Java Unit - 2

Loops : Working with for loop, while loop, do-while loop and for-each loop

Arrays and Enums : Fundamentals about Arrays, Multi-dimensional arrays, Array Access and Iterations, Using varargs, Enumerations

OOP Concepts : Basics of class and objects, Writing constructors and methods, Overloading methods and constructors, this keyword, initializer blocks

String Class : Constructors and methods of String and String Builder class

Loops

Iteration Statements in Java

Iteration statements, also known as loops, allow executing a block of code multiple times. This concept is fundamental in programming and helps automate repetitive tasks. Here, we'll cover the main types of loops in Java: `for`, `while`, and `do-while` loops.

1. `for` Loop

The `for` loop is used when the number of iterations is known beforehand. It's commonly used for counting iterations or iterating through arrays.

Syntax:

- **Initialization**: Sets the loop variable's initial value.
- **Condition**: Evaluated before each iteration; if `true`, the loop continues; if `false`, it stops.
- **Update**: Updates the loop variable after each iteration.

Example:

```
int a;

for (a=1;a<100;a++){

   System.out.println(a);

}
```

Explanation: This loop prints the iteration number from 0 to 4.

2. `while` Loop

The `while` loop is used when the number of iterations is not known beforehand and depends on a condition.

Syntax:

- The loop runs as long as the condition is `true`.

Example:

```
int a = 1;

boolean b = true;

while (b){       // while(true/false)

   System.out.println("While loop execute");

   System.out.println(a);

   if (a  == 20){

     b = false;

   }

   a++;

}
```

Explanation: This loop behaves similarly to the `for` loop but uses a different structure.

3. `do-while` Loop

The `do-while` loop is similar to the `while` loop, but it guarantees that the code block is **executed at least once**, even if the condition is `false` initially.

Syntax:

Example:

```
int a = 100;

do {

   System.out.println("do-while loop execute");

   System.out.println(a);

   a++; // a = 2

}while (a<1);
```

Explanation: The loop prints the iteration number from 0 to 4, similar to the previous examples.

4. Enhanced `for` Loop (For-Each Loop)

The enhanced `for` loop is used primarily for iterating through arrays or collections.

Syntax:

Example:

```
int[] arr = {10, 20, 30, 40}; // length = 4

for(int arrVal: arr){

   System.out.println(arrVal);

}
```

Explanation: This loop iterates through each element in the array `numbers`.

5. Common Pitfalls and Best Practices

- **Infinite Loops**: Ensure the loop's condition will eventually become `false`; otherwise, the loop may run indefinitely.
- **Nested Loops**: Be cautious with nested loops as they can lead to increased time complexity.

Example: Sum of Numbers

Jump Statements

Jump statements in Java are used to transfer control to different parts of the code. These statements can alter the flow of control in loops, switch statements, or other parts of the program. Java provides several types of jump statements: `break`, `continue`, and `return`.

1. `break` Statement

The `break` statement is used to **exit a loop or switch statement prematurely**. It helps in stopping the execution of the loop when a certain condition is met.

Syntax: `break;`

Use Cases:

- Exiting a loop when a specific condition is met.
- Exiting a `switch` statement after executing a particular case block.

Example in a Loop:

```
public static void main(String[] args) {

   for (int i=0;i<10;i++){

       if (i==5){

           break;

       }

       System.out.println("Iteration: " + i);

   }
```

```
}
```

Explanation: This loop prints numbers from 0 to 4 and exits when `i` reaches 5.

Example in a Switch Statement:

```
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; // Exits after matching the case

   default:

        System.out.println("Another day");

}
```

Explanation: The `break` statement stops further checking of cases once a match is found.

2. `continue` Statement

The `continue` statement is used to **skip the current iteration** of a loop and proceed with the next iteration. Unlike `break`, it does not terminate the loop but skips the remaining code in the current iteration.

Syntax: `continue;`

Example:

```java
public static void main(String[] args) {
   for (int i=0;i<10;i++){
       if (i % 2 == 0){
           continue; // skip the rest of the loop for even numbers
       }
       System.out.println("Odd number: " + i);
   }
}
```

Explanation: This loop skips the even numbers and only prints odd numbers.

3. `return` Statement

The `return` statement is used to **exit from the current method** and optionally return a value to the caller. It's commonly used to exit a method when a specific condition is met.

Syntax:

```
return;       // Exits the method without returning a value
return value; // Exits the method and returns a value
```

Example in a Method:

```java
public static void main(String[] args) {
   System.out.println("Result: "+ checkNumber(5));
}
static String checkNumber(int num) {
   if (num > 0) {
       return "Positive"; // Exits the method if the condition is true
   }
   return "Non-Positive";
```

```
}
```

Explanation: The `return` statement exits the `checkNumber` method and provides the result based on the condition.

4. Labeled Break and Continue

Java also allows labeled `break` and `continue` statements to provide more control over nested loops.

Syntax:

```
label: // Defines a label
       {        // Code block
           break label; // Breaks out of the labeled block
           continue label; // Skips to the next iteration of the labeled loop
       }
```

Example with Labeled `break`:

```
public static void main(String[] args) {
   outer: // Label for the outer loop
   for (int i = 2; i < 3; i++) {
       for (int j = 0; j < 3; j++) {
           if (i == j) {
               break outer; // Breaks out of the outer loop
           }
           System.out.println("i = "+i+", j "+ j);
       }
   }
}
```

Explanation: The labeled `break` exits both loops when `i == j`.

Example with Labeled `continue`:

```
public static void main(String[] args) {

   outer: // Label for the outer loop

   for (int i = 2; i < 3; i++) {

       for (int j = 0; j < 3; j++) {

           if (i == j) {

               continue outer; // Skips to the next iteration of the outer
loop

           }

           System.out.println("i = "+i+", j "+ j);

       }

   }

}
```

Explanation: The labeled `continue` skips the current iteration of the outer loop when `i == j`.

Arrays and Enums

Arrays in Java

An array is a **collection of elements of the same type stored in contiguous memory locations**. Arrays allow you to store multiple values of a similar type in a single variable, making it easy to manage and manipulate large sets of data.

Key Features of Arrays

1. **Fixed Size:** The size of an array is fixed and defined when it is created. This means you cannot change the size of an array once it's set.

2. **Homogeneous Elements:** Arrays can store only elements of the same data type (e.g., all integers or all strings).
3. **Zero-Based Indexing:** Array indexing starts at 0, so the first element is accessed with index 0, the second with index 1, and so on.
4. **Direct Access:** You can access elements directly using their index, which makes arrays fast for data retrieval.

Types of Arrays in Java

1. **Single-Dimensional Arrays:**
    - A list of elements arranged in a single row.

```
int[] numbers = new int[];

numbers[0] = 10;

numbers[1] = 20;
```

Multi-Dimensional Arrays

- **Definition**: Multi-dimensional arrays are arrays of arrays, allowing storage of data in a matrix or grid-like structure.
- **Key Features**:
    - Often used for representing data in tables or matrix.
    - Access elements using multiple indices.

Syntax:
// Declaration

dataType[][] arrayName;

// Instantiation

arrayName = new dataType[rows][columns];

// Combined

- dataType[][] arrayName = new dataType[rows][columns];

Example:

```
int[][] matrix = new int[2][3];

       matrix[0][0] = 1;

       matrix[0][1] = 2;

       matrix[0][2] = 3;

       matrix[1][0] = 4;

       matrix[1][1] = 5;

       matrix[1][2] = 6;



// Output each element

       for (int i = 0; i < 2; i++) {

           for (int j = 0; j < 3; j++) {

               System.out.print(matrix[i][j] + " ");

           }

           System.out.println(); // Move to the next row

       }
```

Jagged Arrays: Arrays with rows of varying lengths.

```
int[][] jaggedArray = new int[2][];

       jaggedArray[0] = new int[3]; // First row has 3 columns

       jaggedArray[1] = new int[2]; // Second row has 2 columns

       jaggedArray[0][0] = 1;

       jaggedArray[0][1] = 2;

       jaggedArray[0][2] = 3;
```

```
        jaggedArray[1][0] = 4;

        jaggedArray[1][1] = 5;

// Output each element

        for (int i = 0; i < jaggedArray.length; i++) {

            for (int j = 0; j < jaggedArray[i].length; j++) {

                System.out.print(jaggedArray[i][j] + " ");

            }

            System.out.println(); // Move to the next row

        }
```

Applications:

- Representing data in tables (e.g., seating charts, scores).
- Storing images or grids in graphical applications.

Declaration, Instantiation, and Initialization of Arrays

1. **Declaration:** Specifies the array's type and name.
    - Syntax: `dataType[] arrayName;`
    - Example: `int[] numbers;`
2. **Instantiation:** Allocates memory for the array using the `new` keyword.
    - Syntax: `arrayName = new dataType[size];`
    - Example: `numbers = new int[5];`
3. **Initialization:** Assigns values to the array elements.
    - Combined syntax: `dataType[] arrayName = new dataType[]{value1, value2, ...};`
    - Example: `int[] numbers = {1, 2, 3, 4, 5};`

Array Access, Iterations and Modifying

- **Accessing Elements**:

- Using the index to access or modify values.
- Syntax: `arrayName[index]`

  Example:
  int[] arr = {1, 2, 3};

- System.out.println(arr[0]); // Output: 1

- **Iterating Through Arrays**:

Using a **for loop**:

```
for (int i = 0; i < arr.length; i++) {

   System.out.println(arr[i]);

}
```

Using a **for-each loop**:

```
for(int arrEle: arr){

   System.out.println(arrEle);

}
```

Modifying Elements: Assign a new value using the index.

- Syntax: `arrayName[index] = newValue;`
- Example: `numbers[1] = 10; // Changes the second element to 10`

The `Arrays` Class in Java

The `Arrays` class in Java, part of the `java.util` package, provides various utility methods for manipulating arrays, such as sorting, searching, comparing, and converting arrays to strings. This class makes working with arrays more efficient and less error-prone.

1. Overview of the `Arrays` Class

The `Arrays` class is a final class, meaning it cannot be extended. It contains static methods, so you don't need to create an instance of `Arrays` to use its methods. The primary operations include:

- Sorting arrays
- Searching for elements
- Comparing arrays
- Filling arrays with values
- Converting arrays to strings

2. Commonly Used Methods in the `Arrays` Class

a. `sort()` Method

The `sort()` method sorts the elements of an array in ascending order. It works on both primitive and object arrays.

Example:

```
int[] numbers = {5, 3, 8, 1, 2};

Arrays.sort(numbers);

System.out.println(Arrays.toString(numbers)); // Output: [1, 2, 3, 5, 8]
```

Explanation: The `sort()` method arranges the array elements in ascending order.

b. `binarySearch()` Method

The `binarySearch()` method searches for a specified element in a sorted array and returns its index. If the element is not found, it returns a negative value.

Example:

```
int[] numbers = {1, 2, 3, 5, 8};

int index = Arrays.binarySearch(numbers, 3);

System.out.println("Index of 3: "+ index); // Output: Index of 3: 2
```

Explanation: The method searches for the element `3` and returns its index.

c. `equals()` Method

The `equals()` method compares two arrays to check if they are equal. It returns `true` if both arrays have the same length and elements in the same order.

Example:

```
int[] arr1 = {1, 2, 3};

int[] arr2 = {1, 2, 3};

System.out.println(Arrays.equals(arr1, arr2)); // Output: true
```

Explanation: The method checks if the arrays `arr1` and `arr2` are equal.

e. `toString()` Method

The `toString()` method converts an array into a string representation, making it easy to print the contents of the array.

Example:

```
int[] numbers = {1, 2, 3, 4, 5};

System.out.println(Arrays.toString(numbers)); // Output: [1, 2,
3, 4, 5]
```

Explanation: The method converts the `numbers` array into a readable string format.

f. `copyOf()` Method

The `copyOf()` method copies an array or a portion of it into a new array.

Example:

```
int[] numbers = {1, 2, 3, 4, 5};

   int[] copy = Arrays.copyOf(numbers, 3);
```

```
    System.out.println(Arrays.toString(copy)); // Output: [1, 2,
3]

}
```

Explanation: The method creates a copy of the first three elements of the `numbers` array.

g. `copyOfRange()` Method

The `copyOfRange()` method copies a range of elements from an array into a new array.

Example:

```
int[] numbers = {1, 2, 3, 4, 5};

    int[] subArray = Arrays.copyOfRange(numbers, 1, 4);

    System.out.println(Arrays.toString(subArray)); // Output: [2,
3, 4]
```

Explanation: The method copies elements from index 1 to 3 from the `numbers` array.

3. Practical Example: Working with Arrays

Here's a practical example that combines multiple methods of the `Arrays` class:

```
int[] data = {10, 5, 3, 8, 2};

// Sorting the array

        Arrays.sort(data);

        System.out.println("Sorted Array: "+
Arrays.toString(data));

// Searching for an element

        int index = Arrays.binarySearch(data, 8);

        System.out.println("Index of 8: "+ index);
```

```
// Filling an array

                Arrays.fill(data, 1);

        System.out.println("Filled Array: "+
Arrays.toString(data));

// Comparing arrays

        int[] dataCopy = {1, 1, 1, 1, 1};

        System.out.println("Arrays Equal: "+ Arrays.equals(data,
dataCopy));
```

```
// Declaration, instantiation, and initialization

        int[] scores = {85, 90, 78, 88, 76};

// Accessing elements

        System.out.println("First score:" + scores[0]);

// Modifying an element

        scores [2] = 80;

// Change third score to 80

        System.out.println("Modified third score: "+ scores[2]);

// Finding the length of the array

                System.out.println("Total scores: " +
scores.length);

// Sorting the array

        Arrays.sort(scores);
```

```
        System.out.println("Sorted scores: "+
Arrays.toString(scores));

// Traversing using enhanced for loop

                System.out.print("All scores: ");

        for (int score :scores) {

            System.out.print(score +" ");

        }
```

Using Varargs

- **Definition**: Varargs (variable-length arguments) allow you to pass an arbitrary number of arguments to a method. This is particularly useful when you don't know in advance how many arguments will be passed.
- **Key Features**:
    - Varargs is represented by three dots (`...`) in the method parameter.
    - Treated as an array within the method, enabling you to perform array-like operations.
    - It simplifies method calls where varying numbers of inputs are required.

Syntax:

```
public static void methodName(dataType... variableName) {

   // Code to process variable-length arguments

}
```

Example:

```
public static void main(String[] args) {

   printNumbers(1, 2, 3, 4); // Output: 1 2 3 4

}

public static void printNumbers(int... numbers) {

   for (int num : numbers) {
```

```
        System.out.println(num);

    }

}
```

Advanced Example with Mixed Parameters: Varargs can be combined with other parameters, but it must always be the last parameter:

```
public static void main(String[] args) {
```

```


printDetails("Numbers are:", 10, 20, 30); // Output: Numbers
are: 10 20 30

    }

    public static void printDetails(String message, int...
numbers) {

        System.out.println(message);

        for (int num : numbers) {

            System.out.print(num + " ");

        }

        System.out.println();

    }
```

How It Works Internally: Varargs is treated as an array within the method. For example:

```
    public static void main(String[] args) {
```

```


        demoVarargs(5, 10, 15); // Output: Total numbers: 3,
First number: 5

    }
```

```
    public static void demoVarargs(int... nums) {

        System.out.println("Total numbers: " + nums.length);

        System.out.println("First number: " + nums[0]);

    }
```

- **Key Points to Remember**:
    - Varargs must be the last parameter in the method signature.
    - Only one varargs parameter is allowed per method.
- **Use Cases**:
    - Creating utility methods that can handle multiple inputs, such as logging or mathematical operations.
    - Building flexible APIs or libraries where the number of arguments is unpredictable.

Introduction to Enumerations

Enumerations, or `enum` in Java, are a special data type that enable a variable to be a set of predefined constants. Enumerations are often used when a variable needs to represent a fixed set of values, such as days of the week, directions, or states of an object. Enumerations make code more readable and less error-prone compared to using constant variables.

Declaring an Enum

An enumeration is declared using the `enum` keyword. Here's a basic example:

```
public enum Day {

    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
SUNDAY;

}
```

In this example, `Day` is an enumeration with seven constants: `MONDAY`, `TUESDAY`, etc. Each constant is implicitly a `public static final`.

Using an Enum

```
public static void main(String[] args) {

        Day today = Day.MONDAY;

        switch (today) {

            case MONDAY:

                System.out.println("Start of the work week!");

                break;

            case FRIDAY:

                System.out.println("Almost the weekend!");

                break;

            case SATURDAY:

            case SUNDAY:

                System.out.println("Weekend!");

                break;

            default:

                System.out.println("Midweek day.");

        }

    }

}
```

Enum Methods

Enums in Java come with some built-in methods:

`values()`: Returns an array of all enum constants.
Example:

```
for (Day day : Day.values()) {
```

```
    System.out.println(day);

}
```

`ordinal()`: Returns the position of the enum constant in the declaration (0-based).
Example:

```
System.out.println(Day.MONDAY.ordinal()); // Outputs: 0
```

- **`name()`**: Returns the name of the enum constant as a string.
  Example:
- `System.out.println(Day.FRIDAY.name()); // Outputs: FRIDAY`
-
- **`toString()`**: Converts the enum constant to a string. By default, it's the same as `name()` but can be overridden.

Advantages of Using Enums

1. **Type Safety**: Prevents invalid values.
2. **Readability**: Improves code readability by representing constant values in a descriptive way.
3. **Rich Functionality**: Can include fields, methods, and constructors.
4. **Integration with Switch**: Works seamlessly with switch statements.

Enum in Java's Collections

Enums can be used in collections such as `EnumSet` and `EnumMap` for efficient storage and retrieval.

EnumSet: A specialized `Set` implementation for enums.

```
EnumSet<Day> weekend = EnumSet.of(Day.SATURDAY, Day.SUNDAY);

System.out.println(weekend);
```

EnumMap: A specialized `Map` implementation for enums.
`EnumMap<Day, String> dayType = new EnumMap<>(Day.class);`

```
dayType.put(Day.MONDAY, "Weekday");
```

```
dayType.put(Day.SATURDAY, "Weekend");

System.out.println(dayType);
```

OOPs Concepts

Object-Oriented Programming (OOP) Concepts

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects, which contain data in the form of fields (attributes) and code in the form of methods (functions). The key principles of OOP include Encapsulation, Inheritance, Polymorphism, and Abstraction.

Procedural Programming	OOPS
Program is divided into parts called functions.	Program is divided into objects.
Doesn't provide a way to hide data, gives importance to function, and data moves freely.	Objects provide data hiding, giving importance to data.
Overloading is not possible.	Overloading is possible.
Inheritance is not possible.	Inheritance is possible.
Code reusability does not exist.	Code reusability is present.
Eg.: Pascal, C, FORTRAN, etc.	**Eg.**: Java, C#, Python, C++, etc.

What is a Class?

- A **class** is a blueprint for creating objects. It defines a set of properties (fields/attributes) and behaviors (methods/functions) that the objects created from the class can have.

- **Analogy**: Think of a class as a template for making something, like a blueprint for a house. The blueprint itself isn't the house; it's the design you use to build a house.

```
class ClassName {

//    Fields (Attributes)

//    Methods (functions)

}
```

Class Fundamentals

- **Components of a Class:**
  - **Attributes (Fields):** Variables that store data about the class.
  - **Methods:** Functions that define the behavior of the class.
  - **Constructors:** Special methods used to initialize objects.
  - **Access Modifiers:** Keywords that define the visibility of class members (e.g., `public`, `private`, `protected`).
- To create an object, a class is required.
- So, class provides the template or blueprint from which an object can be created.
- From a class, we can create multiple objects.
- To create a class, use the keyword `class`.

Declaring a Class in Java

```
public class Car {

   // Fields

   String color;

   String model;

   int year;

   // Constructor

   public Car(String color, String model, int year) {

       this.color = color;

       this.model = model;
```

```
        this.year = year;

    }

    // Method

    public void displayInfo() {

        System.out.println("Car Model: "+ model);

        System.out.println("Car Color: "+ color);

        System.out.println("Car Year: "+ year);

    }

}
```

Types of Classes

1. **Concrete Classes:** A concrete class is a class that can be instantiated (i.e., objects can be created from it). It provides complete implementations of its methods.
2. **Abstract Classes:** An abstract class cannot be instantiated. It can contain abstract methods (methods without a body) and concrete methods (methods with a body). Subclasses must implement the abstract methods.
3. **Nested Classes:** A class defined within another class. Nested classes can be categorized into:
    a. **Static Nested Classes:** These do not have access to instance variables and methods of the outer class. They are associated with the class itself rather than an instance of the class.
    b. **Inner Classes:** Non-static nested classes that have access to instance variables and methods of the outer class.
4. **Anonymous Classes:** A type of inner class that does not have a name. They are used to instantiate classes with a single use, often for implementing interfaces or extending classes with a concise syntax.

Objects

- **Object** has 2 things:
    - Properties or State
    - Behavior or Function

For Example:

- **Dog** is an object because:
  - Properties like: Age, Colour, Breed, etc.
  - Behavior like: Bark, Sleep, Eat, etc.
- **Car** is an object because it has:
  - Properties like: Colour, Type, Brand, Weight, etc.
  - Behavior like: Apply brake, Drive, Increase speed, etc.

Creating and Using Objects: Using the new Keyword

```
ClassName ObjectName = new ClassName();
```

Method Declaration in Java

- **Definition:** A method is a **block of code** that performs a specific task and can be called upon multiple times. It helps break down programs into smaller, manageable parts.
- **Syntax:**

```
returnType methodName(parameterList){

   //    method body

}
```

```
class Calculator {

   // Method to add two numbers

   public int add(int a, int b) {

       return a + b;

   }

}

public class CalMain {

   public static void main(String[] args) {

       Calculator calc = new Calculator();
```

```
        int result = calc.add(5, 3); // Method call

        System.out.println("Sum: "+ result); // Output: Sum:
8

    }

}
```

Rules for Method Naming:

1. **Valid Identifier:** Method names must be valid Java identifiers (letters, digits, underscores, dollar signs) and cannot start with a digit.
2. **Case-Sensitive:** Java method names are case-sensitive (e.g., `myMethod()` and `mymethod()` are different).
3. **Camel Case:** Follow camel case convention (e.g., `calculateSum()`, `getData()`), starting with a lowercase letter.
4. **Descriptive:** Method names should describe the action they perform, typically verbs or verb phrases.
5. **No Reserved Keywords:** Method names cannot be Java reserved keywords like `class`, `static`, `int`.

Rules for Method Declaration:

1. **Method Signature:** Includes the method name and parameter list; the return type and access modifiers are not part of the signature.
2. **Return Type:** Specifies the type of value the method returns; use `void` if no return value.
3. **Method Name:** Must follow Java naming rules and conventions (e.g., camelCase, no reserved keywords).
4. **Parameter List:** Defines input parameters with types and names, separated by commas; use empty parentheses if no parameters.
5. **Access Modifiers:** Control method visibility (`public`, `private`, `protected`, or default package-private).
6. **Optional Modifiers:** Includes `static`, `final`, `abstract`, or `synchronized` for specific behavior.
7. **Method Body:** Contains the code to be executed; must include a `return` statement if the method has a return type.

Method Overloading

- **Definition:** Method overloading occurs when multiple methods in the same class have the same name but different parameter lists (number, type, or order of parameters). This allows methods to handle different types or numbers of arguments.

```
class Calculator1 {

    // Method to add two integers

    public int add(int a, int b) {

        return a + b;

    }

    // Method to add three integers (Overloaded method)

    public int add(int a, int b, int c) {

        return a + b + c;

    }

    // Method to add two doubles (Overloaded method)

    public double add(double a, double b) {

        return a + b;

    }

}

public class MethodOverloading {

    public static void main(String[] args) {

        Calculator1 calc = new Calculator1();

        System.out.println("Sum (2 ints): "+ calc.add(5,
3)); // Output: 8

        System.out.println("Sum (3 ints): " + calc.add(5, 3,
2)); // Output: 10

        System.out.println("Sum (2 doubles): "+
calc.add(2.5, 3.7)); // Output: 6.2
```

```
    }

}
```

Using Objects as Parameters

- **Definition:** In Java, objects can be passed to methods as parameters. This is useful for passing complex data structures like custom objects.

```
class Car {

    String model;

    int year;

    public Car(String model, int year) {

        this.model = model;

        this.year = year;

    }

    public void displayInfo() {

        System.out.println("Model: "+ model + "Year: "+
year);

    }

}

class Garage {

    // Method that accepts a Car object as a parameter

    public void parkCar(Car car) {

        System.out.println("Parking car...");

        car.displayInfo();

    }
```

```
}

public class ObjAsParameters {

   public static void main(String[] args) {

        Car myCar = new Car("Toyota", 2022);

        Garage myGarage = new Garage();

// Passing an object as a parameter to the method

        myGarage.parkCar(myCar); // Output: Parking
car... Model: Toyota, Year: 2022

    }

}
```

Constructors in Java

Definition:
Constructors are **special methods** in Java used to initialize objects. They are **called when an instance of a class is created**. Constructors have the same name as the class and no return type.

Key Points About Constructors:

1. **Name:** The constructor must have the same name as the class.
2. **No Return Type:** Constructors do not return any value, not even `void`.
3. **Automatic Call:** A constructor is automatically called when an object of the class is created.
4. **Types of Constructors:** There are two main types of constructors in Java:
    - **Default Constructor**
    - **Parameterized Constructor**

1. Default Constructor:

- A default constructor is a no-argument constructor provided by Java if no other constructors are defined in the class.

- If a class has any constructor, the default constructor is not provided by Java automatically.

```
public class Student {

    String name;

    // Default Constructor

    public Student() {

        name = "Unknown"; // Setting default value

    }

    public static void main(String[] args) {

        Student student = new Student();

// Default constructor is called

        System.out.println(student.name); // Output: Unknown

    }

}
```

2. Parameterized Constructor:

- A parameterized constructor accepts arguments that allow you to set initial values for the object's properties when it is created.
- It allows for more flexibility when creating objects.

```
public class SecStudent {

    String name;

    // Parameterized Constructor

    public SecStudent(String studentName) {

        name = studentName;

    }
```

```
    public static void main(String[] args) {
        SecStudent student1 = new SecStudent("Alice");
// Parameterized constructor is called
        SecStudent student2 = new SecStudent("Bob");
        System.out.println(student1.name); // Output: Alice
        System.out.println(student2.name); // Output: Bob
    }
}
```

How Constructors Work:

- When you create an object using the `new` keyword, the appropriate constructor is called to initialize the object.

Overloading Constructors:

- Just like methods, constructors can be overloaded by defining multiple constructors with different parameter lists.
- This allows the creation of objects with different initialization options.

```
class ThirdStudent {
    String name;
    int age;
    // Default Constructor
    public ThirdStudent() {
        name = "Unknown";
        age = 0;
    }
    // Parameterized Constructor
```

```
    public ThirdStudent(String studentName) {
        name = studentName;
        age = 8;
    }
    // Another Parameterized Constructor
    public ThirdStudent(String studentName, int studentAge)
{
        name = studentName;
        age = studentAge;
    }
    public static void main(String[] args) {
        ThirdStudent s1 = new ThirdStudent();
        ThirdStudent s2 = new ThirdStudent("Mohit");
        ThirdStudent s3 = new ThirdStudent("Ashish", 20);
        System.out.println(s1.name + " " + s1.age);
        System.out.println(s2.name + " " + s2.age);
        System.out.println(s3.name + " " + s3.age);
    }
}
// Default constructor
// Single-parameter constructor
// Two-parameter constructor
// Output: Unknown 0
// Output: Mohit 8
```

```
// Output: Ashish 20
```

Constructor Chaining:

- **Constructor chaining** occurs when one constructor calls another constructor within the same class. This can be done using the `this()` keyword.

```
public class AnotherStudent {

    String name;

    int age;

    // Default Constructor

    public AnotherStudent() {

        this("Unknown", 0); // Calls the two-parameter
constructor

    }

    // Parameterized Constructor

    public AnotherStudent(String studentName, int
studentAge) {

        name = studentName;

        age = studentAge;

    }

    public static void main(String[] args) {

        Student s1 = new Student();
// Calls the default constructor

        System.out.println(s1.name + + s1.age);
// Output: Unknown 0

    }
```

```
}
```

Key Points:

1. **Constructors Cannot be `final`, `static`, or `abstract`.**
2. **Constructors in Inheritance:** In inheritance, the child class constructor calls the parent class constructor using the `super()` keyword, either implicitly or explicitly.
3. **Constructor Overloading vs Method Overloading**

`this` Keyword: This keyword in Java is a reference variable that **refers to the current object** within an instance method or constructor. It is primarily used to **differentiate between instance variables and parameters** or to invoke other methods and constructors within the same class.

1. **Refers to current object's fields:**
   Used to differentiate between instance variables and parameters with the same name.
2. **Invokes current class methods:**
   Used to call one method from another method within the same class.
3. **Invokes current class constructors:**
   Used for constructor chaining within the same class to call another constructor.
4. **Returns the current object:**
   Used to return the current object for method chaining.

```
public class StudentLPU {

   String name;

   int age;

   // Constructor with 'this' to differentiate between
instance variables and parameters

   public StudentLPU(String name, int age) {

       this.name = name; // Refers to current object's
fields

       this.age = age;

   }
```

```
    // Method to display student details
    public void display() {
        System.out.println("Name:" + this.name + ", Age: " +
this.age);
    }
    // Constructor chaining using 'this()'
    public StudentLPU() {
        this("Unknown", 18); // Invokes current class
constructor
    }
public StudentLPU(String name, int age) {
        this("Unknown", 18);    }

    // Method chaining using 'this' to return current object
    public StudentLPU setName(String name) {
        this.name = name;
        return this; // Returns the current object
    }

    public static void main(String[] args) {
        StudentLPU student1 = new StudentLPU("Alice", 20);
// Using parameterized constructor
```

```
        student1.display();

        StudentLPU student2 = new StudentLPU(); // Using
constructor chaining

        student2.display();

        student2.setName("Bob").display(); // Method
chaining with 'this'

    }

}
```

When to Use `this`:

1. **Resolving Ambiguity:** The primary use of `this` is to resolve naming conflicts between instance variables and parameters.
2. **Invoking Methods and Constructors:** `this` can be used to call methods or constructors within the same class.
3. **Returning Objects:** You can return the current object using `this` to facilitate method chaining.

Initializer Blocks

Initializer blocks are used to initialize instance variables before the constructor executes. They help in executing common setup code before an object is fully initialized.

Types of Initializer Blocks:

1. **Instance Initializer Block** – Runs every time an object is created and is executed before the constructor.
2. **Static Initializer Block** – Runs once when the class is loaded, typically used for static variable initialization.

Example of Instance Initializer Blocks:

```
public class InitializerBlocks {
    String name;
```

```
    int age;
    // Instance Initializer Block
    {
        System.out.println("Instance Initializer Block Executed - 
Preparing Student Object");
    }
    // Constructor
    InitializerBlocks(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public static void main(String[] args) {

    }
}
```

Example of Static Initializer Block:

```
static {
    System.out.println("Static Block Executed - Initializing 
University Data");
}
```

What is a String?

In Java, a **String** is a sequence of characters used to store and manipulate text. The `String` class is a part of `java.lang` package and is one of the most commonly used classes in Java.

Characteristics of String in Java

1. **Immutable**: Once a `String` object is created, its value cannot be changed.
2. **Stored in the String Pool**: Java optimizes memory usage by storing string literals in a special memory area called the **String Pool**.
3. **Implemented as an Array of Characters**: Internally, a `String` is implemented using a **character array (`char[]`)**.
4. **Provides Built-in Methods**: The `String` class provides multiple methods for string manipulation, like `length()`, `concat()`, `toUpperCase()`, etc.

String Class Constructors

Java provides multiple constructors to create `String` objects:

Default Constructor (Not explicitly provided in `String` class)

String s = new String(); // Creates an empty string

1. **String from Literal**

```
String str = "Hello";
```

2. **String from Another String**

```
String str = "Hello";

String str3 = new String(str);
```

3. **String from Character Array**

```
char[] chArr = {'J', 'a', 'v', 'a'};

String str4 = new String(chArr);

System.out.println(str4);
```

4. **String from Byte Array**

```java
byte[] bArr = {65, 66, 68};

String str5 = new String(bArr);

System.out.println(str5);
```

Important Methods of String Class

Below are some commonly used methods in the `String` class:

1. String Length

2. String Concatenation

3. String Comparison

- **equals()** (Case-sensitive)
- **equalsIgnoreCase()** (Case-insensitive)
- **compareTo()** (Lexicographical comparison)
-

4. Character Extraction

- **charAt()**

5. Searching in Strings

- **indexOf()**
- **lastIndexOf()**

6. Substring Extraction

7. String Case Conversion

8. String Trimming

9. Replacing Characters

10. Splitting a String

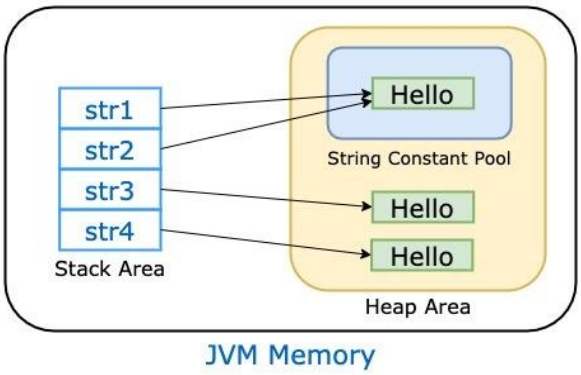
How String is Stored in Memory

To understand how `String` is stored in memory, we need to explore **Heap Memory** and **String Constant Pool (SCP)**.

Memory Areas in Java

Java divides memory into different areas:

1. **Heap Memory** (Stores objects created using `new` keyword)
2. **Stack Memory** (Stores local variables and references)
3. **Method Area** (Stores class metadata, static variables, and the constant pool)
4. **String Constant Pool (SCP)** (Stores string literals for memory optimization)



1. String Constant Pool (SCP)

- The **String Constant Pool (SCP)** is a part of the heap memory where Java stores **string literals**.
- When a string is created using **double quotes (`""`)**, it is stored in the SCP if it is not already present.
- If a string already exists in the SCP, the reference to the existing object is returned instead of creating a new one.

Example of String Storage in SCP

```
String s1 = "Hello";  // Stored in SCP
String s2 = "Hello";  // Reuses "Hello" from SCP
System.out.println(s1 == s2); // true (same reference)
```

Explanation:

- `"Hello"` is created in the SCP.
- `s1` and `s2` refer to the same object.
- Since they have the same reference, `s1 == s2` returns **true**.

---

2. String in Heap Memory

When a `String` is created using the `new` keyword, it is stored in the **Heap Memory** instead of SCP.

Example of String in Heap Memory

```
String s1 = new String("Hello");  // New object in Heap Memory
String s2 = new String("Hello");  // Another new object in Heap
Memory
System.out.println(s1 == s2); // false (different references)
```

Explanation:

- Even though both `s1` and `s2` have the same value `"Hello"`, they are separate objects in heap memory.
- Since they have different references, `s1 == s2` returns **false**.

How Java Stores Strings in Memory

Scenario 1: Creating String Using String Literals

```
String s1 = "Java";
String s2 = "Java";
```

- The `"Java"` string is stored in the **String Pool**.
- `s1` and `s2` will point to the **same memory location** in the String Pool.

Memory Representation:

```
String Constant Pool (SCP):
---------------------------
"Java"   <--- s1, s2 (both refer to the same object)
```

Scenario 2: Creating String Using `new` Keyword

```
String s3 = new String("Java");
String s4 = new String("Java");
```

- Two new objects are created **in the heap memory**.
- Even though `"Java"` already exists in the String Pool, new memory is allocated in the heap.

Memory Representation:

```
Heap Memory:
--------------------------
  "Java"   <--- s3 (new object)
  "Java"   <--- s4 (new object)

String Constant Pool (SCP):
--------------------------
"Java"   <--- (existing from previous literals)
```

Here, `s3 == s4` will return **false** because they refer to different objects.

String Interning in Java

Java provides the `intern()` method to manually store a string in the **String Pool**.

```
String s1 = new String("Hello");
String s2 = s1.intern(); // Places "Hello" in SCP if not present

String s3 = "Hello"; // Refers to SCP entry

System.out.println(s2 == s3); // true (both point to SCP)
System.out.println(s1 == s3); // false (s1 is still in heap)
```

Key Points:

- `s1` is in heap memory.
- `s2 = s1.intern();` places `"Hello"` in SCP (if not already present).
- `s3` refers to the existing SCP string.
- `s2 == s3` is **true**, but `s1 == s3` is **false**.

Why is String Immutable in Java?

The `String` class is **immutable** for several reasons:

1. **Memory Efficiency**
    - If `String` were mutable, modifying one variable would change all references.
    - This ensures **String Pool optimization**.
2. **Thread Safety**
    - Strings are shared across multiple threads.
    - Immutability makes them **safe for multi-threading** without explicit synchronization.
3. **Security**
    - Used in **passwords, database connections, and networking**.
    - Prevents accidental modification of sensitive data.
4. **Hashcode Caching**
    - The hash code of a `String` is **computed only once** and stored.
    - Helps in fast lookups in HashMaps and Sets.

When to Use `String`?

✔ If the string **won't change** frequently.
✔ If **memory efficiency** is important due to the SCP mechanism.
✔ When working with **constants**.

StringBuilder (Mutable, Not Thread-Safe)

`StringBuilder` is a **mutable** sequence of characters, meaning it allows modifications **without creating new objects**, leading to better performance.

Example:

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");
System.out.println(sb);  // Output: Hello World
```

Unlike `String`, `StringBuilder` **modifies the existing object** instead of creating a new one.

StringBuilder Constructors

Default Constructor

StringBuilder sb = new StringBuilder();

1. **String from Existing String**

   StringBuilder sb = new StringBuilder("Hello");
2. **StringBuilder with Capacity**

   StringBuilder sb = new StringBuilder(50); // Allocates space for 50 characters

---

Important Methods of StringBuilder

Since `StringBuilder` is mutable, methods modify the same object instead of creating a new one.

1. Append

2. Insert

3. Replace

4. Delete

5. Reverse

6. Get Length and Capacity

Performance (Faster than String)

- Since `StringBuilder` does **not create new objects**, it is **faster** than `String` in scenarios where frequent modifications occur.

When to Use `StringBuilder`?

✔ If the string **changes frequently** (e.g., in loops, dynamic operations).
✔ When **performance is critical** and **thread-safety is not required**.

---

3. StringBuffer (Mutable, Thread-Safe)

`StringBuffer` is similar to `StringBuilder`, but it is **thread-safe** because its methods are **synchronized**.

Example:

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World");
System.out.println(sb);  // Output: Hello World
```

- The behavior is the same as `StringBuilder`, but `StringBuffer` ensures **safe execution in multi-threaded environments**.

Performance (Slower than StringBuilder)

Since `StringBuffer` uses **synchronization**, it is **slower** than `StringBuilder` in **single-threaded** scenarios.

When to Use `StringBuffer`?

✔ If **multiple threads** need to modify the same string.
✔ When **synchronization** is required to prevent data inconsistencies.

Key Differences (Comparison Table)

Feature	String	StringBuilder	StringBuffer
Mutability	Immutable	Mutable	Mutable
Thread-Safe	Yes (implicitly due to immutability)	No	Yes

Performance	Slow (creates new objects)	Fast	Slower than `StringBuilder` due to synchronization
Stored in SCP?	Yes (if created as a literal)	No	No
Use Case	When the string doesn't change	When fast performance is needed in a single thread	When working with multi-threaded applications

---

Use `String` if:

- The string is a **constant** and won't change.
- You want to **store strings efficiently** in the SCP.
- You are working with **small strings or text processing**.

Use `StringBuilder` if:

- You need to modify the string frequently (e.g., loops, concatenations).
- Performance is important and **thread-safety is not needed**.
- Suitable for **string manipulation in single-threaded applications**.

Use `StringBuffer` if:

- You are working in a **multi-threaded** environment.
- Multiple threads **need to modify the same string** safely.
- You require **synchronization for thread safety**.
