

```
public class CommandLineExample {
    public static void main(String[] args) {
        System.out.println("Number of arguments: " + args.length);
        for (int i = 0; i < args.length; i++) {
            System.out.println("Argument " + i + ": " + args[i]);
        }
    }
}
```

- **Running the Program:** java CommandLineExample Hello World 123

Common Use Cases

- **Passing Configuration Values**
  - Command-line arguments can be used to pass configuration values such as file paths, user credentials, or application settings.
- **Example:**

```
public class ConfigExample {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.out.println("Usage: java ConfigExample <username> <password>");
            return;
        }
        String username = args[0];
        String password = args[1];
        System.out.println("Username: " + username);
        System.out.println("Password: " + password);
    }
}
```

- **Running the Program:**
  - java ConfigExample admin 1234
- **Output:**
  - Username: admin
  - Password: 1234

- **Performing Mathematical Operations**
  - Command-line arguments can be used to perform calculations or process data provided by the user.
  - **Example**:

```
public class MathExample {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.out.println("Usage: java MathExample <num1> <num2>");
            return;
        }
        int num1 = Integer.parseInt(args[0]);
        int num2 = Integer.parseInt(args[1]);
        System.out.println("Sum: " + (num1 + num2));
        System.out.println("Product: " + (num1 * num2));
    }
}
```

- **Running the Program:**
  - java MathExample 5 10
- **Output:**
  - Sum: 15
  - Product: 50

• File Operations

- Command-line arguments can specify file paths for reading or writing data.
- **Example**:

```
import java.io.*;
public class FileExample {
        public static void main(String[] args) {
                if (args.length != 1) {
                        System.out.println("Usage: java FileExample <filePath>");
                        return;
                }
                String filePath = args[0];
                try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
                        String line;
                        while ((line = reader.readLine()) != null) {
                                System.out.println(line);
                        }
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
```

- **Running the Program:**
  - java FileExample example.txt
- **Output:**
  - <Contents of example.txt>

- **Error Handling**
  - It is important to handle errors and validate command-line arguments to prevent runtime exceptions.
  - **Example**:

```
public class ErrorHandlingExample {
    public static void main(String[] args) {
        try {
            int num = Integer.parseInt(args[0]);
            System.out.println("Square: " + (num * num));
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error: No argument provided.");
        } catch (NumberFormatException e) {
            System.out.println("Error: Argument is not a valid number.");
        }
    }
}
```

Advantages of Command-Line Arguments

- **Flexibility:** Allows dynamic input without modifying the source code.
- **Automation:** Facilitates scripting and automation of tasks.
- **Simplicity:** Reduces the need for additional input mechanisms.

Limitations

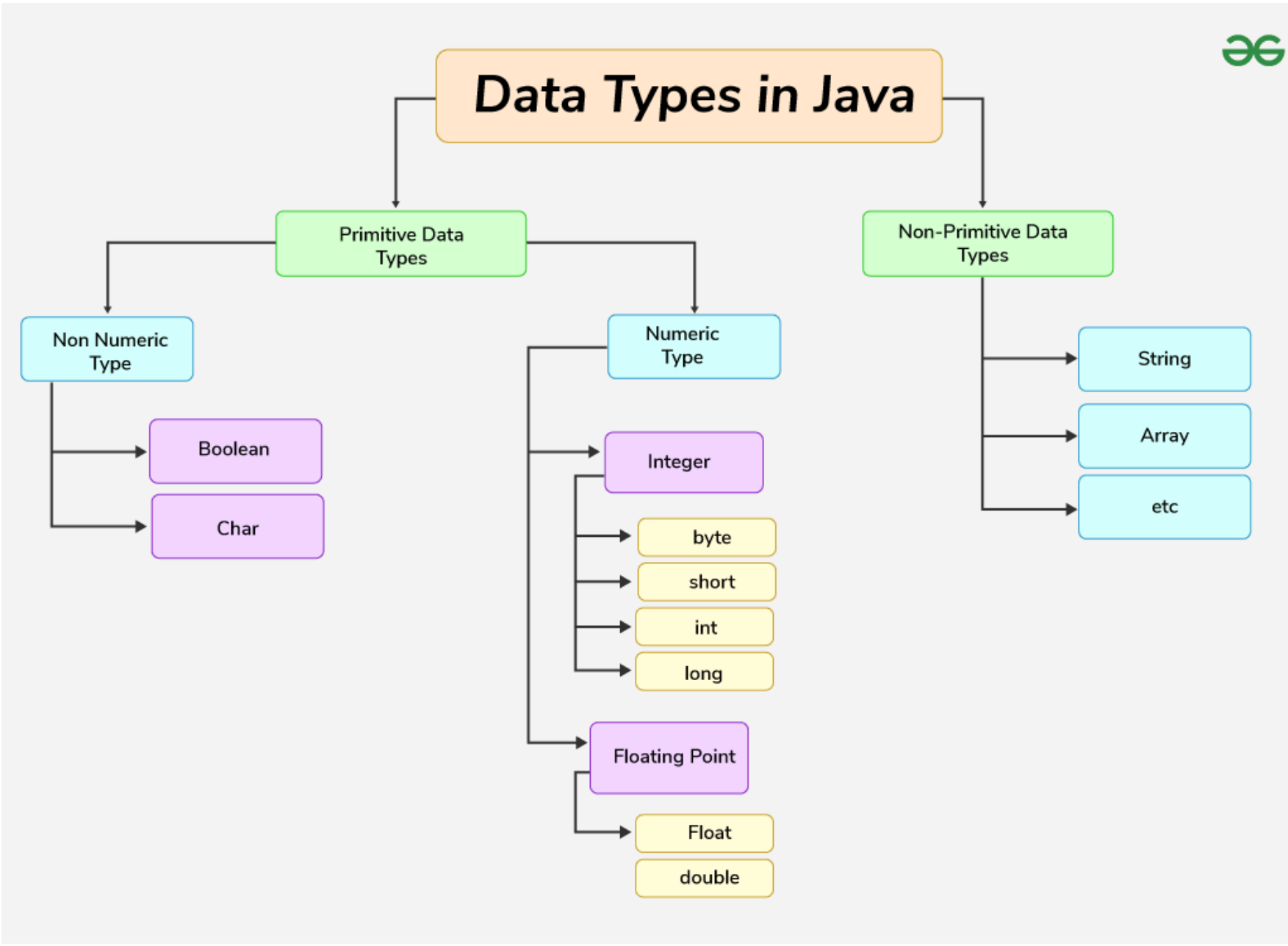
- **Fixed Length:** The number of arguments must be predefined or handled dynamically.
- **Type Safety:** All arguments are treated as strings and need to be explicitly converted to other data types.

Data Types

- In Java, data types specify the type of data that a variable can hold. This helps ensure data integrity and consistency, as Java is a strongly typed language.
- Data types are broadly classified into two categories:
  - Primitive Data Types
  - Non-Primitive Data Types

Primitive Data Types

- The primitive data types include boolean, char, byte, short, int, long, float and double.
- Java Primitive data types:
  - boolean data type
  - byte data type
  - char data type
  - short data type
  - int data type
  - long data type
  - float data type
  - double data type


Data Types in Java
Primitive Data Types
Non-Primitive Data Types
Non Numeric Type
Numeric Type
Boolean
Char
Integer
byte
short
int
long
Floating Point
Float
double
String
Array
etc

Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Non-Primitive Data Types in Java

- In Java, non-primitive data types, also known as reference data types, are used to store complex objects rather than simple values.
- Unlike primitive data types that store the actual values, reference data types store references or memory addresses that point to the location of the object in memory.
- This distinction is important because it affects how these data types are stored, passed, and manipulated in Java programs.

- **Strings**
  - Strings are defined as an array of characters. The difference between a character array and a string in Java is, that the string is designed to hold a sequence of characters in a single variable whereas, a character array is a collection of separate char-type entities. Unlike C/C++, Java strings are not terminated with a null character.
  - Example:

```
// Declare String without using new operator
String s = "GeeksforGeeks";
// Declare String using new operator
String s1 = new String("GeeksforGeeks");
```

- **Class**
  - One common non-primitive data type in Java is the class. Classes are used to create objects, which are instances of the class. A class defines the properties and behaviors of objects, including variables (fields) and methods.
  - For example, you might create a Person class to represent a person, with variables for the person's name, age, and address, and methods to set and get these values.

- **Object**
  - An Object is a basic unit of Object-Oriented Programming and represents real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :
    - State : It is represented by the attributes of an object. It also reflects the properties of an object.
    - Behavior : It is represented by the methods of an object. It also reflects the response of an object to other objects.
    - Identity : It gives a unique name to an object and enables one object to interact with other objects.

- **Interface**
  - Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).
    - Interfaces specify what a class must do and not how. It is the blueprint of the class.
    - An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
    - If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
    - A Java library example is Comparator Interface . If a class implements this interface, then it can be used to sort a collection.

- **Arrays**
  - Arrays are a fundamental non-primitive data type in Java that allow you to store multiple values of the same type in a single variable. Arrays have a fixed size, which is specified when the array is created, and can be accessed using an index. Arrays are commonly used to store lists of values or to represent matrices and other multi-dimensional data structures.
- **Enum**
  - Java also includes other non-primitive data types, such as enums and collections. Enums are used to define a set of named constants, providing a way to represent a fixed set of values. Collections are a framework of classes and interfaces that provide dynamic data structures such as lists, sets, and maps, which can grow or shrink in size as needed.

Variables

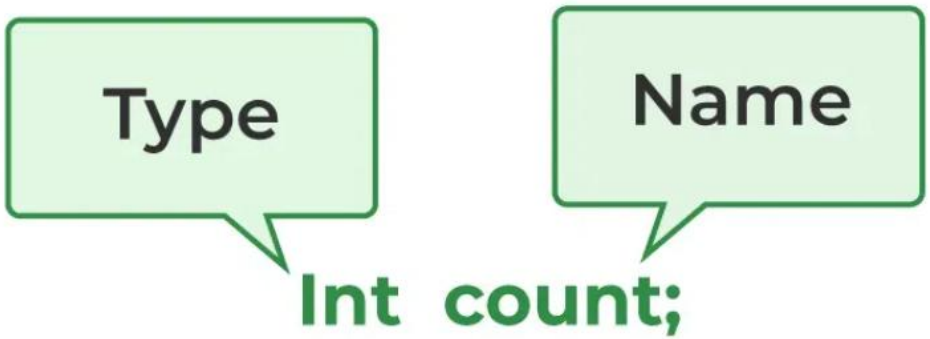
- Variables are the containers for storing the data values or you can also call it a memory location name for the data. Every variable has a:
  - Data Type – The kind of data that it can hold. For example, int, string, float, char, etc.
  - Variable Name – To identify the variable uniquely within the scope.
  - Value – The data assigned to the variable.
- There are three types of variables in Java – Local, Instance, and Static.

- **Example:**

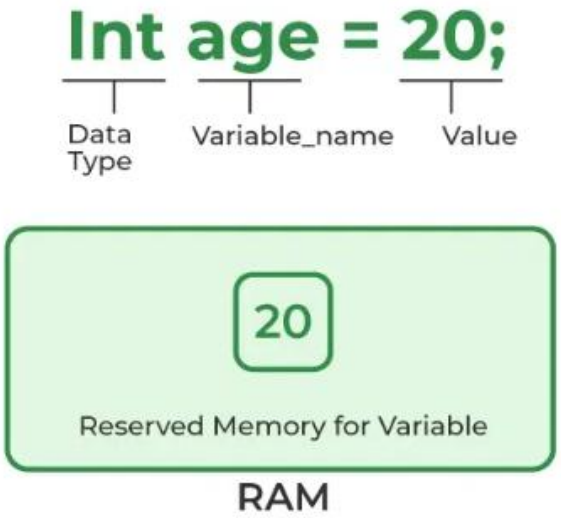
int age = 27;// integer variable having value 27

String name = "gfg" // string variable

- **How to declare variables:**



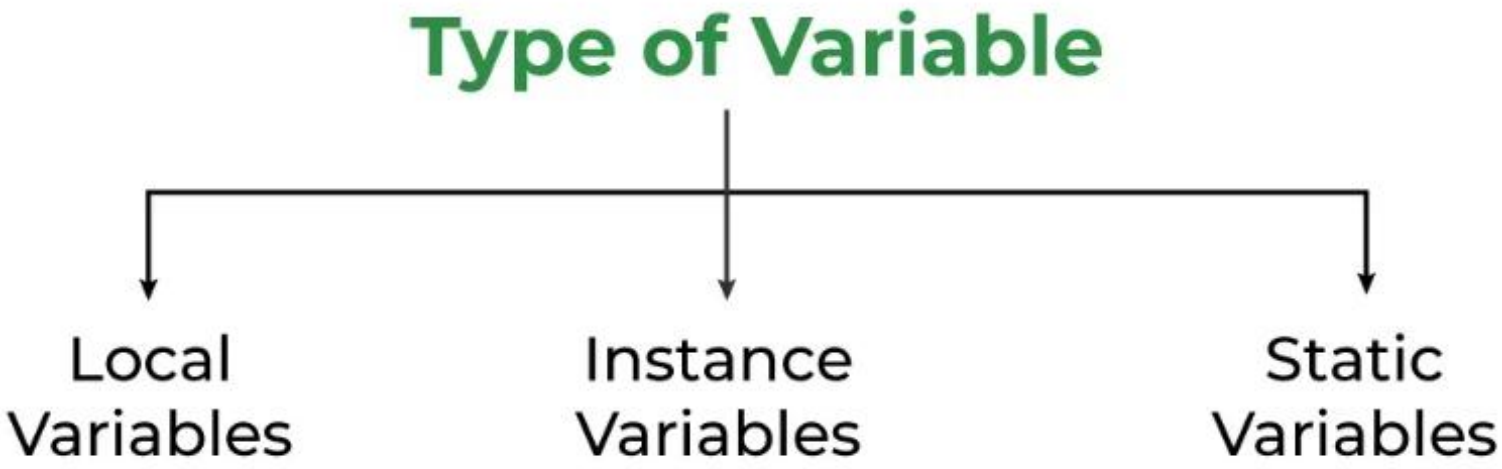
- **How to Initialize:**



- **Example:**

```
// Declaring float variable
float simpleInterest;
// Declaring and initializing integer variable
int time = 10, speed = 20;
// Declaring and initializing character variable
char var = 'h';
```

- **Types of Java Variables**
  - Local Variables
  - Instance Variables
  - Static Variables


Type of Variable
Local Variables
Instance Variables
Static Variables

Local Variables

- A variable defined within a block or method or constructor is called a local variable.
- The Local variable is created at the time of declaration and destroyed after exiting from the block or when the call returns from the function.
- The scope of these variables exists only within the block in which the variables are declared, i.e., we can access these variables only within that block.
- Initialization of the local variable is mandatory before using it in the defined scope.

- Example:
- // Java Program to show the use of local variables
- import java.io.*;

- class GFG {
-     public static void main(String[] args)
-     {
-         // Declared a Local Variable
-         int var = 10;

-         // This variable is local to this main method only
-         System.out.println("Local Variable: " + var);
-     }
- }

Instance Variables

- Instance variables are non-static variables and are declared in a class outside of any method, constructor, or block.
- As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
- Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier, then the default access specifier will be used.
- Initialization of an instance variable is not mandatory. Its default value is dependent on the data type of variable. For String it is null, for float it is 0.0f, for int it is 0, for Wrapper classes like Integer it is null, etc.
- Scope of instance variables are throughout the class except the static contexts.
- Instance variables can be accessed only by creating objects.
- We initialize instance variables using constructors while creating an object. We can also use instance blocks to initialize the instance variables.

```
Example:
// Java Program to show the use of
// Instance Variables
import java.io.*;

class GFG {

    // Declared Instance Variable
    public String geek;
    public int i;
    public Integer I;
    public GFG()
    {
        // Default Constructor
        // initializing Instance Variable
        this.geek = "Shubham Jain";
    }

    // Main Method
    public static void main(String[] args)
    {
        // Object Creation
        GFG name = new GFG();

        // Displaying O/P
        System.out.println("Geek name is: " + name.geek);
        System.out.println("Default value for int is "+ name.i);

        // toString() called internally
        System.out.println("Default value for Integer is "+ name.I);
    }
}
```

Static Variables

- Static variables are also known as class variables.
- These variables are declared similarly to instance variables. The difference is that static variables are declared using the static keyword within a class outside of any method, constructor, or block.
- Unlike instance variables, we can only have one copy of a static variable per class, irrespective of how many objects we create.
- Static variables are created at the start of program execution and destroyed automatically when execution ends.
- Initialization of a static variable is not mandatory. Its default value is dependent on the data type of variable. For String it is null, for float it is 0.0f, for int it is 0, for Wrapper classes like Integer it is null, etc.
- If we access a static variable like an instance variable (through an object), the compiler will show a warning message, which won't halt the program. The compiler will replace the object name with the class name automatically.
- If we access a static variable without the class name, the compiler will automatically append the class name. But for accessing the static variable of a different class, we must mention the class name as 2 different classes might have a static variable with the same name.
- Static variables cannot be declared locally inside an instance method.
- Static blocks can be used to initialize static variables.

```
// Java Program to show the use of
// Static variables
import java.io.*;

class GFG {
    // Declared static variable
    public static String geek = "Shubham Jain";

    public static void main(String[] args)
    {

        // geek variable can be accessed without object
        // creation Displaying O/P GFG.geek --> using the
        // static variable
        System.out.println("Geek Name is : " + GFG.geek);

        // static int c = 0;
        // above line, when uncommented,
        // will throw an error as static variables cannot be
        // declared locally.
    }
}
```

Operators

- Java operators are special symbols that perform operations on variables or values. They can be classified into several categories based on their functionality. These operators play a crucial role in performing arithmetic, logical, relational, and bitwise operations etc.

- **Example:** Here, we are using **+** and **–** operators.
- // Java program to show the use of + and - operators
- public class Geeks
- {
- public static void main(String[] args)
- {
-
- // Declare and initialize variables
- int num1 = 500;
- int num2 = 100;
-
- // Using the + (addition) operator
- int sum = num1 + num2;
- System.out.println("The Sum is: "+sum);
-
- // Using the - (subtraction) operator
- int diff = num1 - num2;
- System.out.println("The Difference is: "+diff);
-
- }
- }

Types of Operators in Java

- Arithmetic Operators
- Unary Operators
- Assignment Operator
- Relational Operators
- Logical Operators
- Ternary Operator
- Bitwise Operators
- Shift Operators
- instance of operator

Arithmetic Operators

- Arithmetic Operators are used to perform simple arithmetic operations on primitive and non-primitive data types.
  - * : Multiplication
  - / : Division
  - % : Modulo
  - + : Addition
  - – : Subtraction

```
Example
// Java Program to show the use of
// Arithmetic Operators
import java.io.*;

class Geeks
{
    public static void main (String[] args)
    {

        // Arithmetic operators on integers
        int a = 10;
        int b = 3;

        // Arithmetic operators on Strings
        String n1 = "15";
        String n2 = "25";

// Convert Strings to integers
        int a1 = Integer.parseInt(n1);
        int b1 = Integer.parseInt(n2);

        System.out.println("a + b = " + (a + b));
        System.out.println("a - b = " + (a - b));
        System.out.println("a * b = " + (a * b));
        System.out.println("a / b = " + (a / b));
        System.out.println("a % b = " + (a % b));
        System.out.println("a1 + b1 = " + (a1 + b1));

    }
}
```

Unary Operators

- Unary Operators need only one operand. They are used to increment, decrement, or negate a value.
  - - , Negates the value.
  - + , Indicates a positive value (automatically converts byte, char, or short to int).
  - ++ , Increments by 1.
    - Post-Increment: Uses value first, then increments.
    - Pre-Increment: Increments first, then uses value.
  - -- , Decrements by 1.
    - Post-Decrement: Uses value first, then decrements.
    - Pre-Decrement: Decrements first, then uses value.
  - ! , Inverts a boolean value.

- Example:
- // Java Program to show the use of
- // Unary Operators
- import java.io.*;

- // Driver Class
- class Geeks {
- // main function
- public static void main(String[] args)
- {
- // Interger declared
- int a = 10;
- int b = 10;
- // Using unary operators
- System.out.println("Postincrement : " + (a++));
- System.out.println("Preincrement : " + (++a));
- System.out.println("Postdecrement : " + (b--));
- System.out.println("Predecrement : " + (--b));
- }
- }

Assignment Operator

- ‘=’ Assignment operator is used to assign a value to any variable. It has right-to-left associativity, i.e. value given on the right-hand side of the operator is assigned to the variable on the left, and therefore right-hand side value must be declared before using it or should be a constant.

- In many cases, the assignment operator can be combined with others to create shorthand compound statements. For example, a += 5 replaces a = a + 5. Common compound operators include:
  - += , Add and assign.
  - -= , Subtract and assign.
  - *= , Multiply and assign.
  - /= , Divide and assign.
  - %= , Modulo and assign.

```
Example
// Java Program to show the use of
// Assignment Operators
import java.io.*;

// Driver Class
class Geeks {
    // Main Function
    public static void main(String[] args)
    {

        // Assignment operators
        int f = 7;
        System.out.println("f += 3: " + (f += 3));
        System.out.println("f -= 2: " + (f -= 2));
        System.out.println("f *= 4: " + (f *= 4));
        System.out.println("f /= 3: " + (f /= 3));
        System.out.println("f %= 2: " + (f %= 2));
        System.out.println("f &= 0b1010: " + (f &= 0b1010));
        System.out.println("f |= 0b1100: " + (f |= 0b1100));
        System.out.println("f ^= 0b1010: " + (f ^= 0b1010));
        System.out.println("f <<= 2: " + (f <<= 2));
        System.out.println("f >>= 1: " + (f >>= 1));
        System.out.println("f >>>= 1: " + (f >>>= 1));
    }
}
```

Relational Operators

- Relational Operators are used to check for relations like equality, greater than, and less than. They return boolean results after the comparison and are extensively used in looping statements as well as conditional if-else statements. The general format is ,
  - *variable* ***relation_operator*** *value*

- Relational operators compare values and return boolean results:
  - == , Equal to.
  - != , Not equal to.
  - < , Less than.
  - <= , Less than or equal to.
  - > , Greater than.
  - >= , Greater than or equal to.

- Example
- // Java Program to show the use of
- // Relational Operators
- import java.io.*;

- // Driver Class
- class Geeks {
- // main function
- public static void main(String[] args)
- {
- // Comparison operators
- int a = 10;
- int b = 3;
- int c = 5;
- System.out.println("a > b: " + (a > b));
- System.out.println("a < b: " + (a < b));
- System.out.println("a >= b: " + (a >= b));
- System.out.println("a <= b: " + (a <= b));
- System.out.println("a == c: " + (a == c));
- System.out.println("a != c: " + (a != c));
- }
- }

Logical Operators

- Logical Operators are used to perform “logical AND” and “logical OR” operations, similar to AND gate and OR gate in digital electronics. They have a short-circuiting effect, meaning the second condition is not evaluated if the first is false.
- Conditional operators are:
  - &&, Logical AND: returns true when both conditions are true.
  - ||, Logical OR: returns true if at least one condition is true.
  - !, Logical NOT: returns true when a condition is false and vice-versa

- Example
- // Java Program to show the use of
- // Logical operators
- import java.io.*;

- class Geeks {
-
- // Main Function
- public static void main (String[] args) {
-
- // Logical operators
- boolean x = true;
- boolean y = false;
-
- System.out.println("x && y: " + (x && y));
- System.out.println("x || y: " + (x || y));
- System.out.println("!x: " + (!x));
- }
- }

Ternary operator

- The Ternary Operator is a shorthand version of the if-else statement. It has three operands and hence the name Ternary. The general format is ,
  - *condition* ***?*** *if true* ***:*** *if false*

- Example
- // Java program to illustrate
- // max of three numbers using
- // ternary operator.

- public class Geeks {
-
- public static void main(String[] args)
- {
- int a = 20, b = 10, c = 30, result;

- // result holds max of three
- // numbers
- result = ((a > b) ? (a > c) ? a : c : (b > c) ? b : c);
- System.out.println("Max of three numbers = "+ result);
- }
- }

Bitwise Operators

- Bitwise Operators are used to perform the manipulation of individual bits of a number and with any of the integer types. They are used when performing update and query operations of the Binary indexed trees.
  - & (Bitwise AND) – returns bit-by-bit AND of input values.
  - | (Bitwise OR) – returns bit-by-bit OR of input values.
  - ^ (Bitwise XOR) – returns bit-by-bit XOR of input values.
  - ~ (Bitwise Complement) – inverts all bits (one's complement).

- Example
- // Java Program to show the use of
- // bitwise operators
- import java.io.*;

- class Geeks
- {
- public static void main(String[] args)
- {
- // Bitwise operators
- int d = 0b1010;
- int e = 0b1100;
-
- System.out.println("d & e : " + (d & e));
- System.out.println("d | e : " + (d | e));
- System.out.println("d ^ e : " + (d ^ e));
- System.out.println("~d : " + (~d));
- System.out.println("d << 2 : " + (d << 2));
- System.out.println("e >> 1 : " + (e >> 1));
- System.out.println("e >>> 1 : " + (e >>> 1));
- }
- }

Shift Operators

- Shift Operators are used to shift the bits of a number left or right, thereby multiplying or dividing the number by two, respectively. They can be used when we have to multiply or divide a number by two. The general format ,
  - *number* ***shift_op*** *number_of_places_to_shift;*

- << (Left shift) – Shifts bits left, filling 0s (multiplies by a power of two).
- >> (Signed right shift) – Shifts bits right, filling 0s (divides by a power of two), with the leftmost bit depending on the sign.
- >>> (Unsigned right shift) – Shifts bits right, filling 0s, with the leftmost bit always 0.

- Example:
- // Java Program to show the use of
- // shift operators
- import java.io.*;

- class Geeks
- {
- public static void main(String[] args)
- {
- int a = 10;
-
- // Using left shift
- System.out.println("a<<1 : " + (a << 1));
-
- // Using right shift
- System.out.println("a>>1 : " + (a >> 1));
- }
- }

instanceof operator

- The instance of operator is used for type checking. It can be used to test if an object is an instance of a class, a subclass, or an interface. The general format ,
  - *object* ***instance of*** *class/subclass/interface*

- **Example**
- // Java program to show the use of
- // Instance of operator
- public class Geeks
- {
- public static void main(String[] args)
- {

- Person obj1 = new Person();
- Person obj2 = new Boy();

- // As obj is of type person, it is not an
- // instance of Boy or interface
- System.out.println("obj1 instanceof Person: "
- + (obj1 instanceof Person));
- System.out.println("obj1 instanceof Boy: "
- + (obj1 instanceof Boy));
- System.out.println("obj1 instanceof MyInterface: "
- + (obj1 instanceof MyInterface));

- // Since obj2 is of type boy,
- // whose parent class is person
- // and it implements the interface Myinterface
- // it is instance of all of these classes

- System.out.println("obj2 instanceof Person: "
- + (obj2 instanceof Person));
- System.out.println("obj2 instanceof Boy: "
- + (obj2 instanceof Boy));
- System.out.println("obj2 instanceof MyInterface: "
- + (obj2 instanceof MyInterface));
- }
- }

- // Classes and Interfaces used
- // are declared here
- class Person {
- }

- class Boy extends Person implements MyInterface {
- }

- interface MyInterface {
- }

Java Control Statements | Control Flow in Java

- Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Types of control flow statements.

- Decision Making statements
  - if statements
  - switch statement
- Loop statements
  - do while loop
  - while loop
  - for loop
  - for-each loop
- Jump statements
  - break statement
  - continue statement

Decision Making in Java

- Decision-making statements in Java execute a block of code based on a condition. Decision-making in programming is similar to decision-making in real life. In programming, we also face situations where we want a certain block of code to be executed when some condition is fulfilled.

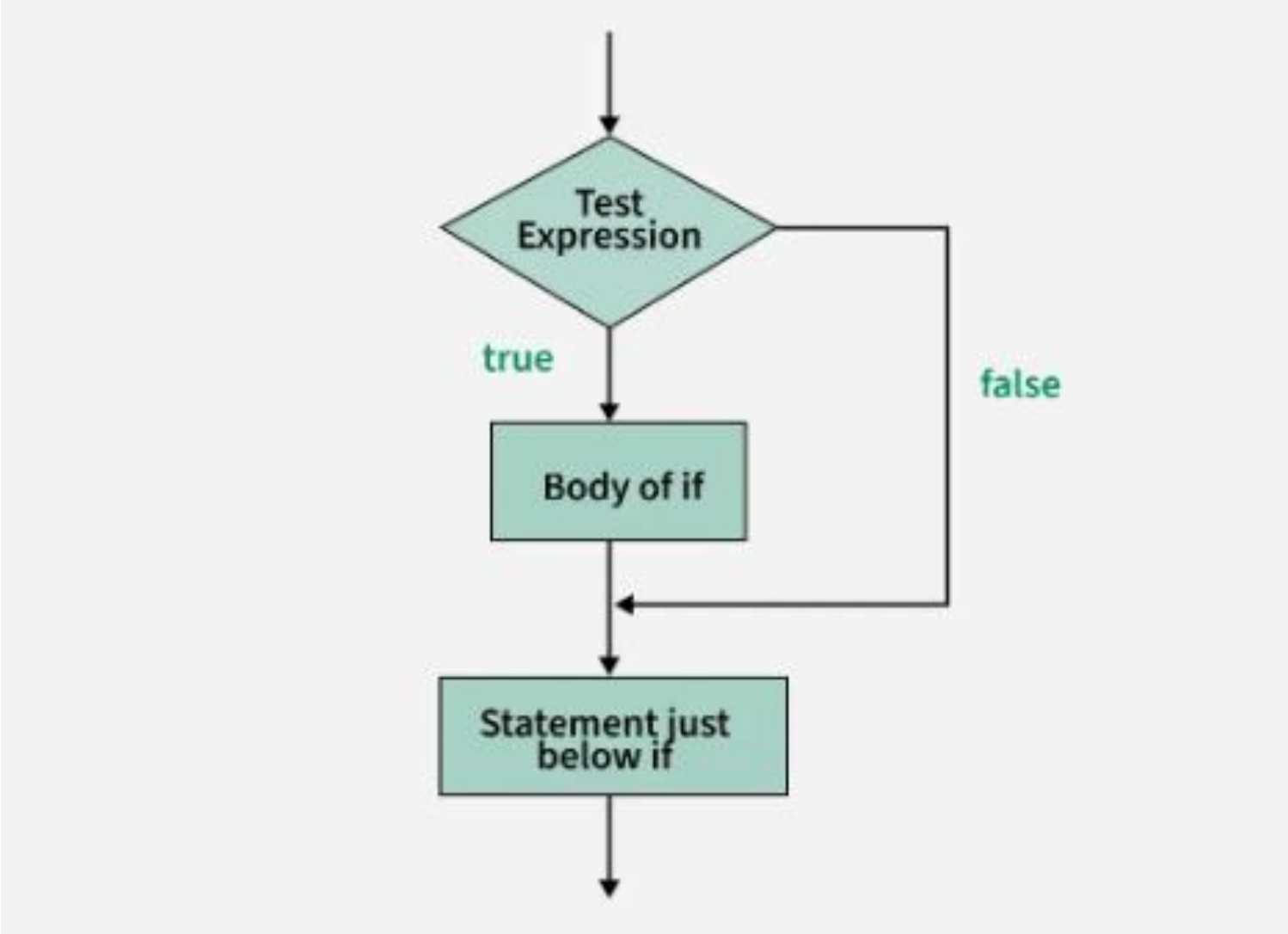
Types of Decision-Making Statements

- if
- if-else
- nested-if
- if-else-if
- switch-case
- jump – break, continue, return

- Java if Statement
  - The if statement is the most simple decision-making statement.
  - It is used to decide whether a certain statement or block of statements will be executed or not i.e. if a certain condition is true then a block of statements is executed otherwise not.
  - Syntax:

```
if(condition)  {
  // Statements to execute if
// condition is true
}
```

- if Statement Execution Flow



- **Example:**
- // Java program to illustrate
- // if statement without curly block
- import java.util.*;

- class Geeks {
- public static void main(String args[])
- {
- int i = 10;

- if (i < 15)
-
- // part of if block(immediate one statement
- // after if condition)
- System.out.println("Inside If block");

- // always executes as it is outside of if block
- System.out.println("10 is less than 15");

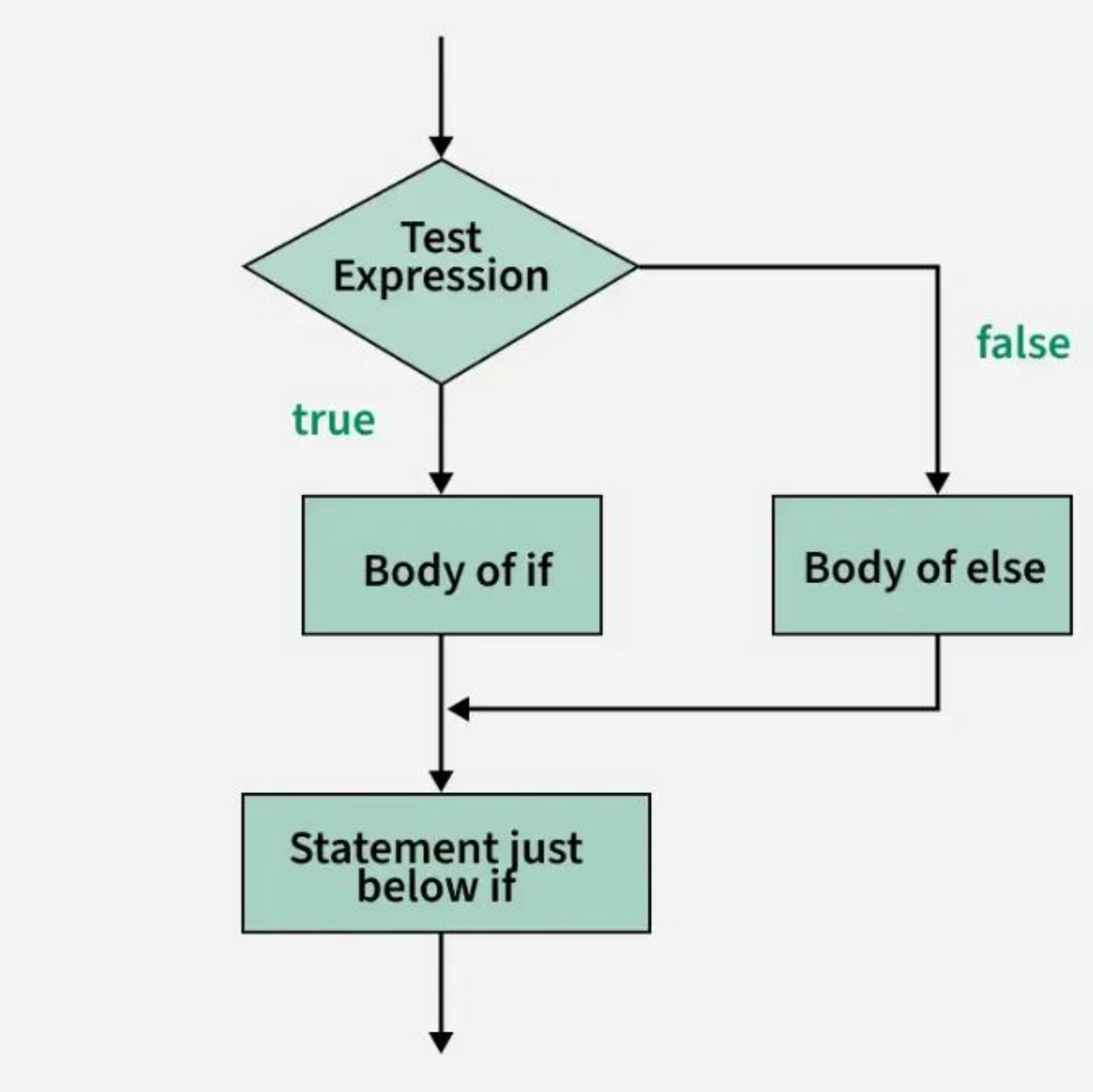
- // This statement will be executed
- // as if considers one statement by default again
- // below statement is outside of if block
- System.out.println("I am Not in if");
- }
- }

Java if-else Statement

- The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't.
- But what if we want to do something else if the condition is false? Here, comes the "else" statement. We can use the else statement with the if statement to execute a block of code when the condition is false.
- Syntax:

```
if(condition){
// Executes this block if
// condition is true
}else{
// Executes this block if
// condition is false
}
```

- if-else Statement Execution flow



- **Example:**

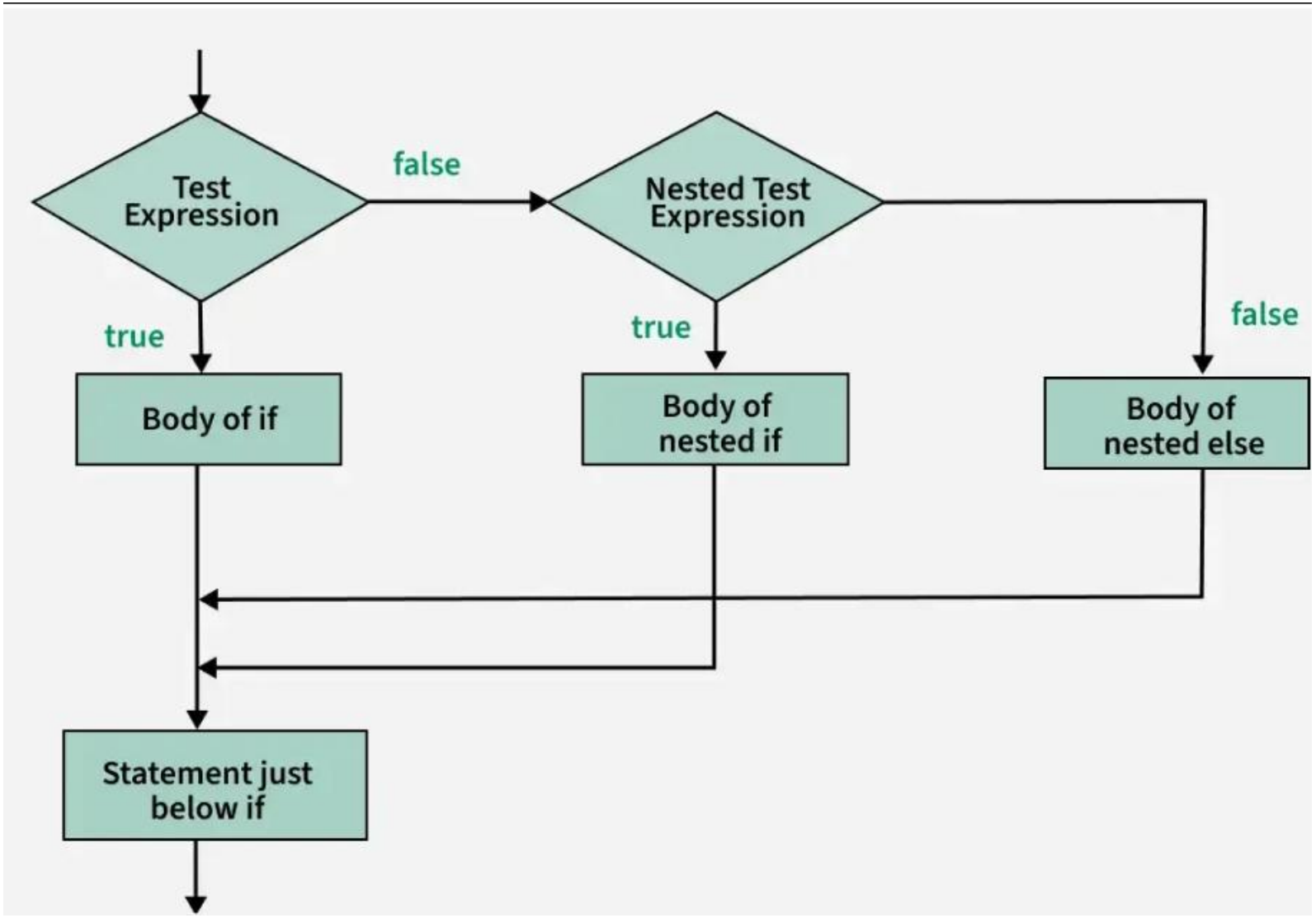
```
// Java program to demonstrate
// the working of if-else statement
import java.util.*;

class Geeks {
    public static void main(String args[])
    {
        int i = 10;

        if (i < 15)
            System.out.println("i is smaller than 15");
        else
            System.out.println("i is greater than 15");
    }
}
```

Java nested-if Statement

- A nested if is an if statement that is the target of another if or else. Nested if statements mean an if statement inside an if statement. Yes, java allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.
- Syntax:

```
if (condition1) {
// Executes when condition1 is true
if (condition2)
{
// Executes when condition2 is true
}
}
```

- nested-if Statement Execution Flow



- **Example:**
- // Java program to demonstrate the
- // working of nested-if statement
- import java.util.*;
- class Geeks {
- public static void main(String args[])
- {
- int i = 10;
- if (i == 10 || i < 15) {
- // First if statement
- if (i < 15)
- System.out.println("i is smaller than 15");
- // Nested - if statement
- // Will only be executed if statement above
- // it is true
- if (i < 12)
- System.out.println(
- "i is smaller than 12 too");
- }
- else {
- System.out.println("i is greater than 15");
- }
- }
- }

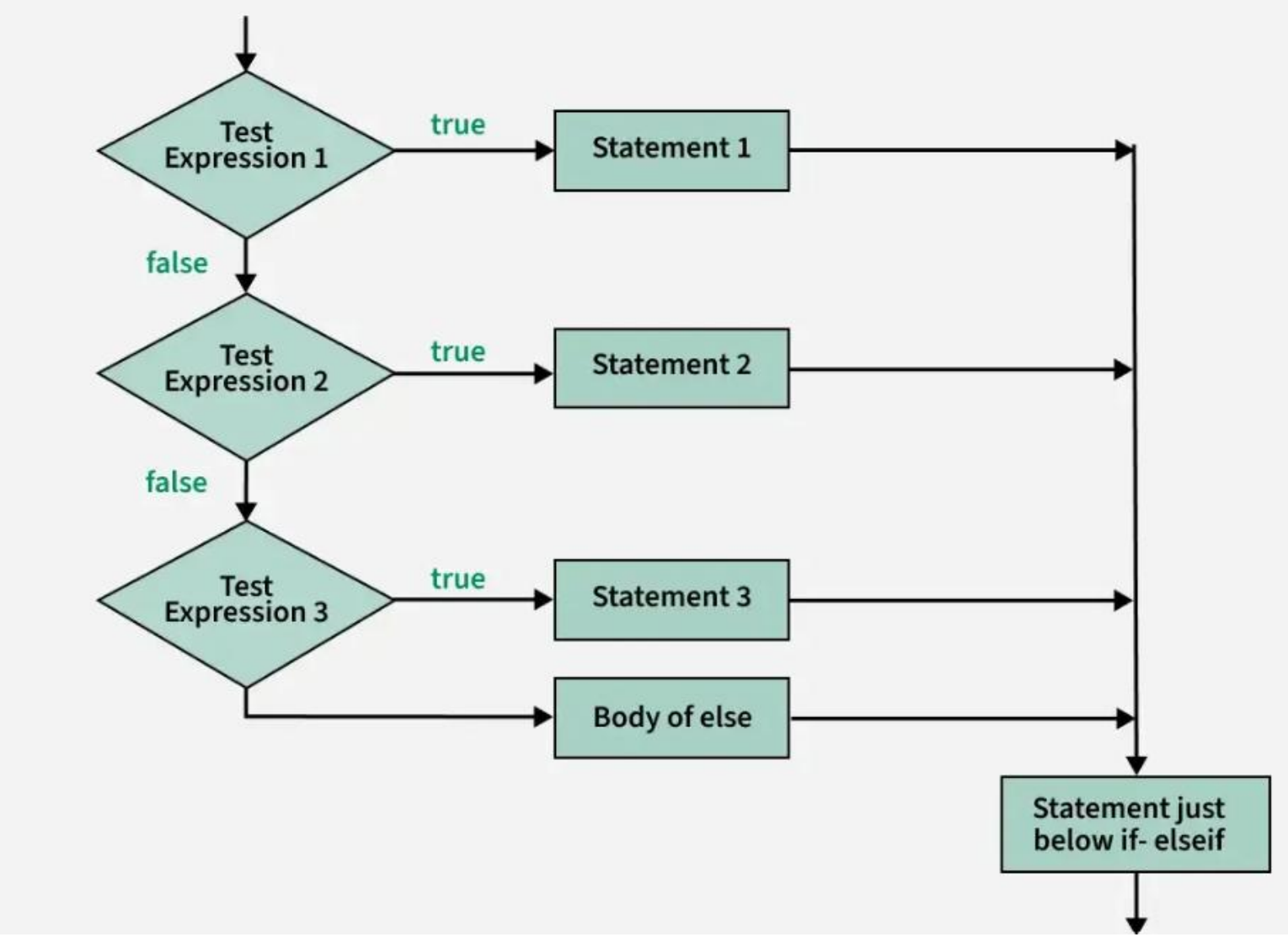
Java if-else-if ladder

- Here, a user can decide among multiple options.The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that ‘if’ is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. There can be as many as ‘else if’ blocks associated with one ‘if’ block but only one ‘else’ block is allowed with one ‘if’ block.

- Syntax:

```
if (condition1) {
// code to be executed if condition1 is true
} else if (condition2) {
// code to be executed if condition2 is true
} else {
// code to be executed if all conditions are false
}
```

- if-else-if ladder Execution Flow



- **Example:**

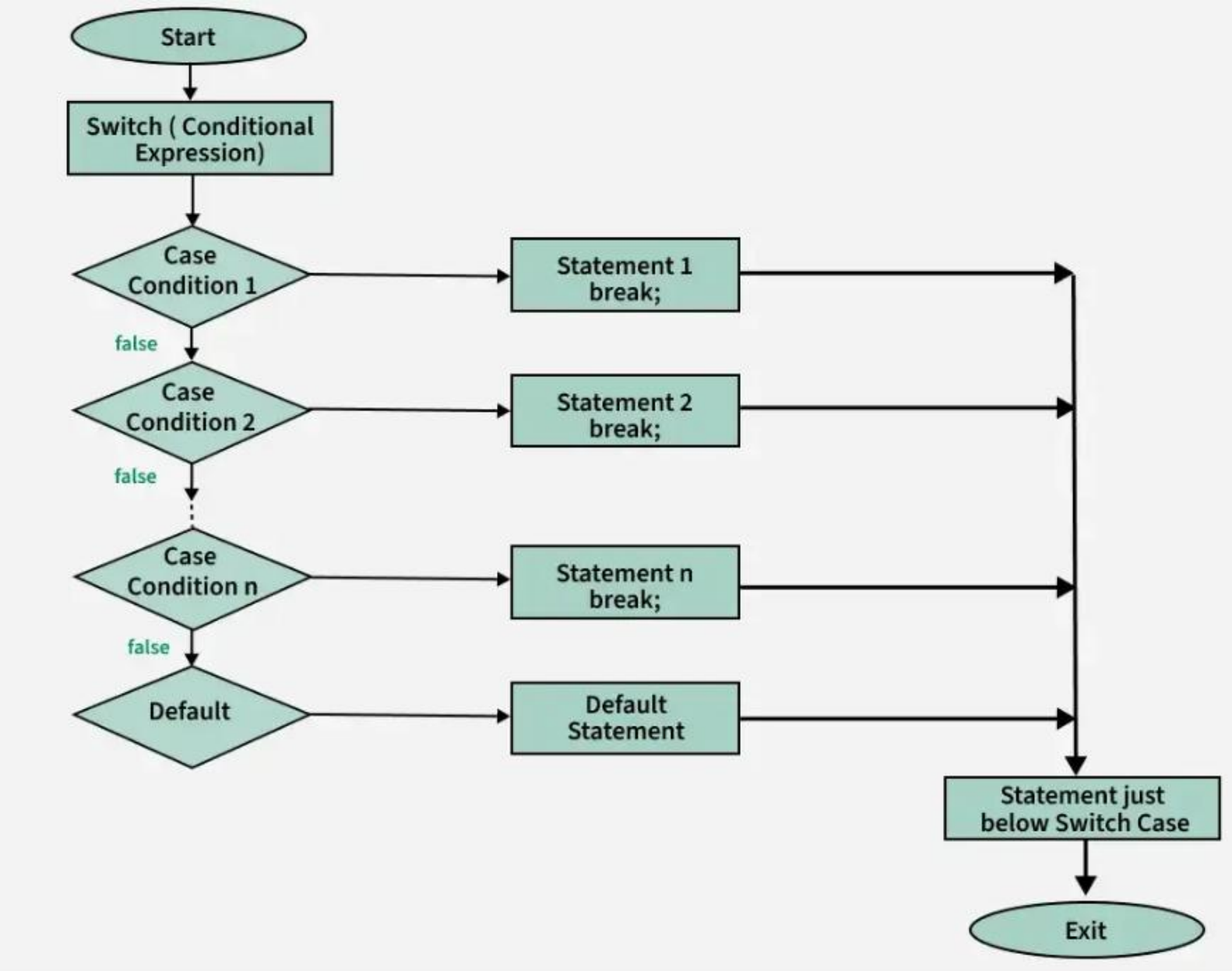
```
// Java program to demonstrate the
//  working of if-else-if ladder
import java.util.*;

class Geeks {
    public static void main(String args[])
    {
        int i = 20;

        if (i == 10)
            System.out.println("i is 10");
        else if (i == 15)
            System.out.println("i is 15");
        else if (i == 20)
            System.out.println("i is 20");
        else
            System.out.println("i is not present");
    }
}
```

Java Switch Case

- The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.
- Syntax:

```
switch (expression) {
case value1:
// code to be executed if expression == value1
break;
case value2:
// code to be executed if expression == value2
break;
// more cases…
default:
// code to be executed if no cases match
}
```

- switch Statements Execution Flow

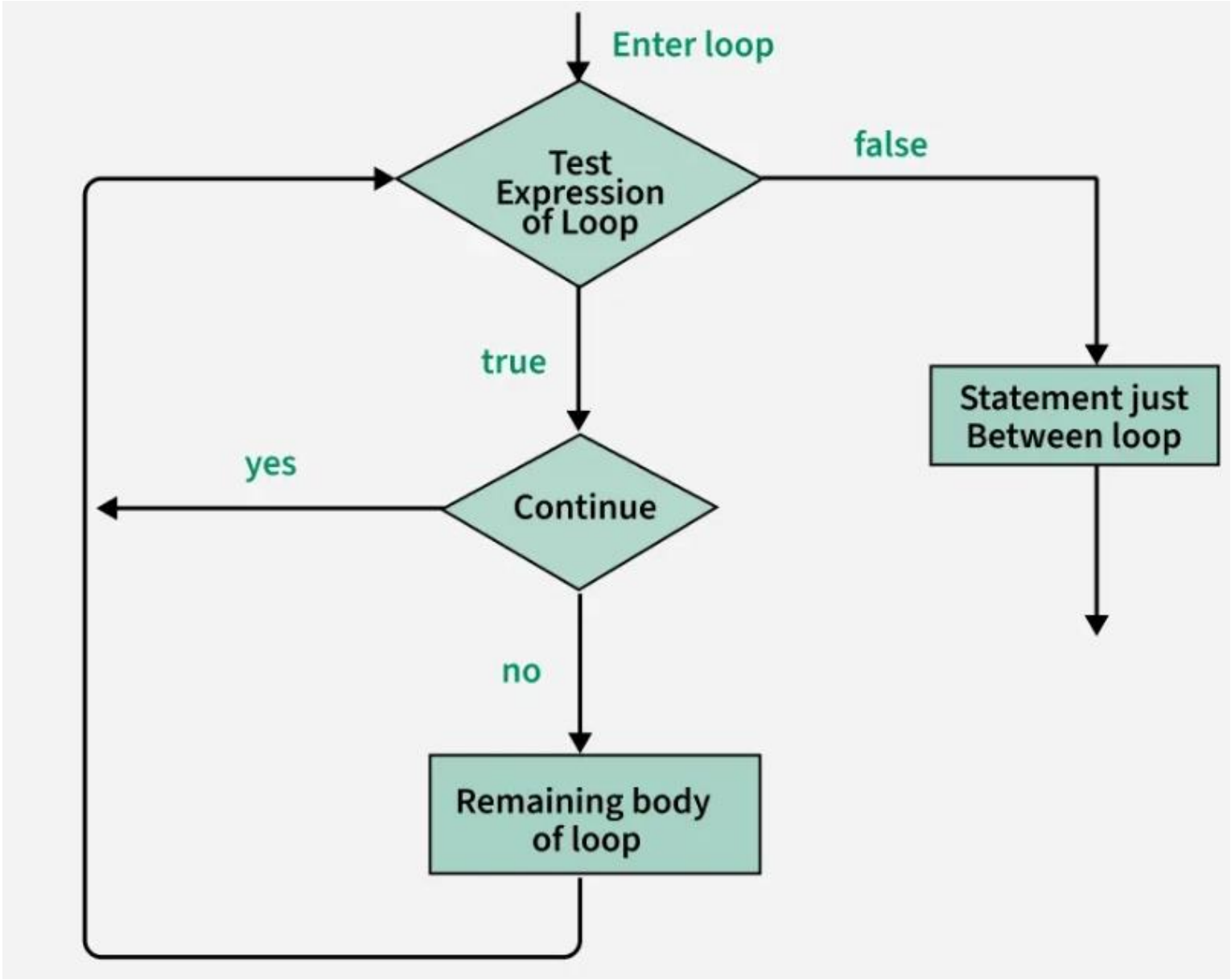


- **Example**
- // Java program to demonstrates the
- // working of switch statements
- import java.io.*;
- class Geeks {
- public static void main(String[] args)
- {
- int num = 20;
- switch (num) {
- case 5:
- System.out.println("It is 5");
- break;
- case 10:
- System.out.println("It is 10");
- break;
- case 15:
- System.out.println("It is 15");
- break;
- case 20:
- System.out.println("It is 20");
- break;
- default:
- System.out.println("Not present");
- }
- }
- }

jump Statements

- Java supports three jump statements: break, continue and return. These three statements transfer control to another part of the program.
- Break: In Java, a break is majorly used for:
  - Terminate a sequence in a switch statement (discussed above).
  - To exit a loop.
  - Used as a “civilized” form of goto.
- Continue: Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop’s end. The continue statement performs such an action.

- **jump Statements Execution Flow**



- **Example**

```
// Java program to demonstrates the use of
// continue in an if statement
import java.util.*;

class Geeks {
    public static void main(String args[])
    {
        for (int i = 0; i < 10; i++) {

            // If the number is even
            // skip and continue
            if (i % 2 == 0)
                continue;

            // If number is odd, print it
            System.out.print(i + " ");
        }
    }
}
```

Return Statement

- The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.
- **Example:** The below Java program demonstrates how the return statements stop a method and skips the rest of the code.

```
// Java program to demonstrate the use of return
import java.util.*;

public class Geeks {
    public static void main(String args[])
    {
        boolean t = true;
        System.out.println("Before the return.");

        if (t)
            return;

        // Compiler will bypass every statement
        // after return
        System.out.println("This won't execute.");
    }
}
```

Arrays in Java

- Arrays are fundamental structures in Java that allow us to store multiple values of the same type in a single variable.
- They are useful for storing and managing collections of data.
- Arrays in Java are objects, which makes them work differently from arrays in C/C++ in terms of memory management.