THE COMPLETE CORE JAVA COURSE

PRESENTED BY,

KGM TECHNICAL TEAM

SLIDES FOR THEORY LECTURES

KGiSL MicrCollege

Genius as a Service (GaaS)

THE COMPLETE CORE JAVA COURSE

PRESENTED BY,

KGM TECHNICAL TEAM

DAY-02

Welcome Section

LECTURE

Watch before you Start

Control Statements in Java

- Control statements help manage the flow of execution in a Java program.
- They are broadly categorized into **conditional statements, looping statements,** and **jump statements.**

---

1. Conditional Statements

Definition:

- Conditional statements allow the program to make decisions based on conditions.
- Depending on whether the condition evaluates to true or false, different code blocks execute.

Types of Conditional Statements:

a. if Statement:

Executes a block of code if a specified condition is true.

Syntax:

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

Example:

```
int number = 10;
if (number > 0) {
   System.out.println("The number is positive.");
}
```

Output:

The number is positive.

Types of Conditional Statements:

b. if-else Statement:

Executes one block of code if the condition is true and another block if the condition is false.

Syntax:

```
if (condition) {
   // code if condition is true
} else {
   // code if condition is false
}
```

Example:

```
int number = -5;
if (number > 0) {
   System.out.println("Positive number.");
} else {
   System.out.println("Non-positive number.");
}
```

Output:

Non-positive number.

Types of Conditional Statements:

c. if-else if-else Ladder:

Checks multiple conditions one after another.

Syntax:

```
if (condition1) {
   // code for condition1
} else if (condition2) {
   // code for condition2
} else {
   // code if all conditions are false
}
```

Example:

```
int marks = 85;
if (marks >= 90) {
    System.out.println("Grade: A+");
} else if (marks >= 75) {
    System.out.println("Grade: A");
} else {
    System.out.println("Grade: B");
}
```

Output:

Grade: A

Types of Conditional Statements:

d. switch-case Statement:

Executes one code block among many based on the value of an expression.

Syntax:

```
switch (variable) {
  case value1:
    // code for value1
    break;
  case value2:
    // code for value2
    break;
  default:
    // default code
}
```

Example:

```
int day = 3;
switch (day) {
   case 1:
      System.out.println("Monday");
      break;
   case 2:
      System.out.println("Tuesday");
      break;
   case 3:
      System.out.println("Wednesday");
      break;
   default:
      System.out.println("Invalid day");
}
```

Output for Example:

Output:

Wednesday

Control Statements in Java

2. Looping Statements

Definition:

- Looping statements execute a block of code multiple times as long as a condition is true.

Types of Loops

- for Loop (Entry Controlled Loop)
- while Loop (Entry Check Loop)
- do while Loop (Exit Check Loop)

Types of Loops:

a. for Loop:

Iterates a block of code for a specified number of times.

Syntax:

```
for (initialization; condition; update) {
   // code to execute
}
```

Example:

```
for (int i = 1; i <= 5; i++) {
    System.out.println("Count: " + i);
}
```

Output:

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
```

Types of Loops:

b. while Loop:

Executes a block of code as long as the condition is true.

Syntax:

```
while (condition) {
   // code to execute
}
```

Example:

```
int i = 1;
while (i <= 3) {
    System.out.println("Hello, World!");
    i++;
}
```

Output:

```
Hello, World!
Hello, World!
Hello, World!
```

Types of Loops:

c. do-while Loop:

Executes the block of code at least once, then repeats the execution as long as the condition is true.

Syntax:

```
do {
   // code to execute
} while (condition);
```

Example:

```
int i = 1;
do {
   System.out.println("Value: " + i);
   i++;
} while (i <= 3);
```

Output:

```
Value: 1
Value: 2
Value: 3
```

Control Statements in Java

3. Jump Statements

Definition:

- Jump statements control the flow of loops by altering their normal execution.

Types of Jump Statements

- break
- continue

Types of Jump Statements:

a. break:

Exits the loop or switch-case statement immediately.

Syntax:

```
jump-statement;
break;
```

Example:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        break; // Exit the loop
    }
    System.out.println("Number: " + i);
}
```

Output:

Number: 1

Number: 2

Types of Jump Statements:

b. continue:

Skips the current iteration of the loop and continues with the next iteration.

Syntax:

```
jump-statement;
continue;
```

Example:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue; // Skip the rest of the code for this iteration
    }
    System.out.println("Number: " + i);
}
```

Output:

```
Number: 1
Number: 2
Number: 4
Number: 5
```

String Handling and Manipulation

Definition:

- In Java, a String is a sequence of characters.
- It is an object that represents a collection of characters and is widely used for handling and manipulating text.
- Strings in Java are immutable, meaning their value cannot be changed after creation.

Why Use Strings?

- To store and manipulate textual data.
- Commonly used for input/output, file handling, and data processing.

Declaration and Initialization of Strings:

- String str1 = "Hello"; // Using string literal
- String str2 = new String("World"); // Using `new` keyword

Common String Methods

1. length()

- Returns the number of characters in the string.

Example:

```
public class StringLengthExample {
  public static void main(String[] args) {
     String str = "Java Programming";
     System.out.println("Length: " + str.length());
  }
}
```

Output:

```
Length: 16
```

Common String Methods

2. substring(int beginIndex)

- Returns a part of the string starting from the specified index.

2.1 substring(int beginIndex, int endIndex)

- Returns the part of the string between beginIndex (inclusive) and endIndex (exclusive).

Example for substring method

Example:

```
public class SubstringExample {
  public static void main(String[] args) {
    String str = "Java Programming";

    System.out.println("Substring from index 5: " + str.substring(5));
    System.out.println("Substring from index 0 to 4: " + str.substring(0, 4));
  }
}
```

Output:

```
Substring from index 5: Programming
Substring from index 0 to 4: Java
```

Common String Methods

3. indexOf(String str)

- Returns the index of the first occurrence of the specified substring. Returns -1 if not found.

Example:

```
public class IndexOfExample {
  public static void main(String[] args) {
    String str = "Java Programming";
    System.out.println("Index of 'Prog': " + str.indexOf("Prog"));
    System.out.println("Index of 'z': " + str.indexOf('z')); // Not found
  }
}
```

Output:

```
Index of 'Prog': 5
Index of 'z': -1
```

Common String Methods

4. toUpperCase() and toLowerCase()

- Converts the string to uppercase or lowercase.

Example:

```
public class CaseConversionExample {
  public static void main(String[] args) {
    String str = "Java Programming";

    System.out.println("Uppercase: " + str.toUpperCase());
    System.out.println("Lowercase: " + str.toLowerCase());
  }
}
```

Output:

```
Uppercase: JAVA PROGRAMMING
Lowercase: java programming
```

Common String Methods

5. replace(char oldChar, char newChar)

- Replaces all occurrences of a character with another character.

Example:

```
public class ReplaceExample {
  public static void main(String[] args) {
    String str = "Java Programming";

    System.out.println("Replace 'a' with 'x': " + str.replace('a', 'x'));
  }
}
```

Output:

```
Replace 'a' with 'x': Jxvx Progrxmming
```

Common String Methods

6. equals(String anotherString)

- Checks if two strings are equal.

Example:

```
public class EqualsExample {
  public static void main(String[] args) {
    String str1 = "Java";
    String str2 = "java";
    System.out.println("Case-sensitive equals: " + str1.equals(str2));
    System.out.println("Case-insensitive equals: " + str1.equalsIgnoreCase(str2));
  }
}
```

Output:

```
Case-sensitive equals: false
Case-insensitive equals: true
```

Common String Methods

7. charAt(int index)

- Returns the character at the specified index.

Example:

```
public class CharAtExample {
  public static void main(String[] args) {
    String str = "Java";

    System.out.println("Character at index 2: " + str.charAt(2));
  }
}
```

Output:

```
Character at index 2: v
```

Additional Example: Combining Methods

Example:

```
public class StringMethodsExample {
  public static void main(String[] args) {
    String str = " Java Programming ";

    // Chaining methods
    String result = str.trim().toUpperCase().substring(0, 4);
    System.out.println("Result: " + result);
  }
}
```

Output:

Result: JAVA

Advanced String Concepts and Methods

1. split(String regex)

- Splits the string into an array of substrings based on a specified delimiter or regular expression.

Example:

```
public class SplitExample {
  public static void main(String[] args) {
    String str = "Java,Python,C++";
    // Split by comma
    String[] languages = str.split(",");
    // Print each language
    for (String language : languages) {
      System.out.println(language);
    }
  }
}
```

Output for Example:

```
Output:
Java
Python
C++
```

Advanced String Concepts and Methods

2. concat(String str)

- Concatenates (joins) the specified string to the end of another string.

Example:

```
public class ConcatExample {
  public static void main(String[] args) {
    String str1 = "Hello";
    String str2 = " World";

    System.out.println("Concatenated String: " + str1.concat(str2));
  }
}
```

Output:

```
Concatenated String: Hello World
```

Advanced String Concepts and Methods

3. trim()

- Removes leading and trailing whitespaces from the string.

Example:

```
public class TrimExample {
  public static void main(String[] args) {
    String str = "  Java Programming  ";

    System.out.println("Before trim: [" + str + "]");
    System.out.println("After trim: [" + str.trim() + "]");
  }
}
```

Output:

```
Before trim: [  Java Programming  ]
After trim: [Java Programming]
```

Advanced String Concepts and Methods

4. compareTo(String anotherString)

- Compares two strings lexicographically. Returns:
  - 0 if strings are equal.
  - A positive value if the first string is lexicographically greater.
  - A negative value if the first string is lexicographically smaller.

Example:

```
public class CompareToExample {
  public static void main(String[] args) {
    String str1 = "Apple";
    String str2 = "Banana";

    System.out.println("Comparison result: " + str1.compareTo(str2));
  }
}
```

Output for Example:

Output:

Comparison result: -1 (because "Apple" comes before "Banana")

Advanced String Concepts and Methods

5. contains(String sequence)

- Checks if the string contains the specified sequence of characters.

Example:

```
public class ContainsExample {
  public static void main(String[] args) {
    String str = "Java Programming";

    System.out.println("Contains 'Java': " + str.contains("Java"));
    System.out.println("Contains 'Python': " + str.contains("Python"));
  }
}
```

Output:

```
Contains 'Java': true
Contains 'Python': false
```

Advanced String Concepts and Methods

6. String intern()

- Returns the interned string.
- It returns the canonical representation of string.
- It can be used to return string from memory if it is created by a new keyword.
- It creates an exact copy of the heap string object in the String Constant Pool.

Example for string intern method:

Example:

```
public class InternExample{
    public static void main(String[] args){
        String s1 = new String("JavaDeveloper");
        String s2 = "JavaDeveloper";
        String s3 = s1.intern();
        System.out.println(s1==s2);
        System.out.println(s2==s3);
 }
}
```

Output:

```
false
true
```

Advanced String Concepts and Methods

7. StringBuilder Class

- Unlike String, the StringBuilder class creates mutable (modifiable) string objects. It is efficient for frequent modifications.

Important Methods of StringBuilder:

1. **append(String str):** Adds a string at the end.
2. **insert(int offset, String str):** Inserts a string at the specified position.
3. **replace(int start, int end, String str):** Replaces characters within a range.
4. **reverse():** Reverses the string.

Example:

Example:

```
public class StringBuilderExample {
  public static void main(String[] args) {
    StringBuilder sb = new StringBuilder("Hello");

    // Append
    sb.append(" World");
    System.out.println("After append: " + sb);

    // Insert
    sb.insert(5, ",");
    System.out.println("After insert: " + sb);
```

Example for StringBuilder:

```
 // Replace
 sb.replace(6, 11, "Java");
 System.out.println("After replace: " + sb);

 // Reverse
 sb.reverse();
 System.out.println("After reverse: " + sb);
 }
}
```

Output:

```
After append: Hello World
After insert: Hello, World
After replace: Hello, Java
After reverse: avaJ ,olleH
```

Advanced String Concepts and Methods

8. StringBuffer (Thread-Safe Alternative to StringBuilder)

- Similar to StringBuilder, but synchronized (thread-safe) for multi-threaded environments. Use StringBuffer when working with threads.

Example:

```
public class StringBufferExample {
  public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("Java");

    sb.append(" Programming");
    System.out.println("StringBuffer: " + sb);
  }
}
```

Output:

```
StringBuffer: Java Programming
```

Combined Advanced Example

Example:

```
import java.util.Scanner;

public class AdvancedStringExample {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    // Input string
    System.out.print("Enter a string: ");
    String input = scanner.nextLine();

    // String manipulation
    StringBuilder sb = new StringBuilder(input.trim());
    sb.reverse(); // Reverse the input
```

Combined Advanced Example

```
        // Output results
        System.out.println("Original string: " + input);
        System.out.println("Reversed string: " + sb.toString());
        System.out.println("Uppercase string: " + input.toUpperCase());
        System.out.println("Does the string contain 'Java'? " + input.contains("Java"));
    }
}
```

Output:

```
Enter a string:  Java Programming
Original string:  Java Programming
Reversed string: gnimmargorP avaJ
Uppercase string:  JAVA PROGRAMMING
Does the string contain 'Java'? true
```

Arrays in Java

Definition:

- An array in Java is a data structure that can store multiple values of the same type in a single variable, using a fixed-size memory block.

Why Use Arrays?

- To store and manage collections of data efficiently.
- Useful when the number of elements is fixed.

Types of Arrays:

1. **Single-Dimensional Arrays:** Linear collection of elements.
2. **Multi-Dimensional Arrays:** Arrays of arrays, often used for matrices or grids.

Arrays in Java

Single-Dimensional Arrays

- A single-dimensional array is a collection of elements with a single index value.
- A single-dimensional array can have multiple columns but one row.

Declaration and Initialization:

```
dataType[] arrayName = new dataType[size]; // Declaration
arrayName[index] = value;                  // Initialization
```

Sample format:

```
String words[] = new String[3];
```

- Array of String, words, which can store three elements with the index of elements ranging from 0 to 2.

Single-Dimensional Arrays Example:

Example:

```
public class SingleDimensionalArray {
  public static void main(String[] args) {
    int[] numbers = new int[5]; // Declare and initialize array

    // Assign values
    numbers[0] = 10;
    numbers[1] = 20;
    numbers[2] = 30;
    numbers[3] = 40;
    numbers[4] = 50;
```

Single-Dimensional Arrays Example:

```
// Access and print array elements
for (int i = 0; i < numbers.length; i++) {
System.out.println("Element at index " + i + ": " + numbers[i]);
}
}
}
```

Output:

```
Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Element at index 3: 40
Element at index 4: 50
```

Arrays in Java

Multi-Dimensional Arrays

- An array having more than one dimension is called a multidimensional array.
- The commonly used multidimensional array is a two-dimensional array where we can have multiple rows and columns.

Declaration and Initialization:

```
dataType[][] arrayName = new dataType[rows][columns]; // Declaration
arrayName[row][column] = value;                       // Initialization
```

Sample format:

```
String[][] words = new String[4][2];
```

Multi-Dimensional Arrays Example:

Example:

```
public class MultiDimensionalArray {
  public static void main(String[] args) {
    int[][] matrix = new int[2][3]; // Declare a 2x3 array

    // Assign values
    matrix[0][0] = 1; matrix[0][1] = 2; matrix[0][2] = 3;
    matrix[1][0] = 4; matrix[1][1] = 5; matrix[1][2] = 6;
```

Multi-Dimensional Arrays Example:

```
// Access and print elements
      for (int i = 0; i < matrix.length; i++) {
         for (int j = 0; j < matrix[i].length; j++) {
            System.out.print(matrix[i][j] + " ");
         }
         System.out.println(); // Newline after each row
      }
   }
}
```

Output:

```
1 2 3
4 5 6
```

Array Methods and Operations

1. Length of an Array:

- The length property gives the number of elements in an array.

Example:

```
public class ArrayLength {
  public static void main(String[] args) {
    int[] numbers = {10, 20, 30, 40, 50};
    System.out.println("Length of the array: " + numbers.length);
  }
}
```

Output:

```
Length of the array: 5
```

Array Methods and Operations

2. Copying an Array (Using System.arraycopy)

Example:

```
public class ArrayCopy {
  public static void main(String[] args) {
    int[] source = {1, 2, 3, 4, 5};
    int[] destination = new int[5];

    // Copy array
    System.arraycopy(source, 0, destination, 0, source.length);
```

Array Methods and Operations

```
// Print copied array
 for (int num : destination) {
 System.out.print(num + " ");
 }
 }
}
```

Output:

1 2 3 4 5

Array Methods and Operations

3. Sorting an Array (Using Arrays.sort)

Example:

```
import java.util.Arrays;

public class ArraySort {
  public static void main(String[] args) {
    int[] numbers = {40, 10, 30, 20, 50};

    // Sort array
    Arrays.sort(numbers);
```

Array Methods and Operations

```
// Print sorted array
      System.out.println("Sorted Array: " + Arrays.toString(numbers));
  }
}
```

Output:

Sorted Array: [10, 20, 30, 40, 50]

Array Methods and Operations

4. Searching an Array (Using Arrays.binarySearch)

👉 **Note:** The array must be sorted before using binarySearch.

Example:

```
import java.util.Arrays;

public class ArraySearch {
  public static void main(String[] args) {
    int[] numbers = {10, 20, 30, 40, 50};
```

Array Methods and Operations

```
// Search for a number
 int index = Arrays.binarySearch(numbers, 30);
 System.out.println("Index of 30: " + index);
 }
}
```

Output:

```
Index of 30: 2
```

Array Methods and Operations

5. Using for-each Loop:

Used for iterating over arrays in a simplified way.

Example:

```
public class ForEachExample {
  public static void main(String[] args) {
    int[] numbers = {10, 20, 30, 40, 50};

    // Iterate using for-each
    for (int num : numbers) {
      System.out.println("Number: " + num);
    }
  }
}
```

Array Methods and Operations

Output:

Number: 10
Number: 20
Number: 30
Number: 40
Number: 50

Combined Example: Operations on Multi-Dimensional Arrays

Example:

```
import java.util.Arrays;

public class MultiArrayOperations {
  public static void main(String[] args) {
    int[][] matrix = {
      {3, 5, 7},
      {2, 4, 6},
      {1, 8, 9}
    };

    // Accessing specific elements
    System.out.println("Element at (1, 2): " + matrix[1][2]);
```

Combined Example: Operations on Multi-Dimensional Arrays

```
        // Calculating row-wise sum
        for (int i = 0; i < matrix.length; i++) {
            int rowSum = Arrays.stream(matrix[i]).sum();
            System.out.println("Sum of row " + (i + 1) + ": " + rowSum);
        }
    }
}
```

Output:

```
Element at (1, 2): 6
Sum of row 1: 15
Sum of row 2: 12
Sum of row 3: 18
```

KGiSL MicroCollege

Genius as a Service (GaaS)

THE COMPLETE CORE JAVA COURSE

PRESENTED BY,

KGM TECHNICAL TEAM

DAY-02 COMPLETED 🎉