

UNIT-1

1.An Overview of Java: Object Oriented Programming:

Java is a high-level, object-oriented programming language that was developed by James Gosling at Sun Microsystems and later acquired by Oracle Corporation. It was first released in 1995 and has since become one of the most widely used programming languages. Here's an overview of Java with a focus on Object-Oriented Programming (OOP):

1. Object-Oriented Programming (OOP):

- Principles: Java follows the core principles of OOP, which include:
 - ✓ **Encapsulation:** Bundling data (variables) and methods (functions) that operate on the data into a single unit called a class.
 - ✓ **Inheritance:** Allowing a class to inherit properties and behaviors from another class, promoting code reusability.
 - ✓ **Polymorphism:** Allowing objects of different classes to be treated as objects of a common superclass, providing a way to implement flexibility and abstraction.
- **Classes and Objects:** Java is fundamentally based on classes and objects. A class is a blueprint for creating objects, and objects are instances of classes. Classes define attributes (fields) and behaviors (methods).
- **Abstraction and Encapsulation:** Java promotes abstraction by allowing developers to focus on the essential features of an object and hide the unnecessary details. Encapsulation ensures that the internal workings of a class are hidden from the outside world, and access to data is controlled through methods.
- **Inheritance:** Java supports single inheritance, where a class can inherit attributes and behaviors from one superclass. This promotes code reuse and the creation of a hierarchy of classes.
- **Polymorphism:** Java supports polymorphism through method overloading and method overriding. Method overloading allows multiple methods with the same name but different parameters, while method overriding enables a subclass to provide a specific implementation of a method defined in its superclass.

- **Interfaces:** Java allows the definition of interfaces, which are similar to abstract classes but provide a way to achieve multiple inheritance. Classes can implement multiple interfaces, enabling the implementation of different sets of behaviors.

2. Key Features of Java:

- ✓ **Platform Independence:** Java code is compiled into an intermediate form called bytecode, which can run on any device with a Java Virtual Machine (JVM). This "write once, run anywhere" philosophy contributes to platform independence.
- ✓ **Automatic Memory Management:** Java features automatic garbage collection, relieving developers from manually managing memory allocation and deallocation. This enhances the language's robustness and reduces the likelihood of memory-related errors.
- ✓ **Security:** Java incorporates security features like a bytecode verifier and runtime security checks, making it a secure language for developing applications.
- ✓ **Rich Standard Library:** Java comes with a vast standard library that provides pre-built functionality for common programming tasks, facilitating faster development.
- ✓ **Multithreading:** Java supports multithreading, allowing the execution of multiple threads concurrently. This feature is crucial for developing responsive and efficient applications.

In summary, Java's strong adherence to object-oriented principles, along with its platform independence, security features, and rich standard library, makes it a versatile and widely adopted programming language in various domains, including web development, enterprise applications, and mobile app development.

2. Data Types, Variables, and Arrays: Primitive Types-Literals Variables

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

- **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float, and double.
- **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

Java Primitive Data Types:

- In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.
- Java is a statically-typed programming language. It means all variables must be declared before their use. That is why we need to declare a variable's type and name.

There are 8 types of primitive data types:

1. **boolean data type**
2. **byte data type**
3. **char data type**
4. **short data type**
5. **int data type**
6. **long data type**
7. **float data type**
8. **double data type**

Java Data Types:

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

Boolean Data Type:

- The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.

Example:

Boolean one = false;

Byte Data Type:

- The byte data type is an 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128, and the maximum value is 127. Its default value is 0.
- The byte data type is used to save memory in large arrays where memory savings are most required.

Example:

byte a = 10, byte b = -20;

Short Data Type:

- The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768, and the maximum value is 32,767. Its default value is 0.
- The short data type can also be used to save memory, just like the byte data type.

Example:

short s = 10000, short r = -5000;

Int Data Type:

- The int data type is a 32-bit signed two's complement integer. Its value-range lies between -2,147,483,648 (-2^{31}) to 2,147,483,647 ($2^{31} - 1$) (inclusive). Its minimum value is -2,147,483,648, and the maximum value is 2,147,483,647. Its default value is 0.
- The int data type is generally used as a default data type for integral values unless there is no problem with memory.

Example:

int a = 100000, int b = -200000;

Long Data Type:

- The long data type is a 64-bit two's complement integer. Its value-range lies between -9,223,372,036,854,775,808 (-2^{63}) to 9,223,372,036,854,775,807 ($2^{63} - 1$) (inclusive). Its minimum value is -9,223,372,036,854,775,808, and the maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.

Example:

```
long a = 100000L, long b = -200000L;
```

Float Data Type:

- The float data type is a single-precision 32-bit IEEE 754 floating point. Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

Example:

```
float f1 = 234.5f;
```

Double Data Type:

- The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values, just like float. The double data type should also never be used for precise values, such as currency. Its default value is 0.0d.

Example:

```
double d1 = 12.3;
```

Char Data Type:

- The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive). The char data type is used to store characters.

Example:

```
char letterA = 'A';
```

Variable:

A variable is like a named box in the computer's memory where you can store information.

Types of Variables:

1. Local Variable:

What: A variable declared inside a method.

Example:

```
void method() {  
    int localVar = 90; // Local variable  
}
```

2. Instance Variable:

What: A variable declared inside a class but not inside any method.

Example:

```
class A {  
    int instanceVar = 50; // Instance variable  
}
```

3. Static Variable:

What: A variable declared as static in a class.

Example:

```
class A {  
    static int staticVar = 100; // Static variable  
}
```

Constants in Java:

A constant is like a variable, but its value can't be changed once set.

Declared using the final keyword.

Example:

```
final int GLOBAL_CONSTANT = 42; // Global constant  
static final int CLASS_CONSTANT = 100; // Class constant
```

Scope and Lifetime of Variables:

- ✓ Scope: Defines where a variable can be used.
- ✓ Lifetime: How long a variable exists.

Instance Variables:

- ✓ Defined in a class.
- ✓ Exist as long as the object (instance) of the class exists.

Argument Variables:

- ✓ Defined in the method or constructor header.
- ✓ Exist as long as the method or constructor is running.

Local Variables:

- ✓ Defined inside a method or constructor.
- ✓ Exist only within that method or constructor.

Important:

- Access specifiers (like public, private, etc.) apply to instance variables, not to argument or local variables.

In Java, you create variables to store information, and the type of variable (local, instance, static) depends on where and how you want to use it. Constants are for values that shouldn't change. Understanding the scope and lifetime helps you manage how and where variables can be used.

Literals in Java:

A literal is a source code representation of a fixed value. It is a constant value that can be assigned to variables.

Types of Literals in Java:

- Integer Literals:
 - Examples: 10, -5, 0x1F (hexadecimal), 077 (octal).
- Floating-Point Literals:
 - Examples: 3.14, -0.5, 2.0f (float), 1.0e3 (scientific notation).
- Character Literals:
 - Examples: 'A', '1', '\n' (newline), '\\' (backslash).
- String Literals:
 - Examples: "Hello", "Java", "123".
- Boolean Literals:
 - Examples: true, false.

Example:

```
int integerValue = 42;      // Integer literal
double doubleValue = 3.14; // Double literal
char charValue = 'A';      // Character literal
String stringValue = "Hello"; // String literal
boolean booleanValue = true; // Boolean literal
```

Type Conversion and Casting in Java

In Java, assigning a value of one type to a variable of another type is a common operation. Java supports both automatic type conversion and explicit casting (type casting) to handle such situations.

Automatic Type Conversions (Widening Conversion):

➤ Conditions for Automatic Conversion:

- The two types are compatible.
- The destination type is larger than the source type.

○ Examples of Widening Conversion:

```
int intValue = 10;
```

```
long longValue = intValue; // Automatic conversion from int to long
```

➤ Numeric Types Compatibility:

- Numeric types, including integers and floating-point types, are compatible for widening conversions.
- No automatic conversions between numeric types and char or boolean.
- char and boolean are not compatible with each other.

Casting (Narrowing Conversion):

➤ Explicit Type Conversion (Casting):

- When automatic conversion is not possible (e.g., assigning an int to a byte), explicit casting is used.
- General form: (target-type) value
- Example:

```
int intValue = 257;
```

```
byte byteValue = (byte) intValue; // Explicit casting from int to byte
```

➤ Truncation in Casting:

- Casting a floating-point value to an integer may result in truncation, losing the fractional part.

○ Example:

```
double doubleValue = 323.142;
```

```
int truncatedValue = (int) doubleValue; // Truncation of fractional part
```


Demonstration Program:

```
class Conversion {  
    public static void main(String args[]) {  
        byte b;  
        int i = 257;  
        double d = 323.142;  
  
        // Conversion of int to byte.  
        b = (byte) i;  
        System.out.println("i and b " + i + " " + b);  
  
        // Conversion of double to int.  
        i = (int) d;  
        System.out.println("d and i " + d + " " + i);  
  
        // Conversion of double to byte.  
        b = (byte) d;  
        System.out.println("d and b " + d + " " + b);  
    }  
}
```

Output of the Demonstration:

Conversion of int to byte.
i and b 257 1

Conversion of double to int.
d and i 323.142 323

Conversion of double to byte.
d and b 323.142 67

This program illustrates the use of explicit casting in scenarios where automatic conversions are not feasible, including cases of narrowing conversion and truncation.

Understanding Arrays in Programming

An array in programming is a collection of variables of the same data type, grouped under a common name. Arrays are widely used in programming languages to organize and manipulate data efficiently. They provide a way to store multiple values of the same type in contiguous memory locations, and each value in the array is identified by an index or a key.

Key characteristics of arrays include:

- 1. Data Type:** All elements in an array must be of the same data type, whether it's integers, floating-point numbers, characters, or any other data type.
- 2. Indexing:** Elements in an array are accessed using an index or a key. The index starts at 0 for the first element, 1 for the second, and so on.
- 3. Contiguous Memory Allocation:** In most programming languages, array elements are stored in contiguous memory locations. This allows for efficient access to elements through indexing.
- 4. Fixed Size:** Arrays usually have a fixed size, determined at the time of declaration. Once an array is created, its size doesn't change. Some languages, like Java, provide dynamic arrays or collections to overcome this limitation.

One-Dimensional Array

A one-dimensional array is a linear list of elements. The general syntax for declaring a one-dimensional array is:

```
type[] arrayName = new type[size];
```

Example:

```
int[] numbers = new int[5];
```

This creates an integer array named `numbers` with five elements.

Multi-Dimensional Array

A multi-dimensional array is an array of arrays, allowing you to represent data in multiple dimensions, such as a matrix. The syntax for declaring a two-dimensional array is:

```
type[][] arrayName = new type[rows][columns];
```

Example:

```
int[][] matrix = new int[3][4];
```

This creates a 3x4 matrix named `matrix`.

Array Initialization

- Arrays can be initialized at the time of declaration or later in the code:
 - `int[] numbers = {1, 2, 3, 4, 5};`
- This initializes a one-dimensional array with values 1, 2, 3, 4, and 5.
- Arrays provide an efficient way to work with large sets of data, and they play a crucial role in various algorithms and data structures. Understanding arrays is fundamental to programming in many languages.

One-Dimensional Array Example:

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

        // Accessing elements using index
        System.out.println("Element at index 2: " + numbers[2]);

        // Modifying an element
        numbers[3] = 45;

        // Displaying all elements using a loop
        System.out.println("All elements:");
        for (int i = 0; i < numbers.length; i++) {
            System.out.println(numbers[i]);
        }
    }
}
```

Element at index 2: 30

All elements:

10
20
30
45
50

Two-Dimensional Array Example:

```
public class TwoDimensionalArrayExample {  
    public static void main(String[] args) {  
        // Declaration and Initialization  
        int[][] matrix = {  
            {1, 2, 3},  
            {4, 5, 6},  
            {7, 8, 9}  
        };  
  
        // Accessing elements using indices  
        System.out.println("Element at row 1, column 2: " + matrix[0][1]);  
  
        // Modifying an element  
        matrix[1][1] = 55;  
  
        // Displaying all elements using nested loops  
        System.out.println("All 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(); // Move to the next line after each row  
        }  
    }  
}
```

Element at row 1, column 2: 2

All elements:

1 2 3

4 55 6

7 8 9

Operators in Java

1. Arithmetic Operators

Arithmetic operators perform mathematical operations on numerical values.

+ (Addition): Adds two values.

- (Subtraction): Subtracts the right operand from the left operand.

* (Multiplication): Multiplies two values.

/ (Division): Divides the left operand by the right operand.

% (Modulo): Returns the remainder of the division.

Examples:

```
int a = 10, b = 3;  
int sum = a + b; // 13  
int difference = a - b; // 7  
int product = a * b; // 30  
int quotient = a / b; // 3  
int remainder = a % b; // 1
```

2. Increment and Decrement Operators

++ (Increment): Increases the value of the operand by 1.

-- (Decrement): Decreases the value of the operand by 1.

Examples:

```
int x = 5;  
x++; // x is now 6  
x--; // x is back to 5
```

3. Assignment Operators

Assignment operators assign values to variables.

= (Assignment): Assigns the value on the right to the variable on the left.

+=, -=, *=, /=, %= (Compound Assignment): Performs the operation and assigns the result to the left operand.

Examples:

```
int x = 5;  
x += 3; // Equivalent to x = x + 3; x is now 8
```

4. Bitwise Operators

Bitwise operators perform operations at the bit level.

& (Bitwise AND): Sets each bit to 1 if both bits are 1.

| (Bitwise OR): Sets each bit to 1 if at least one bit is 1.

^ (Bitwise XOR): Sets each bit to 1 if only one of the bits is 1.

~ (Bitwise NOT): Flips the bits.

Examples:

```
int a = 0b1010, b = 0b1101;  
int c = a & b; // 0b1000
```

5. Relational Operators

Relational operators compare two values and return a boolean result.

== (Equal to), != (Not equal to): Compare for equality.

>, < (Greater than, Less than): Compare magnitudes.

>=, <= (Greater than or equal to, Less than or equal to): Compare magnitudes with equality.

Examples:

```
int a = 10, b = 20;  
if (a > b) {  
    System.out.println("A is Bigger than B");  
} else {  
    System.out.println("B is Bigger than A");  
}
```

6. Logical Operators

Logical operators perform operations on boolean values.

&& (Logical AND), || (Logical OR): Perform short-circuit AND and OR operations.

! (Logical NOT): Negates a boolean value.

Examples:

```
boolean x = true, y = false, z = true;  
boolean d = x || y; // true  
boolean e = !x; // false
```

7. Conditional Operator (?:)

The conditional operator is a ternary operator that works like a concise form of the if-else statement.

Example:

```
int a = 25, b = 34;  
int x = (a > b) ? a : b; // If a is greater than b, x = a; otherwise, x = b
```

8. Operator Precedence

Operators in Java have their own priority. It is essential to understand the priority to write correct expressions. Parentheses can change the priority of operators.

Highest Priority: () []

Lowest Priority: = operator=

3. Control Statements-Classes and Methods-Inheritance-Exception Handling.

Control Statements in Java

Java provides various control statements to manage the flow of a program. There are three main types:

1. Decision-Making Statements:

if Statement:

```
if (condition) {  
    // code when the condition is true  
}
```

if-else Statement:

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

if-else-if ladder:

```
if (condition1) {  
    // code when condition1 is true  
} else if (condition2) {  
    // code when condition2 is true  
} else {  
    // code when none of the conditions are true  
}
```

Nested if Statement:

```
if (condition1) {  
    if (condition2) {  
        // code when both conditions are true  
    }  
}
```

2. Switch Statement:

Used to perform different actions based on the value of an expression.

```
switch (expression) {  
    case value1:  
        // code for value1  
        break;
```

```

        case value2:
            // code for value2
            break;
        default:
            // default code
    }

```

3. Loop Statements:

for Loop:

```

    for (initialization; condition; update) {
        // code to be executed in each iteration
    }

```

while Loop:

```

    while (condition) {
        // code to be executed while the condition is true
    }

```

do-while Loop:

```

    do {
        // code to be executed at least once
    } while (condition);

```

4. Jump Statements:

break Statement:

```

    for (int i = 0; i < 10; i++) {
        if (i == 5) {
            break; // exit the loop when i equals 5
        }
    }

```

continue Statement:

```

    for (int i = 0; i < 10; i++) {
        if (i == 5) {
            continue; // skip the rest of the code for i equals 5
        }
    }

```

These control statements are fundamental for structuring and controlling the flow of Java programs.

Note: Write program and flowchart diagram

Classes and Methods

In object-oriented programming, a class is a blueprint or template for creating objects. Objects are instances of a class, and a class defines the properties (attributes) and behaviors (methods) that its objects will have. Classes are a fundamental concept in languages like Java, and they provide a way to model and structure code in a modular and reusable manner.

Here's a breakdown of key concepts related to classes and methods in the context of Java:

Class:

Definition: A class is a user-defined data type in Java that encapsulates data (attributes) and methods (functions) to operate on that data.

Purpose: It serves as a template for creating objects with a common structure and behavior.

Example: In the provided Java code snippets, `Ovload`, `Consovload`, and `Complex` are examples of classes.

Object:

Definition: An object is an instance of a class. It represents a real-world entity and is created based on the structure defined by the class.

Example: `d1`, `d2`, `d3`, `c1`, `c2`, and `c3` are instances (objects) of the classes `Ovload` and `Complex` in the given code.

Method:

Definition: A method is a block of code within a class that performs a specific task. It defines the behavior of the class.

Types: There are two types of methods: instance methods and static methods. Instance methods operate on an instance of the class (object), while static methods belong to the class itself.

Example: In the provided code snippets, `area`, `input`, `process`, `output`, `fact`, and `output` are examples of methods.

Method Overloading:

Definition: It allows a class to have multiple methods with the same name but different parameters (number, type, or order).

Example: The `area` method in the `Ovload` class demonstrates method overloading for calculating the area of a square, rectangle, and circle with different sets of parameters.

Constructor:

Definition: A constructor is a special type of method that is invoked when an object is created. It initializes the object's state.

Purpose: It ensures that the object is properly initialized before it's used.

Example: The `Consoload` class has constructors for initializing objects with different sets of parameters.

Access Control (Modifiers):

Public: Accessible from anywhere.

Private: Accessible only within the same class.

Protected: Accessible within the same class and its subclasses.

Default (No Modifier): Accessible within the same package.

Example: The `public`, `private`, and `protected` keywords are used to control access to class members in the `Demoaccs` class.

Static:

Definition: The `static` keyword is used to declare class-level members (variables or methods) that belong to the class rather than a specific instance of the class.

Purpose: Static members are shared among all instances of the class.

Example: The `a` and `b` variables in the `Demostat` class are declared as `static`.

Final:

Definition: The `final` keyword is used to declare constants, variables, methods, or classes that cannot be modified.

Purpose: It ensures that the value or implementation cannot be changed.

Example: The `final` keyword is used in the `Demofinal` class to declare constant variables and a `final` method.

Recursion:

Definition: Recursion is a programming technique where a method calls itself to solve a smaller instance of the same problem.

Example: The `fact` method in the `Recur` class demonstrates recursion by calculating the factorial of a number.

Command Line Arguments:

Definition: Command line arguments are values provided to a program when it is executed from the command line.

Purpose: They allow external data to be passed into the program.

Example: The main method in the Cmdlnarg class takes command line arguments and processes them to find the sum of two numbers.

These concepts collectively contribute to the creation of modular, reusable, and well-organized code in Java. Each class encapsulates data and behavior, promoting a clear separation of concerns and enhancing code maintainability.

Here's an example Java program that incorporates various concepts related to classes and methods, including method overloading, constructors, access control, static members, and command line arguments:

```
class Rectangle {
    double length, width;
    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
    double calculateArea() {
        return length * width;
    }
    double calculatePerimeter() {
        return 2 * (length + width);
    }
}
class Circle {
    double radius;
    Circle(double radius) {
        this.radius = radius;
    }
    double calculateArea() {
        return Math.PI * radius * radius;
    }
    double calculatePerimeter() {
        return 2 * Math.PI * radius;
    }
}
public class Main {
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter the length of the rectangle: ");
    double length = scanner.nextDouble();
```

```

        System.out.print("Enter the width of the rectangle: ");
        double width = scanner.nextDouble();
        Rectangle rectangle = new Rectangle(length, width);
        double rectangleArea = rectangle.calculateArea();
        double rectanglePerimeter = rectangle.calculatePerimeter();
        System.out.println("Rectangle Area: " + rectangleArea);
        System.out.println("Rectangle Perimeter: " + rectanglePerimeter);
        System.out.print("Enter the radius of the circle: ");
        double radius = scanner.nextDouble();
        Circle circle = new Circle(radius);
        double circleArea = circle.calculateArea();
        double circlePerimeter = circle.calculatePerimeter();
        System.out.println("Circle Area: " + circleArea);
        System.out.println("Circle Perimeter: " + circlePerimeter);
        scanner.close();
    }
}

```

This program defines a Shape class with constructors, method overloading, access control, and a static method. The MainClass demonstrates the usage of command line arguments to create objects of the Shape class and calculates their areas based on their dimensions. It also calls a static method of the class.

Inheritance in Java

Inheritance in Java is a way for one class to acquire the properties and behaviors of another class. It's a key concept in Object-Oriented Programming (OOP), allowing you to create new classes based on existing ones.

Why Use Inheritance in Java?

1. **Method Overriding:** Achieve runtime polymorphism by allowing a subclass to provide a specific implementation of a method already defined in its superclass.
2. **Code Reusability:** Reuse fields and methods of an existing class when creating a new class, promoting efficient and organized code.

Key Terms:

- **Class:** A blueprint or template for creating objects.
- **Subclass/Child Class:** Inherits properties and behaviors from another class. Also called a derived or extended class.
- **Superclass/Parent Class:** Provides properties and behaviors to be inherited by a subclass. Also called a base class.

Syntax:

```
class Subclass extends Superclass {  
    // methods and fields  
}
```

Example:

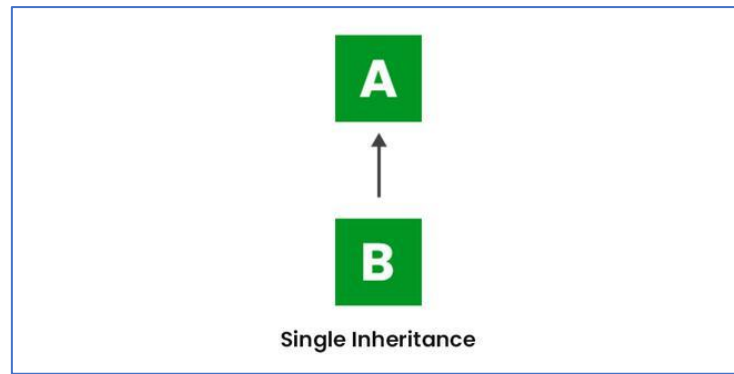
```
class Employee {  
    float salary = 40000;  
}  
  
class Programmer extends Employee {  
    int bonus = 10000;  
  
    public static void main(String args[]) {  
        Programmer p = new Programmer();  
        System.out.println("Programmer salary: " + p.salary);  
        System.out.println("Bonus of Programmer: " + p.bonus);  
    }  
}
```

Types of Inheritance:

1. Single Inheritance: A subclass can inherit from only one superclass.

Example:

```
class Animal {  
    void eat() { System.out.println("eating..."); }  
}  
  
class Dog extends Animal {  
    void bark() { System.out.println("barking..."); }  
}
```



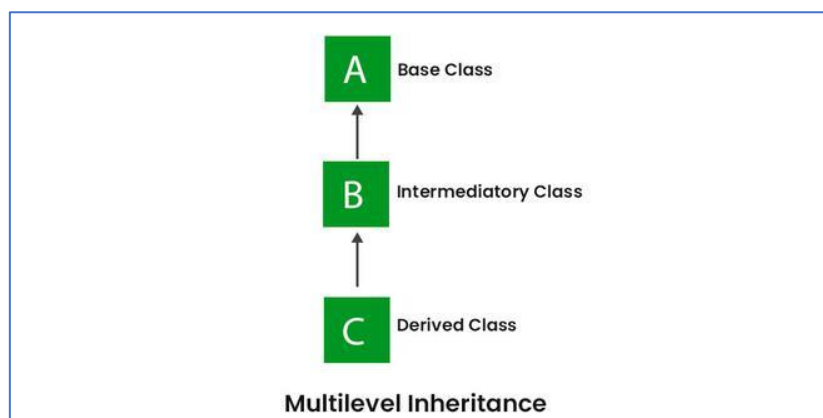
2.Multilevel Inheritance: A class can inherit from another class, and then another class can inherit from it, forming a chain of inheritance.

-Example:

```
class Animal {  
    void eat() { System.out.println("eating..."); }  
}
```

```
class Dog extends Animal {  
    void bark() { System.out.println("barking..."); }  
}
```

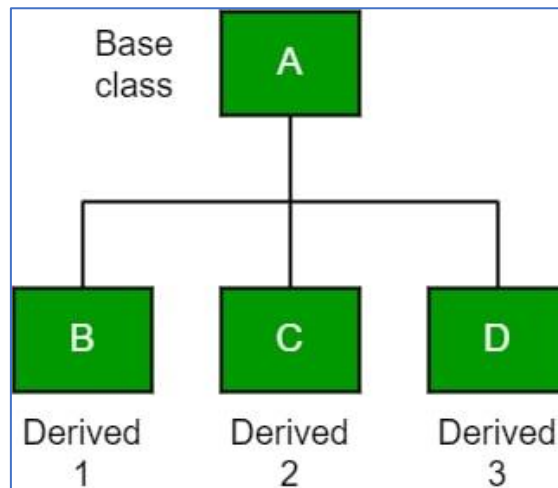
```
class BabyDog extends Dog {  
    void weep() { System.out.println("weeping..."); }  
}
```



3. Hierarchical Inheritance: Multiple classes can inherit from a single superclass.

- Example:

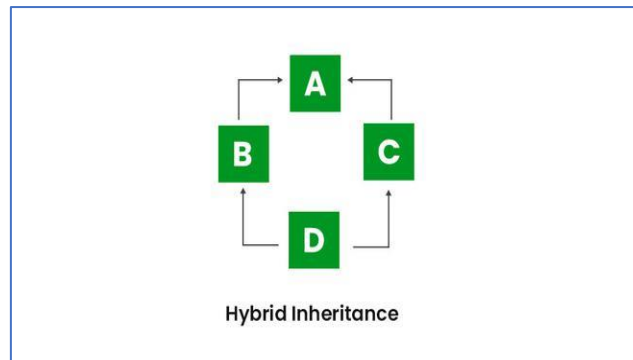
```
class Animal {  
    void eat() { System.out.println("eating..."); }  
}  
class Dog extends Animal {  
    void bark() { System.out.println("barking..."); }  
}  
  
class Cat extends Animal {  
    void meow() { System.out.println("meowing..."); }  
}
```



4. Hybrid Inheritance: Hybrid inheritance is a combination of two or more types of inheritance within a single program.

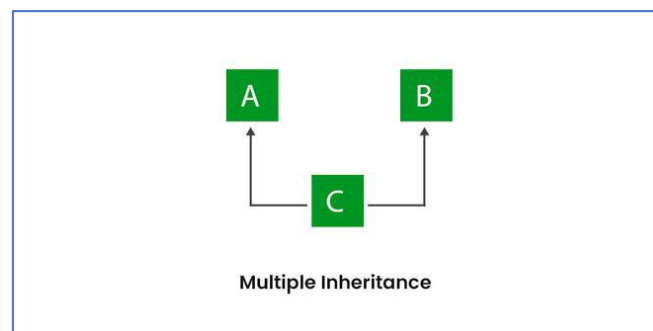
- Example:

```
public class SolarSystem {  
}  
public class Earth extends SolarSystem {  
}  
public class Mars extends SolarSystem {  
}  
public class Moon extends Earth {  
}
```



Why No Multiple Inheritance in Java?

Multiple inheritance (inheriting from more than one class) is not supported in Java to avoid complications and maintain language simplicity. It helps prevent ambiguity and compile-time errors that may arise when multiple parent classes have methods with the same name. Java supports multiple inheritance through interfaces.



Example of Unsupported Multiple Inheritance:

```

class A {
    void msg() { System.out.println("Hello"); }
}
class B {
    void msg() { System.out.println("Welcome"); }
}
// Error: Multiple inheritance is not allowed
class C extends A, B {
    public static void main(String args[]) {
        C obj = new C();
        obj.msg(); // Compile Time Error - Ambiguity
    }
}
  
```

Java prefers compile-time errors over potential runtime errors to maintain code reliability.

Exception Handling:

Exception handling in Java is a mechanism to deal with runtime errors or abnormal conditions that may occur during the execution of a program. It uses five keywords: try, catch, throw, throws, and finally. Let's simplify these concepts.

Try-Catch Block:

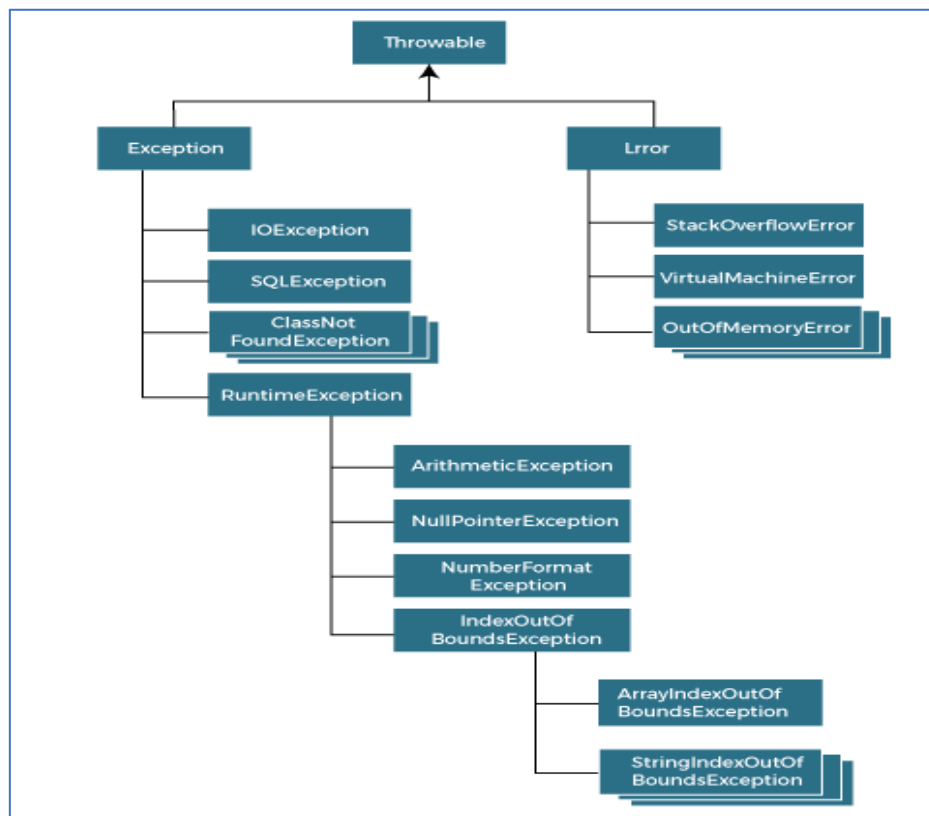
```
try {  
    // Code that might cause an exception  
} catch (ExceptionType1 ex1) {  
    // Handle exception of type ExceptionType1  
} catch (ExceptionType2 ex2) {  
    // Handle exception of type ExceptionType2  
} finally {  
    // Code that always gets executed (optional)  
}
```

Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Exception Types:

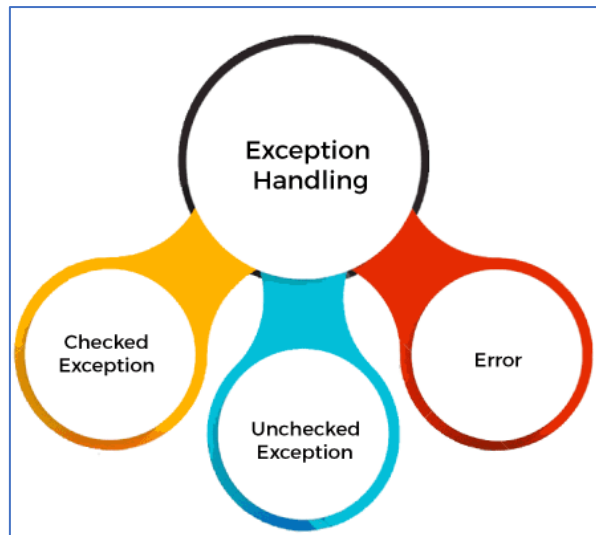


- **ArithmeticException:** Arithmetic errors like divide by zero.
- **ArrayIndexOutOfBoundsException:** Array index exceeds the range.
- **NegativeArraySizeException:** Array index is negative.
- **ClassCastException:** Invalid cast.
- **NullPointerException:** Invalid use of a null pointer.
- **StringIndexOutOfBoundsException:** Outside the bounds of a String.

Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error



1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Example - Array Index Out of Bounds Exception:

```
class ArrayExceptionExample {  
    public static void main(String[] args) {  
        try {  
            int a[] = new int[5];  
            for (int i = 0; i < 5; i++)  
                a[i] = i;  
            a[20] = 15; // Array Index out of Range  
            for (int i = 0; i < 5; i++)
```

```
        System.out.println(a[i]);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Array Index Out of Range: " + e);
    }
    System.out.println("After Catch");
}
```

In the second example, the catch block handles the `ArrayIndexOutOfBoundsException` when trying to access an index outside the array's range.

Exception handling is crucial for writing robust and reliable Java programs. It allows you to gracefully handle unexpected errors and maintain the normal flow of your application.

UNIT-2

1. String Handling: The String Constructors - String Length - Special String Operations - Character Extraction - String Comparison - Searching Strings - Modifying a String

String Handling: The String Constructor

The String class in Java offers various constructors to create and initialize String objects.

Default Constructor:

To create an empty String, use the default constructor:

```
String s = new String();
```

This results in an instance of String with no characters.

Constructor with Character Array:

To initialize a String using an array of characters:

```
char chars[] = { 'a', 'b', 'c' };
```

```
String s = new String(chars);
```

This creates a String with the value "abc."

Constructor with Character Array Subrange:

Specify a subrange of a character array as an initializer:

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
```

```
String s = new String(chars, 2, 3);
```

This initializes the String with the characters "cde."

Constructor from Another String:

Construct a String object with the same character sequence as another String:

```
String s1 = new String(c);
```

```
String s2 = new String(s1);
```

The result is that s1 and s2 contain the same string.

Constructor with Byte Array:

Initialize a String when given a byte array:

```
byte ascii[] = {65, 66, 67, 68, 69, 70 };
```

```
String s1 = new String(ascii);
```

This creates a String with the value "ABCDEF."

Constructor with Byte Array Subrange:

Specify a subrange when initializing from a byte array:

```
String s2 = new String(ascii, 2, 3);
```

This results in the String "CDE."

Constructors with Character Encoding:

Extended versions allow specifying character encoding for byte-to-string conversion.

String from StringBuffer or StringBuilder:

Construct a String from a StringBuffer or a StringBuilder:

```
String s3 = new String(StringBufferObj);
```

```
String s4 = new String(StringBuilderObj);
```

Constructor for Unicode Character Set:

Supports the extended Unicode character set using code points:

```
String s5 = new String(codePoints, startIndex, numChars);
```

Note: Modifying the array after creating a String object won't affect the String.

String Length:

- The length of a string is the number of characters that it contains.
- To obtain this value, call the **length()** method, shown here:

```
int length( )
```

The following fragment prints "3", since there are three characters in the string s:

```
char chars[] = { 'a', 'b', 'c' };  
String s = new String(chars);  
System.out.println(s.length());
```

Special String Operations:

Strings play a crucial role in programming, and Java provides convenient support for various string operations. These operations include the automatic creation of String instances from string literals, concatenation using the + operator, and conversion of other data types to string representation.

String Literals

Instead of explicitly creating String instances using the new operator, Java allows the use of string literals. For each string literal, Java automatically constructs a String object. For example:

```
char chars[] = { 'a', 'b', 'c' };  
String s1 = new String(chars);  
String s2 = "abc"; // using string literal
```

String Concatenation

Java allows the use of the + operator to concatenate strings, producing a new String object as the result. This enables chaining multiple concatenation operations. For example:

```
String age = "9";  
String s = "He is " + age + " years old."  
System.out.println(s); // Outputs: "He is 9 years old."
```

String concatenation is practical when creating long strings to prevent them from wrapping within the source code.

```
String longStr = "This could have been " +  
                "a very long line that would have " +  
                "wrapped around. But string concatenation " +  
                "prevents this."  
System.out.println(longStr);
```

String Concatenation with Other Data Types

Strings can be concatenated with other data types. For instance:

```
int age = 9;  
String s = "He is " + age + " years old."  
System.out.println(s); // Outputs: "He is 9 years old."
```

Java automatically converts other data types to their string representation during concatenation.

String Conversion and toString()

During concatenation, Java uses the `valueOf()` method for string conversion. For objects, it invokes the `toString()` method. It is recommended to override `toString()` in user-defined classes to provide a meaningful string representation. For example:

```
class Box {  
    double width, height, depth;  
  
    // constructor and other methods here  
    public String toString() {  
        return "Dimensions are " + width + " by " + depth + " by " +  
height + ".";  
    }  
}
```

Now, instances of the `Box` class can be seamlessly used in concatenation expressions or `println()` statements, invoking the overridden `toString()` method.

```
Box b = new Box(10, 12, 14);  
String s = "Box b: " + b;  
System.out.println(b); // Automatically calls toString()  
System.out.println(s);
```

This ensures integration into Java's programming environment.

Character Extraction in Java: Overview

The `String` class in Java offers various methods for extracting characters from a `String` object. These methods provide flexibility in obtaining single or multiple characters, converting strings into different representations, and more.

charAt() Method:

Extracts a single character from a specified index.

Example:

```
char ch = "abc".charAt(1); // assigns 'b' to ch
```

getChars() Method:

Extracts a range of characters and stores them in a character array.

Example:

```
String s = "This is a demo of the getChars method.";  
int start = 10, end = 14;  
char buf[] = new char[end - start];  
s.getChars(start, end, buf, 0);
```


getBytes() Method:

Stores characters in an array of bytes using default character-to-byte conversions.

Example:

```
byte[] byteArray = "Hello".getBytes();
```

Useful for exporting a String value into environments not supporting 16-bit Unicode characters, such as Internet protocols using 8-bit ASCII.

toArray() Method:

Converts all characters in a String into a character array.

Example:

```
char[] charArray = "abc".toArray();
```

Provides a convenient way to achieve the same result as getChars().

These methods enhance the programmer's ability to manipulate strings efficiently, catering to different scenarios where character extraction or conversion is required.

String Comparison in Java: Understanding and Applying

String comparison in Java is a critical aspect of working with textual data. The String class provides various methods for comparing strings or substrings within strings. Let's explore these methods in detail:

1. equals() and equalsIgnoreCase()

- equals(Object str) compares two strings for equality in a case-sensitive manner.
- equalsIgnoreCase(String str) performs a case-insensitive comparison.

Example:

```
String s1 = "Hello";  
String s2 = "HELLO";  
System.out.println(s1.equals(s2)); // true  
System.out.println(s1.equalsIgnoreCase(s2)); // true
```

2. regionMatches:

- Compares specific regions inside two strings, with an option to ignore case.
- General Forms:
boolean regionMatches(int startIndex, String str2, int str2StartIndex, int numChars)

`boolean regionMatches(boolean ignoreCase, int startIndex, String str2,
int str2StartIndex, int numChars)`

Example:

```
String s1 = "This is a test";  
String s2 = "This can be a TEST";  
int start = 10, start1 = 14, numChars = 4;  
System.out.println(s1.regionMatches(start, s2, start1,  
numChars)); // true
```

3. startsWith() and endsWith() (2 Marks):

- `startsWith(String str)` and `endsWith(String str)` check if a string begins or ends with a specified substring.

Example:

```
System.out.println("Foobar".endsWith("bar")); // true  
System.out.println("Foobar".startsWith("Foo")); // true
```

4. equals() Versus ==:

- `equals()` compares characters inside a String object, while `==` compares object references.

Example:

```
String s1 = "Hello";  
String s2 = new String(s1);  
System.out.println(s1.equals(s2)); // true  
System.out.println(s1 == s2); // false
```

5. compareTo() :

- `compareTo(String str)` determines the dictionary order of strings.

Returns:

< 0 if invoking string is less than str.
> 0 if greater than str.
= 0 if equal.

Example:

```
String s1 = "GF";  
String s2 = "Now";  
System.out.println(s1.compareTo(s2)); // Negative value
```

6. compareToIgnoreCase():

- `compareToIgnoreCase(String str)` performs case-insensitive comparison.

Example:

```
String s1 = "GF";  
String s2 = "Now";  
System.out.println(s1.compareToIgnoreCase(s2)); // Ignoring case
```

Understanding these methods is crucial for effective string manipulation and comparison in Java. They play a vital role in scenarios such as sorting and searching within text data.

Searching Strings in Java:

In Java, the String class provides methods to search for specific characters or substrings within a string. Let's explore the main methods used for string searching:

1. indexOf() and lastIndexOf() (5 Marks):

indexOf(int ch) and lastIndexOf(int ch):

- Searches for the first or last occurrence of a character in the string.
- Returns the index at which the character was found or -1 if not found.

indexOf(String str) and lastIndexOf(String str):

- Searches for the first or last occurrence of a substring.
- Returns the index at which the substring was found or -1 if not found.

Overloaded forms to specify a starting point:

- `int indexOf(int ch, int startIndex)`
- `int lastIndexOf(int ch, int startIndex)`
- `int indexOf(String str, int startIndex)`
- `int lastIndexOf(String str, int startIndex)`

```
String s = "Now is the time for all good men to come to the aid of their country.";
```

```
System.out.println(s.indexOf('t')); // 7
```

```
System.out.println(s.lastIndexOf('t')); // 65
```

```
System.out.println(s.indexOf("the")); // 7
```

```
System.out.println(s.lastIndexOf("the")); // 55
```

```
System.out.println(s.indexOf('t', 10)); // 11
```

```
System.out.println(s.lastIndexOf('t', 60)); // 55
```

2. substring() (2 Marks):

substring(int startIndex):

- Returns a copy of the substring from startIndex to the end of the string.

substring(int startIndex, int endIndex):

- Returns a copy of the substring from startIndex to endIndex - 1.

```
String s = "This is a test. This is, too.";
String search = "is";
String sub = "was";
int i;
do {
    i = s.indexOf(search);
    if (i != -1) {
        s = s.substring(0, i) + sub + s.substring(i + search.length());
    }
} while (i != -1);
```

These string searching methods are crucial for finding specific characters or substrings within a larger string. They provide flexibility by allowing you to search from the beginning or end of a string and specify starting points for the search.

Modifying a String in Java (10 Marks):

In Java, the String class is immutable, meaning its content cannot be changed once created. To modify a string, you need to use various methods that create a new string with the desired modifications. Here are some commonly used methods for modifying strings:

1. substring():

Purpose:

To extract a substring from the original string.

Forms:

String substring(int startIndex): Returns a substring from startIndex to the end of the invoking string.

String substring(int startIndex, int endIndex): Returns a substring from startIndex to endIndex - 1.

Example:

```
String original = "This is a test.";
String sub1 = original.substring(5); // "is a test."
String sub2 = original.substring(0, 4); // "This"
```

2. concat() :

Purpose:

To concatenate two strings.

General Form:

```
String concat(String str)
```

Example:

```
String s1 = "One";
String s2 = s1.concat(" Two");
System.out.println(s2); // Output: One Two
```

3. replace() (2 Marks):

Purpose:

Replaces all occurrences of one character in the invoking string with another character.

General Form:

```
String replace(char original, char replacement)
```

Example:

```
String s = "Hello";
String replaced = s.replace('l', 'w');
System.out.println(replaced); // Output: Hewwo
```

4. trim() (2 Marks):

Purpose:

Returns a copy of the invoking string from which any leading and trailing whitespace has been removed.

General Form:

```
String trim()
```

Example:

```
String withSpaces = " Hello World ";
String trimmed = withSpaces.trim();
System.out.println(trimmed); // Output: Hello World
```

5. Changing the Case :

toLowerCase() and toUpperCase():

- String toLowerCase(): Converts all characters in a string from uppercase to lowercase.
- String toUpperCase(): Converts all characters in a string from lowercase to uppercase.

Example:

```
String s = "This is a test";  
String upper = s.toUpperCase();  
String lower = s.toLowerCase();  
System.out.println(upper); // Output: THIS IS A TEST  
System.out.println(lower); // Output: this is a test
```

These methods allow you to manipulate strings according to your needs, creating new strings with the desired modifications while leaving the original strings unchanged.

The I/O Classes and Interfaces in Java

Java's Input/Output (I/O) system is built around the `java.io` package, offering a rich set of classes and interfaces for handling input and output operations. The primary interfaces include `InputStream`, `OutputStream`, `Reader`, and `Writer`. These interfaces are implemented by various classes that cater to different types of data and sources.

1. `InputStream` and `OutputStream`:

a. `InputStream`:

Purpose:

Used for reading binary data.

Commonly Used Methods:

`int read()`: Reads the next byte of data.

`int read(byte[] b)`: Reads some number of bytes into an array.

b. `OutputStream`:

Purpose:

Used for writing binary data.

Commonly Used Methods:

`void write(int b)`: Writes the specified byte to the output stream.

`void write(byte[] b)`: Writes bytes from the specified byte array to the output stream.

2. `Reader` and `Writer`:

a. `Reader`:

Purpose:

Used for reading character data.

Commonly Used Methods:

`int read()`: Reads a single character.

`int read(char[] c)`: Reads characters into an array.

b. Writer:

Purpose:

Used for writing character data.

Commonly Used Methods:

void write(int c): Writes a single character.

void write(char[] c): Writes characters from an array.

3. BufferedReader and BufferedWriter:

a. BufferedReader:

Purpose:

Efficiently reads text from a character-based input stream.

Commonly Used Methods:

String readLine(): Reads a line of text.

b. BufferedWriter:

Purpose:

Efficiently writes text to a character-based output stream.

Commonly Used Methods:

void write(String s): Writes a string.

4. FileReader and FileWriter:

a. FileReader:

Purpose:

Reads the contents of a file as a stream of characters.

Commonly Used Constructors:

FileReader(String fileName): Creates a new FileReader, given the name of the file.

b. FileWriter:

Purpose:

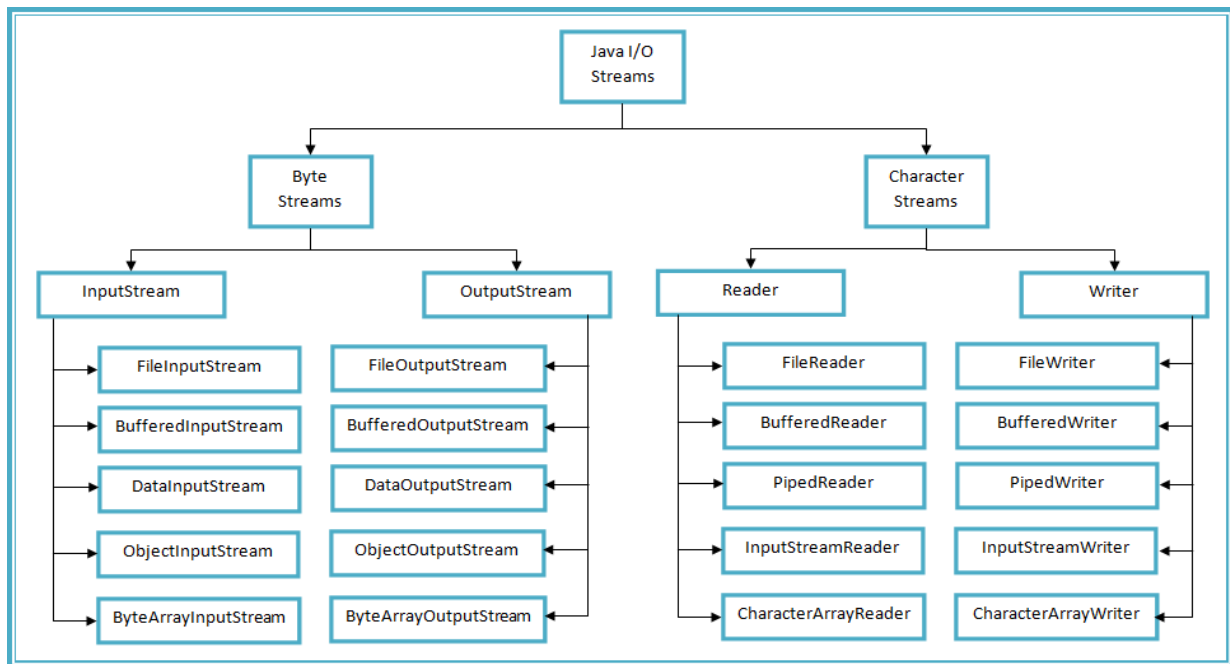
Writes characters to a file.

Commonly Used Constructors:

FileWriter(String fileName): Creates a new FileWriter, given the name of the file.

Conclusion:

Java's I/O classes and interfaces provide a flexible and powerful mechanism for handling various input and output scenarios. Whether dealing with binary or character data, interacting with files, or utilizing buffered streams for efficiency, these classes offer a comprehensive toolkit for developers working on diverse applications. Understanding and utilizing these classes is crucial for effective data processing in Java programs.



ByteStream Classes in Java:

Byte streams in Java are used for handling the input and output of raw binary data. They are suitable for reading and writing non-text data, such as images, audio, video, and other types of files where the data is not composed of characters. Byte streams operate on the level of individual bytes and are part of the java.io package.

The two main abstract classes for byte streams are `InputStream` and `OutputStream`.

InputStream Class

The `InputStream` class is the superclass for all classes that represent input streams of bytes. It provides methods for reading bytes from a source. Some important methods include:

`int read():` Reads a single byte as an integer.
`int read(byte[] b):` Reads bytes into an array.
`int read(byte[] b, int off, int len):` Reads bytes into a portion of an array.
`void close():` Closes the stream.

Common subclasses of `InputStream` include `FileInputStream`, `ByteArrayInputStream`, and `BufferedInputStream`.

OutputStream Class

The `OutputStream` class is the superclass for all classes that represent output streams of bytes. It provides methods for writing bytes to a destination. Important methods include:

`void write(int b):` Writes a single byte.
`void write(byte[] b):` Writes an array of bytes.
`void write(byte[] b, int off, int len):` Writes a portion of an array.
`void close():` Closes the stream.

Common subclasses of `OutputStream` include `FileOutputStream`, `ByteArrayOutputStream`, and `BufferedOutputStream`.

FileInputStream and FileOutputStream

`FileInputStream` and `FileOutputStream` are concrete classes that allow reading from and writing to files using byte streams.

Example of reading from a file using FileInputStream:

```
try (InputStream inputStream = new FileInputStream("example.bin")) {
    int data;
    while ((data = inputStream.read()) != -1) {
        System.out.print(data + " ");
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Example of writing to a file using FileOutputStream:

```
try (OutputStream outputStream = new FileOutputStream("output.bin")) {
    byte[] data = { 65, 66, 67, 68, 69 };
    outputStream.write(data);
} catch (IOException e) {
    e.printStackTrace();
}
```

BufferedInputStream and BufferedOutputStream

`BufferedInputStream` and `BufferedOutputStream` are classes that provide buffering for byte input and output streams, enhancing the performance and efficiency of reading and writing.

Example of reading from a file using BufferedInputStream:

```
try (BufferedInputStream bufferedInputStream = new
BufferedInputStream(new FileInputStream("example.bin"))) {
    int data;
    while ((data = bufferedInputStream.read()) != -1) {
        System.out.print(data + " ");
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Example of writing to a file using `BufferedOutputStream`:

```
try (BufferedOutputStream bufferedOutputStream = new
BufferedOutputStream(new FileOutputStream("output.bin"))) {
    byte[] data = { 65, 66, 67, 68, 69 };
    bufferedOutputStream.write(data);
} catch (IOException e) {
    e.printStackTrace();
}
```

Byte streams are essential for handling binary data and are commonly used for tasks involving reading and writing files that are not text-based. They provide a low-level, efficient mechanism for working with raw binary data.

Character streams

Character streams in Java are used for handling the input and output of characters. Unlike byte streams, which deal with raw binary data, character streams are designed for reading and writing text data. Character streams use Unicode and provide a higher-level abstraction for working with characters, making them suitable for processing text files.

The primary classes for character streams are `Reader` and `Writer`, which are abstract classes extended by various concrete classes for specific purposes. These classes are part of the `java.io` package.

Reader Class:

The `Reader` class is an abstract class that serves as the superclass for all classes that read characters. Some important methods of the `Reader` class include:

- `int read()`: Reads a single character as an integer.
- `int read(char[] cbuf)`: Reads characters into an array.
- `int read(char[] cbuf, int off, int len)`: Reads characters into a portion of an array.
- `void close()`: Closes the stream.

Common subclasses of `Reader` include `FileReader`, `BufferedReader`, and `StringReader`.

Writer Class:

The `Writer` class is an abstract class that serves as the superclass for all classes that write characters. Important methods of the `Writer` class include:

- `void write(int c)`: Writes a single character.
- `void write(char[] cbuf)`: Writes an array of characters.
- `void write(String str)`: Writes a string.

`void write(char[] cbuf, int off, int len):` Writes a portion of an array.

`void close():` Closes the stream.

Common subclasses of `Writer` include `FileWriter`, `BufferedWriter`, and `StringWriter`.

FileReader and FileWriter:

`FileReader` and `FileWriter` are concrete classes that allow reading from and writing to files using character streams.

Example of reading from a file using FileReader:

```
try (Reader reader = new FileReader("example.txt")) {
    int data;
    while ((data = reader.read()) != -1) {
        char character = (char) data;
        System.out.print(character);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Example of writing to a file using FileWriter:

```
try (Writer writer = new FileWriter("output.txt")) {
    String content = "Hello, Character Streams!";
    writer.write(content);
} catch (IOException e) {
    e.printStackTrace();
}
```

BufferedReader and BufferedWriter

`BufferedReader` and `BufferedWriter` are classes that provide buffering for character input and output streams, enhancing the performance and efficiency of reading and writing.

Example of reading from a file using BufferedReader:

```
try (BufferedReader reader = new BufferedReader(new
FileReader("example.txt"))) {
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Example of writing to a file using BufferedWriter:

```
try (BufferedWriter writer = new BufferedWriter(new
FileWriter("output.txt"))) {
    String content = "Hello, Buffered Streams!";
    writer.write(content);
} catch (IOException e) {
    e.printStackTrace();
}
```

Character streams are particularly useful when dealing with text-based data, providing a convenient and efficient way to work with characters and strings.

Difference b/w character and byte stream:

Aspect	Character Streams	Byte Streams
Data Handling	Handle character-based data	Handle raw binary data
Representation	Classes end with "Reader" or "Writer"	Classes end with "InputStream" or "OutputStream"
Suitable for	Textual data, strings, human-readable info	Non-textual data, binary files, multimedia
Character Encoding	Automatic encoding and decoding	No encoding or decoding
Text vs non-Text data	Text-based data, strings	Binary data, images, audio, video
Performance	Additional conversion may impact performance	Efficient for handling large binary data
Handle Large Text Files	May impact performance due to encoding	Efficient, no encoding overhead
String Operations	Convenient methods for string operations	Not specifically designed for string operations
Convenience Methods	Higher-level abstractions for text data	Low-level interface for byte data
Reading Line by Line	Convenient methods for reading lines	Byte-oriented, no built-in line-reading methods
File Handling	Read/write text files	Read/write binary files
Network Communication	Sending/receiving text data	Sending/receiving binary data
Handling Images/Audio/Video	Not designed for handling binary data directly	Suitable for handling binary multimedia data
Text Encoding	Supports various character encodings	No specific text encoding support

UNIT - 3

1. Swing Introducing Swing - Swing Is Built on the AWT- Two Key Swing Features - The MVC Connection - Components and Containers – The Swing Packages - A Simple Swing Application - Exploring Swing.

Swing Is Built on the AWT:

- Swing was created to address the limitations present in the AWT
- Swing also uses the same event handling mechanism as the AWT.

Two Key Swing Features:

The Swing framework in Java introduced two key features to address limitations present in AWT. These features are:

1. Lightweight Components:

Description: Swing components are lightweight, meaning they are written entirely in Java and do not map directly to platform-specific peers. Unlike AWT components, Swing components do not rely on native code for rendering.

Advantages:

- More efficient and flexible.
- Consistent behavior across platforms.
- Determined by Swing, not the underlying operating system.
- Classes: Examples include JButton, JTextField, and other Swing components.

2.Pluggable Look and Feel:

Description: Swing supports a pluggable look and feel, allowing developers to change the appearance of components without affecting their logic. Each Swing component is rendered by Java code rather than native peers, giving control of the look and feel to Swing.

Advantages:

- Separation of the look and feel from the component logic.
- Possibility to plug in new looks and feels without modifying code.
- Consistent appearance across platforms.

Implementation:

- It is possible to plug in or change the look and feel dynamically at runtime.
- Java provides default look and feels like "Metal" and "Nimbus," and it is also possible to design custom look and feels.

3.Examples:

- ✓ **Metal Look and Feel:**
 - a. A platform-independent look and feel.
 - b. Default in Java Swing.
- ✓ **Nimbus Look and Feel:**
 - a. Another default look and feel in Java Swing.
- ✓ **Windows Look and Feel:**
 - a. Specific to Windows environments.
 - b. Provides a native Windows appearance.

4.Dynamic Changes:

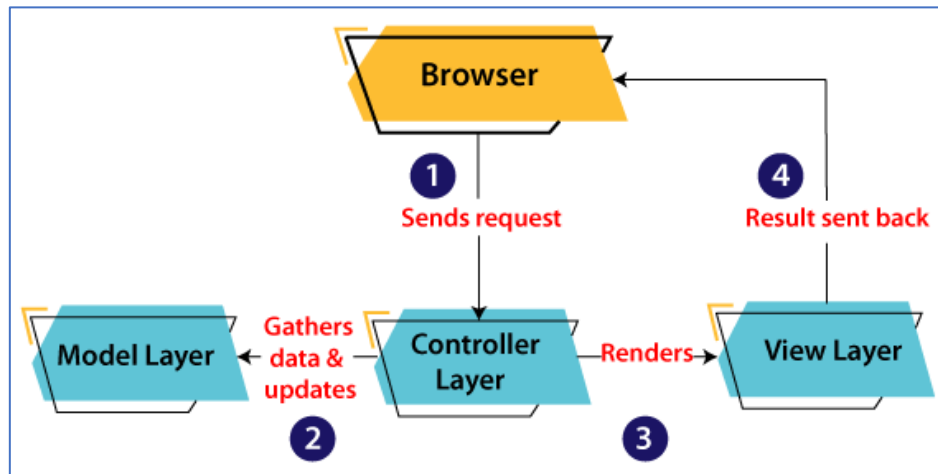
- Developers can dynamically change the look and feel during runtime.
- For example, an application can use the default look and feel on one platform and switch to a Windows-specific look and feel on another platform.

5.Advantages of Pluggable Look and Feel:

- Consistent appearance across different platforms.
- Possibility to define custom looks and feels.
- Separation of concerns between the visual presentation and the logical behavior of components.

In summary, Swing's lightweight components and pluggable look and feel address limitations of AWT, providing a more efficient, flexible, and platform-independent solution for building graphical user interfaces in Java.

The MVC Connection:



The MVC Connection in Swing

The Model-View-Controller (MVC) architecture is a fundamental design pattern used in software development, emphasizing the separation of concerns. In Swing, the graphical user interface (GUI) framework for Java, the MVC architecture is modified to suit the needs of creating interactive and customizable user interfaces. Here, we delve into the key aspects of the MVC connection in Swing, exploring its components and how it facilitates Swing's pluggable look and feel.

1. Overview of MVC in Swing:

- **Model:** Represents the state information associated with a component. For instance, in a checkbox, the model holds information about its checked or unchecked state.
- **View:** Determines the visual representation of the component on the screen, including aspects influenced by the current state of the model.
- **Controller:** Governs the component's response to user actions. For example, a controller reacts to a user clicking a checkbox by updating the underlying model.

2. Modified MVC in Swing:

- Swing introduces a modified version of MVC known as the Model-Delegate architecture or Separable Model architecture.
- In this approach, the view and controller are combined into a single logical entity called the UI delegate.
- The UI delegate separates the look (view) from the feel (controller), allowing for independent customization of appearance and behavior.

3. Pluggable Look and Feel:

- The Separable Model architecture in Swing enables a pluggable look and feel.
- Look and feel can be changed without impacting the component's usage within a program.
- Customizing the model does not affect the component's appearance or user interaction, providing a high level of flexibility.

4. Components in Swing:

Model:

- Represented by interfaces such as ButtonModel for a button.
- Holds the state information specific to the component.

UI Delegate:

- Represented by classes inheriting from ComponentUI (e.g., ButtonUI for a button).
- Defines the visual representation and user interaction of the component.

Programs typically interact with the model, and the UI delegate is managed internally by Swing.

5. Conclusion:

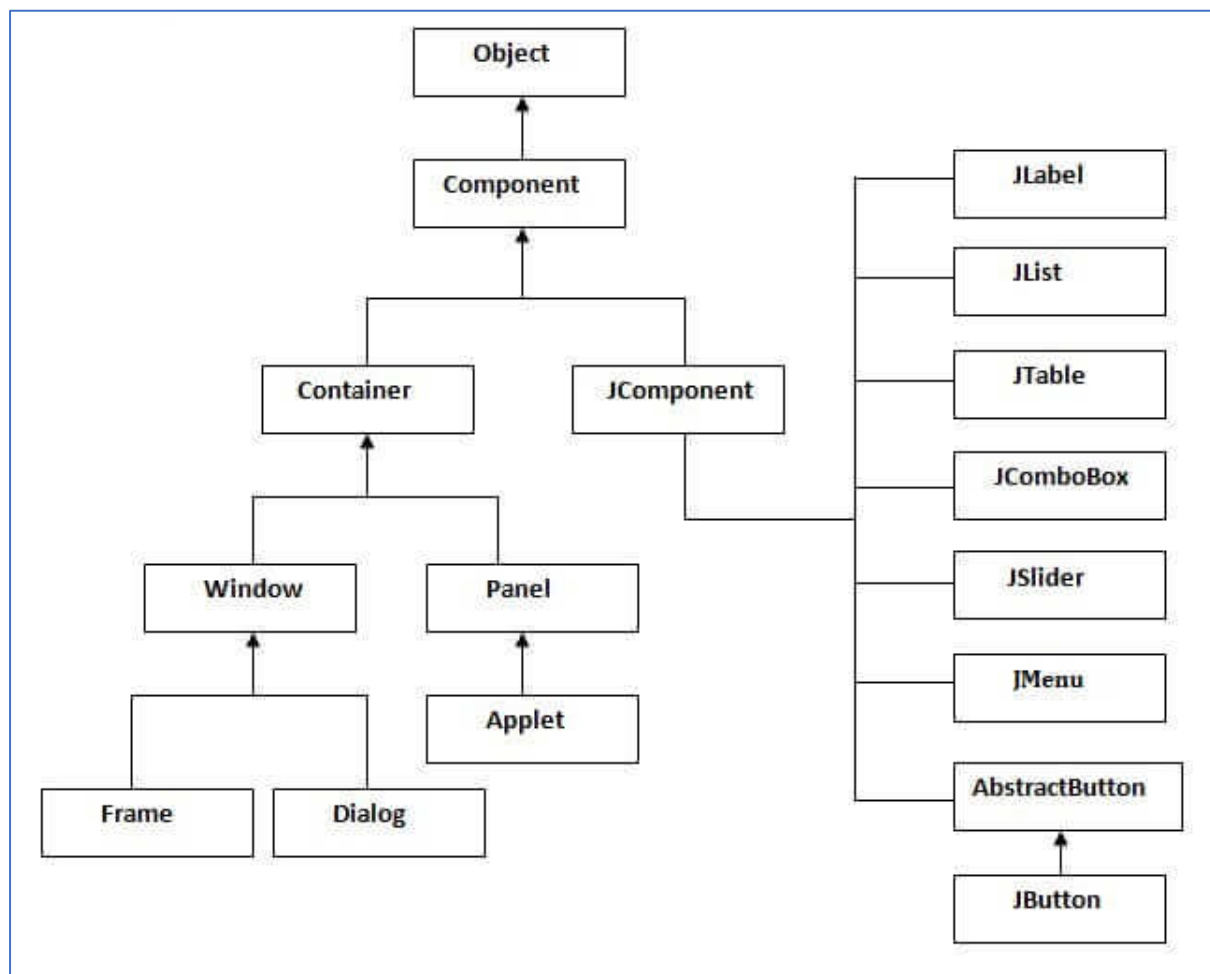
- Swing's approach to MVC provides a robust foundation for creating GUIs in Java.
- The Separable Model architecture facilitates a pluggable look and feel, allowing developers to customize the appearance and behavior of components independently.

In summary, the MVC connection in Swing, with its Model-Delegate architecture, underlines the flexibility and modularity of Swing components, making it a powerful framework for building dynamic and visually appealing user interfaces in Java.

Components and Containers in Swing

Swing, the GUI toolkit for Java, organizes its graphical user interfaces (GUIs) through the concepts of components and containers. Here, we delve into the distinctions between components and containers in Swing.

JApplet	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayer
JLayeredPane	JList	JMenu	JMenuBar
JMenuItem	JOptionPane	JPanel	JPasswordField
JPopupMenu	JProgressBar	JRadioButton	JRadioButtonMenuItem
JRootPane	JScrollBar	JScrollPane	JSeparator
JSlider	JSpinner	JSplitPane	JTabbedPane
JTable	JTextArea	JTextField	JTextPane
JToggleButton	JToolBar	JToolTip	JTree
JViewport	JWindow		



1. Components and Containers Overview:

- **Component:** Represents an independent visual control (e.g., push button, slider).
- **Container:** Serves as a special type of component designed to hold a group of components.
- All containers are components, but their intended purposes differ.

2. Components in Swing:

- Swing components are primarily derived from the JComponent class, supporting common functionality.
- JComponent is based on AWT classes Container and Component, ensuring compatibility.
- Examples of Swing component classes include JButton, JLabel, and JScrollPane.

3. Containers in Swing:

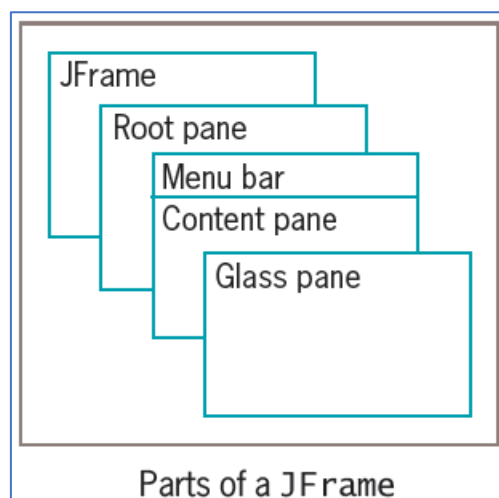
Top-Level Containers: Include JFrame, JApplet, JWindow, and JDialog.

- Not derived from JComponent.
- Considered heavyweight and serve as the top of the containment hierarchy.
- Every containment hierarchy must begin with a top-level container.

Lightweight Containers: Examples include JPanel.

- Derived from JComponent.
- Used for organizing and managing groups of related components within an outer container.

4. Top-Level Container Panes:



Each top-level container defines a set of panes, managed by an instance of `JRootPane`.

- **Glass Pane:** Positioned above and covers all other panes, often transparent and used for managing global mouse events.
- **Layered Pane:** An instance of `JLayeredPane`, allowing components to have a depth value, determining overlay order.
- **Content Pane:** The main container for visual components, usually an opaque instance of `JPanel`.
- Components are added to the content pane by default.

5. Purpose of Glass Pane and Layered Pane:

- **Glass Pane:** Manages mouse events affecting the entire container and can paint over other components.
- **Layered Pane:** Allows components to be given a depth value, specifying overlay order (Z-order).

6. Content Pane:


- The primary pane for adding visual components to a top-level container.
- By default, an opaque instance of `JPanel`.

Understanding the distinctions between components and containers in Swing is crucial for effective GUI design. Components represent individual visual elements, while containers organize and hold groups of components, forming the structure of the GUI. Top-level containers and lightweight containers each play specific roles in creating dynamic and responsive Swing applications.

JButton:

- Represents a button.
- Constructor Example:

java


 Copy code

```
JButton bt1 = new JButton("Yes");  
JButton bt2 = new JButton("No");
```

.JTextField:

- Takes input of a single line of text.
- Constructor Example:

java


 Copy code

```
JTextField jtf = new JTextField(20);
```

JComboBox:

- Creates a combo box (text field + drop-down list).
- Example:

java

 Copy code

```
String name[] = {"Abhi", "Adam", "Alex", "Ashkay"};  
JComboBox jc = new JComboBox(name);
```

Examples:


Change Background Color:

- Program to change the background color of a frame using `ActionEvent`.

JLabel:

- Displays text in a box.
- Example:

java


 Copy code

```
JLabel label_11 = new JLabel("Welcome to studytonight.com");
```

JTextArea:

- Displays multiple lines of text.
- Example:

java


 Copy code

```
JTextArea textArea_area = new JTextArea("Welcome to studytonight.com ");
```

JPasswordField:

- Used for password input.
- Example:

java


 Copy code

```
JPasswordField passWord_value = new JPasswordField();
```

JCheckBox:

- Creates checkboxes for user response.
- Example:

java


 Copy code

```
JCheckBox jcb = new JCheckBox("yes");
```

JRadioButton:

- Represents a group of related buttons.
- Example:

java


 Copy code

```
JRadioButton jcb = new JRadioButton("A");
```

JTable:

- Displays data in tabular form.
- Example:

java


 Copy code

```
String table_data[][] = {{ "1001", "Cherry"}, { "1002", "Candy"}, { "1003", "C
```

JList:

- Represents a list of items.
- Example:


java

 Copy code

```
DefaultListModel<String> list_l1 = new DefaultListModel<>();  
list_l1.addElement("Red");
```

Conclusion:

Understanding Swing components and containers is crucial for building interactive and user-friendly Java GUI applications. These elements provide a rich set of functionalities for creating diverse graphical interfaces.

 Regenerate

The Swing Packages

Swing is a very large subsystem and makes use of many packages. At the time of this writing, these are the packages defined by Swing.

javax.swing	javax.swing.plaf.basic	javax.swing.text
javax.swing.border	javax.swing.plaf.metal	javax.swing.text.html
javax.swing.colorchooser	javax.swing.plaf.multi	javax.swing.text.html.parser
javax.swing.event	javax.swing.plaf.nimbus	javax.swing.text.rtf
javax.swing.filechooser	javax.swing.plaf.synth	javax.swing.tree
javax.swing.plaf	javax.swing.table	javax.swing.undo

The main package is **javax.swing**. This package must be imported into any program that uses Swing. It contains the classes that implement the basic Swing components, such as push buttons, labels, and check boxes.

A Simple Swing Application:

This example demonstrates a simple Swing application that creates a JFrame and adds a JLabel to it. The JFrame contains a label with the text "Swing Metric Conversion App GUIs."

```
package swing_application;
```

```
import javax.swing.*;
```

```
import java.awt.*;
```

```
public class MetricConversionApp extends JFrame {  
    private JTextField cmTextField, inchTextField;
```

```
    public MetricConversionApp() {  
        setTitle("Metric Conversion");  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        setLayout(new FlowLayout());
```

```
        cmTextField = new JTextField(10);  
        inchTextField = new JTextField(10);  
        JButton convertButton = new JButton("Convert");
```

```
        convertButton.addActionListener(e -> {  
            try {  
                double inches = Double.parseDouble(cmTextField.getText()) / 2.54;  
                inchTextField.setText(String.format("%.2f", inches));  
            }  
        });  
    }
```

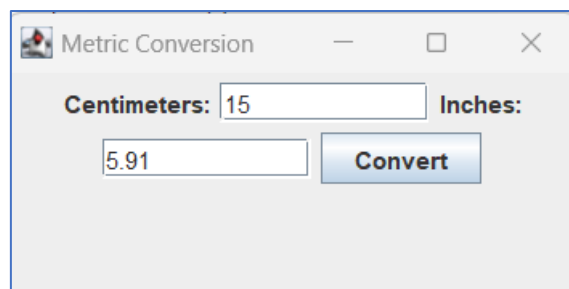
```

        catch (NumberFormatException ex) {
            JOptionPane.showMessageDialog(null, "Invalid input. Please enter a
valid number.");
        }
    });

    add(new JLabel("Centimeters:"));
    add(cmTextField);
    add(new JLabel("Inches:"));
    add(inchTextField);
    add(convertButton);
    setSize(300, 150);
    setVisible(true);
}

public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> new MetricConversionApp());
}
}

```



This program creates a JFrame, sets its size, sets the default close operation, creates a JLabel, adds the label to the content pane of the frame, and finally, makes the frame visible.

Exploring Swing.

JButton	JCheckBox	JComboBox	JLabel
JList	JRadioButton	JScrollPane	JTabbedPane
JTable	TextField	JToggleButton	JTree

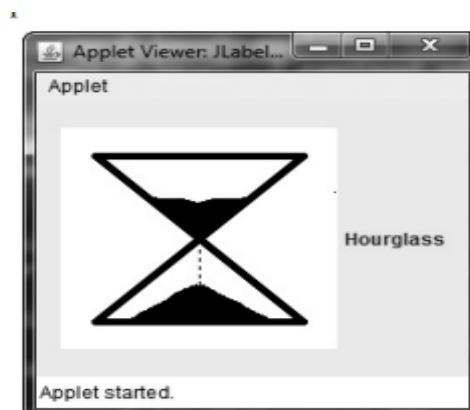
Example :

```
import javax.swing.*;
import javax.swing.tree.DefaultMutableTreeNode;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class SwingComponentsExample {
    public static void main(String[] args) {
        SwingUtilities.invokeLater() -> createAndShowGUI();
    }

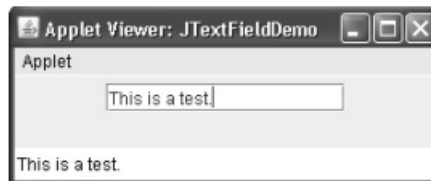
    private static void createAndShowGUI() {
        JFrame frame = new JFrame("Swing Components Example");
        frame.setSize(500, 400);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // JLabel and ImageIcon
        ImageIcon icon = new ImageIcon("hourglass.png");
        JLabel label = new JLabel("Hourglass", icon, JLabel.CENTER);
    }
}
```



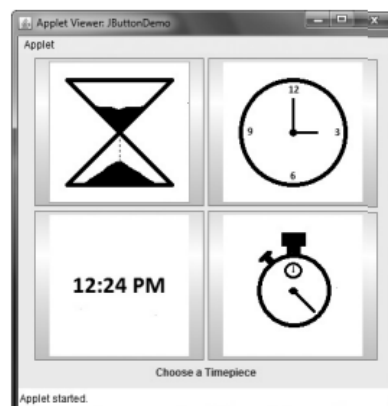
// JTextField

```
JTextField textField = new JTextField(15);
textField.addActionListener(e ->
    JOptionPane.showMessageDialog(frame, "Entered Text: " +
textField.getText()));
```



// JButton

```
JButton button = new JButton("Click Me");
button.addActionListener(e ->
    JOptionPane.showMessageDialog(frame, "Button Clicked!"));
```



// JToggleButton

```
JToggleButton toggleButton = new JToggleButton("Toggle Me");
toggleButton.addActionListener(e -> {
    boolean isSelected = toggleButton.isSelected();
    JOptionPane.showMessageDialog(frame, "Toggle Button is " +
(isSelected ? "selected" : "not selected"));
});
```



// JCheckBox

```
JCheckBox checkBox = new JCheckBox("Check Me");  
checkBox.addActionListener(e ->  
    JOptionPane.showMessageDialog(frame, "Checkbox is " +  
    (checkBox.isSelected() ? "checked" : "unchecked"))));
```



// JRadioButton

```
JRadioButton radioButton = new JRadioButton("Radio Me");  
radioButton.addActionListener(e ->  
    JOptionPane.showMessageDialog(frame, "Radio Button is " +  
    (radioButton.isSelected() ? "selected" : "not selected"))));
```



// JTabbedPane

```
JTabbedPane tabbedPane = new JTabbedPane();  
tabbedPane.addTab("Tab 1", new JLabel("Content for Tab 1"));  
tabbedPane.addTab("Tab 2", new JLabel("Content for Tab 2"));
```



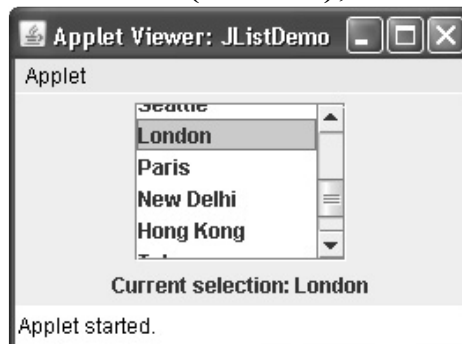
// JScrollPane

```
JTextArea textArea = new JTextArea("This is a JTextArea");  
JScrollPane scrollPane = new JScrollPane(textArea);
```



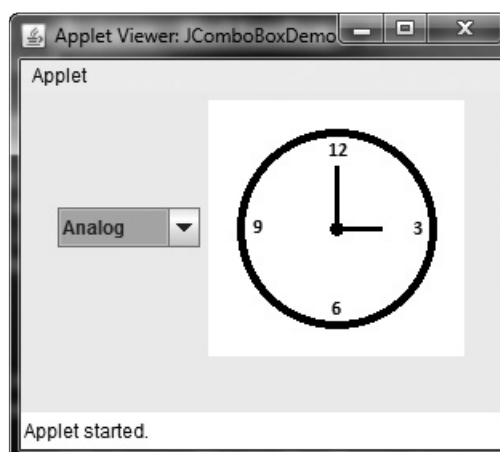
// JList

```
String[] listData = {"Item 1", "Item 2", "Item 3"};  
JList<String> list = new JList<>(listData);
```



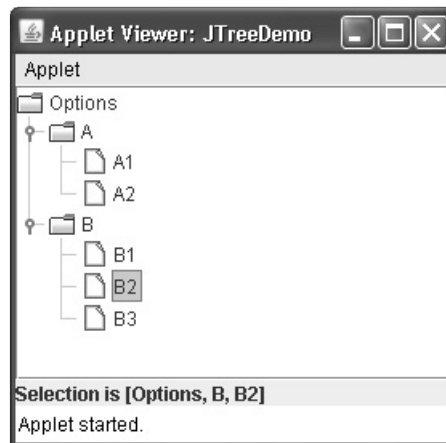
// JComboBox

```
String[] comboData = {"Option 1", "Option 2", "Option 3"};  
JComboBox<String> comboBox = new JComboBox<>(comboData);
```



// JTree

```
DefaultMutableTreeNode rootNode = new  
DefaultMutableTreeNode("Root");  
JTree tree = new JTree(rootNode);
```



// JTable

```
Object[][] tableData = {"Row 1, Col 1", "Row 1, Col 2"}, {"Row 2, Col  
1", "Row 2, Col 2"};  
Object[] columnNames = {"Column 1", "Column 2"};  
JTable table = new JTable(tableData, columnNames);
```

Name	Extension	ID#
Gail	4567	865
Ken	7566	555
Viviane	5634	587
Melanie	7345	922
Anne	1237	333
John	5656	314
Matt	5672	217
Claire	6741	444
Erwin	9023	519

// Layout

```
frame.setLayout(new FlowLayout());  
frame.add(label);  
frame.add(textField);  
frame.add(button);  
frame.add(toggleButton);  
frame.add(checkBox);  
frame.add(radioButton);
```

```
    frame.add(tabbedPane);
    frame.add(scrollPane);
    frame.add(list);
    frame.add(comboBox);
    frame.add(tree);
    frame.add(new JScrollPane(table));

    // Make the frame visible
    frame.setVisible(true);
}
```

UNIT – IV

1. **Java Beans: Introduction - Advantages of Beans Introspection – The JavaBeans API - A Bean Example.**
2. **Servlets: Life Cycle Simple Servlet-Servlet PI-Packages-Cookies session tracking.**

Java Beans: Introduction

- A *Java Bean* is a software component that has been designed to be reusable in a variety of different environments.
- It may perform a simple function, such as obtaining an inventory value, or a complex function, such as forecasting the performance of a stock portfolio.
- JavaBeans is a portable, platform-independent model written in Java Programming Language.
- Its components are referred to as beans. In simple terms, JavaBeans are classes which encapsulate several objects into a single object.

Advantages of Java Beans

- A Bean obtains all the benefits of Java's "write-once, run-anywhere" paradigm.
- The properties, events, and methods of a Bean that are exposed to another application can be controlled.
- Auxiliary software can be provided to help configure a Bean. This software is only needed when the design-time parameters for that component are being set.
- It does not need to be included in the run-time environment.
- The state of a Bean can be saved in persistent storage and restored at a later time.
- A Bean may register to receive events from other objects and can generate events that are sent to other objects.

Introspection:

- This is the process of analyzing a Bean to determine its capabilities. This is an essential feature of the Java Beans API because it allows another application, such as a design tool, to obtain information about a component. Without introspection, the Java Beans technology could not operate.
- There are two ways in which the developer of a Bean can indicate which of its properties, events, and methods should be exposed.
- With the first method, simple naming conventions are used. These allow the introspection mechanisms to infer information about a Bean.
- In the second way, an additional class that extends the **BeanInfo** interface is provided that explicitly supplies this information.

Design Patterns for Properties:

- A *property* is a subset of a Bean's state. The values assigned to the properties determine the behaviour and appearance of that component. A property is set through a *setter* method.
- A property is obtained by a *getter* method. There are two types of properties: simple and indexed

Simple Properties

- A simple property has a single value. It can be identified by the following design patterns, where **N** is the name of the property and **T** is its type:

```
public T getN( )  
public void setN(T arg)
```

- A read/write property has both of these methods to access its values. A read-only property has only a get method. A write-only property has only a set method.

```

private double depth, height, width;

public double getDepth( ) {
    return depth;
}
public void setDepth(double d) {
    depth = d;
}

public double getHeight( ) {
    return height;
}
public void setHeight(double h) {
    height = h;
}

public double getWidth( ) {
    return width;
}
public void setWidth(double w) {
    width = w;
}

```

Indexed Properties

- An indexed property consists of multiple values. It can be identified by the following design patterns, where N is the name of the property and T is its type:

```

public T getN(int index);
public void setN(int index, T value);
public T[ ] getN( );
public void setN(T values[ ]);

```

```

private double data[ ];

public double getData(int index) {
    return data[index];
}
public void setData(int index, double value) {
    data[index] = value;
}
public double[ ] getData( ) {
    return data;
}
public void setData(double[ ] values) {
    data = new double[values.length];
    System.arraycopy(values, 0, data, 0, values.length);
}

```


Design Patterns for Events

- Beans can generate events and send them to other objects. These can be identified by the following design patterns, where **T** is the type of the event:

```
public void addTListener(TListener eventListener)
public void addTListener(TListener eventListener)
throws java.util.TooManyListenersException
public void removeTListener(TListener eventListener)
```

- These methods are used to add or remove a listener for the specified event. The version of **addTListener()** that does not throw an exception can be used to *multicast* an event, which means that more than one listener can register for the event notification. The version that throws **TooManyListenersException** *unicasts* the event, which means that the number of listeners can be restricted to one. In either case, **removeTListener()** is used to remove the listener.

```
public void addTemperatureListener(TemperatureListener tl)
{
...
}
public void removeTemperatureListener(TemperatureListener tl) {
...
}
```

Methods and Design Patterns

- Design patterns are not used for naming nonproperty methods. The introspection mechanism finds all of the public methods of a Bean. Protected and private methods are not presented.

Using the BeanInfo Interface

- As the preceding discussion shows, design patterns *implicitly* determine what information is available to the user of a Bean. The **BeanInfo** interface enables you to *explicitly* control what information is available.
- The **BeanInfo** interface defines several methods, including these:

```
PropertyDescriptor[ ] getPropertyDescriptors( )  
EventSetDescriptor[ ] getEventSetDescriptors( )  
MethodDescriptor[ ] getMethodDescriptors( )
```

- They return arrays of objects that provide information about the properties, events, and methods of a Bean. The classes **PropertyDescriptor**, **EventSetDescriptor**, and **MethodDescriptor** are defined within the **java.beans** package, and they describe the indicated elements. By implementing these methods, a developer can designate exactly what is presented to a user, bypassing introspection based on design patterns.
- When creating a class that implements **BeanInfo**, you must call that class *bname*BeanInfo, where *bname* is the name of the Bean. For example, if the Bean is called **MyBean**, then the information class must be called **MyBeanBeanInfo**.
-

Bound and Constrained Properties

- A Bean that has a *bound* property generates an event when the property is changed. The event is of type **PropertyChangeEvent** and is sent to objects that previously registered an interest in receiving such notifications. A class that handles this event must implement the **PropertyChangeListener** interface.
- A Bean that has a *constrained* property generates an event when an attempt is made to change its value. It also generates an event of type **PropertyChangeEvent**. It too is sent to objects that previously registered an interest in receiving such notifications. However, those other objects have the ability to veto the proposed change by throwing a **PropertyVetoException**. This capability allows a Bean to operate differently according to its run-time environment. A class that handles this event must implement the **VetoableChangeListener** interface.

Persistence

- *Persistence* is the ability to save the current state of a Bean, including the values of a Bean's properties and instance variables, to nonvolatile storage and to retrieve them at a later time.
- The object serialization capabilities provided by the Java class libraries are used to provide persistence for Beans.
- The easiest way to serialize a Bean is to have it implement the **java.io.Serializable** interface, which is simply a marker interface. Implementing **java.io.Serializable** makes serialization automatic. Your Bean need take no other action.
- Automatic serialization can also be inherited. Therefore, if any superclass of a Bean implements **java.io.Serializable**, then automatic serialization is obtained.
- When using automatic serialization, you can selectively prevent a field from being saved through the use of the **transient** keyword. Thus, data members of a Bean specified as **transient** will not be serialized.
- If a Bean does not implement **java.io.Serializable**, you must provide serialization yourself, such as by implementing **java.io.Externalizable**. Otherwise, containers cannot save the configuration of your component.

Customizers

- A Bean developer can provide a *customizer* that helps another developer configure the Bean.
- A customizer can provide a step-by-step guide through the process that must be followed to use the component in a specific context. Online documentation can also be provided.
- A Bean developer has great flexibility to develop a customizer that can differentiate his or her product in the marketplace.

Interface	Description
AppletInitializer	Methods in this interface are used to initialize Beans that are also applets.
BeanInfo	This interface allows a designer to specify information about the properties, events, and methods of a Bean.
Customizer	This interface allows a designer to provide a graphical user interface through which a Bean may be configured.
DesignMode	Methods in this interface determine if a Bean is executing in design mode.
ExceptionListener	A method in this interface is invoked when an exception has occurred.
PropertyChangeListener	A method in this interface is invoked when a bound property is changed.
PropertyEditor	Objects that implement this interface allow designers to change and display property values.
VetoableChangeListener	A method in this interface is invoked when a constrained property is changed.
Visibility	Methods in this interface allow a Bean to execute in environments where a graphical user interface is not available.

The Java Beans API

The Java Beans functionality is provided by a set of classes and interfaces in the **java.beans** package. This section provides a brief overview of its contents. Table 37-1 lists the interfaces in **java.beans** and provides a brief description of their functionality. Table 37-2 lists the classes in **java.beans**.

Class	Description
BeanDescriptor	This class provides information about a Bean. It also allows you to associate a customizer with a Bean.
Beans	This class is used to obtain information about a Bean.
DefaultPersistenceDelegate	A concrete subclass of PersistenceDelegate .
Encoder	Encodes the state of a set of Beans. Can be used to write this information to a stream.
EventHandler	Supports dynamic event listener creation.
EventSetDescriptor	Instances of this class describe an event that can be generated by a Bean.
Expression	Encapsulates a call to a method that returns a result.
FeatureDescriptor	This is the superclass of the PropertyDescriptor , EventSetDescriptor , and MethodDescriptor classes, among others.

Table 37-2 The Classes in **java.beans**

IndexedPropertyChangeEvent	A subclass of PropertyChangeEvent that represents a change to an indexed property.
IndexedPropertyDescriptor	Instances of this class describe an indexed property of a Bean.
IntrospectionException	An exception of this type is generated if a problem occurs when analyzing a Bean.
Introspector	This class analyzes a Bean and constructs a BeanInfo object that describes the component.
MethodDescriptor	Instances of this class describe a method of a Bean.
ParameterDescriptor	Instances of this class describe a method parameter.
PersistenceDelegate	Handles the state information of an object.
PropertyChangeEvent	This event is generated when bound or constrained properties are changed. It is sent to objects that registered an interest in these events and that implement either the PropertyChangeListener or VetoableChangeListener interfaces.
PropertyChangeListenerProxy	Extends EventListenerProxy and implements PropertyChangeListener .
PropertyChangeSupport	Beans that support bound properties can use this class to notify PropertyChangeListener objects.
PropertyDescriptor	Instances of this class describe a property of a Bean.
PropertyEditorManager	This class locates a PropertyEditor object for a given type.
PropertyEditorSupport	This class provides functionality that can be used when writing property editors.
PropertyVetoException	An exception of this type is generated if a change to a constrained property is vetoed.
SimpleBeanInfo	This class provides functionality that can be used when writing BeanInfo classes.
Statement	Encapsulates a call to a method.
VetoableChangeListenerProxy	Extends EventListenerProxy and implements VetoableChangeListener .
VetoableChangeSupport	Beans that support constrained properties can use this class to notify VetoableChangeListener objects.
XMLDecoder	Used to read a Bean from an XML document.
XMLEncoder	Used to write a Bean to an XML document.

Table 37-2 The Classes in **java.beans**

Although it is beyond the scope of this chapter to discuss all of the classes, four are of particular interest: **Introspector**, **PropertyDescriptor**, **EventSetDescriptor**, and **MethodDescriptor**. Each is briefly examined here.

Introspector

- The **Introspector** class provides static methods for introspection, allowing you to analyze the properties, events, and methods of a **JavaBean** at runtime.
- One of the key methods is `getBeanInfo(Class<?> bean)`, which returns a **BeanInfo** object containing information about the specified bean.

PropertyDescriptor

- The PropertyDescriptor class describes the characteristics of a JavaBean property.
- It provides methods to manage and describe properties, such as isBound() to determine if a property is bound, isConstrained() to check if it's constrained, and getName() to obtain the name of a property.

EventSetDescriptor

- The EventSetDescriptor class represents a JavaBean event.
- It supports methods to obtain the methods used to add or remove event listeners (getAddListenerMethod() and getRemoveListenerMethod()), as well as methods to manage events.
- getListenerType() allows you to obtain the type of a listener, and getName() provides the name of an event.

MethodDescriptor

- The MethodDescriptor class represents a JavaBean method.
- It provides methods such as getName() to obtain the name of the method.
- getMethod() returns an object of type Method that describes the method.

A Bean Example

```
package javaBean;

import java.io.Serializable;

public class PersonBean implements Serializable {
    private String name;
    private int age;

    public PersonBean() {
        // Default constructor
    }

    public PersonBean(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }
}
```

```

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}

```

JavaBean.java

```

package javaBean;

public class JavaBean {
    public static void main(String[] args) {
        PersonBean person = new PersonBean();
        person.setName("MCA");
        person.setAge(30);

        System.out.println("Name: " + person.getName());
        System.out.println("Age: " + person.getAge());
    }
}

```

```

Name: Elanchezhian M
Age: 22
Address: E/15, Annathanapatty Police Quarters, Salem-02

```

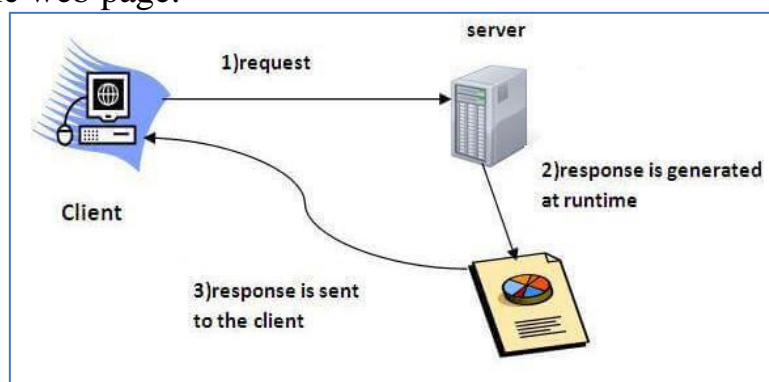

Servlet

- **Servlet** technology is used to create a web application (resides at server side and generates a dynamic web page).
- **Servlet** technology is robust and scalable because of java language. Before Servlet, CGI (Common Gateway Interface) scripting language was common as a server-side programming language. However, there were many disadvantages to this technology. We have discussed these disadvantages below.
- There are many interfaces and classes in the Servlet API such as Servlet, GenericServlet, HttpServlet, ServletRequest, ServletResponse, etc.

What is a Servlet?

Servlet can be described in many ways, depending on the context.

- Servlet is a technology which is used to create a web application.
- Servlet is an API that provides many interfaces and classes including documentation.
- Servlet is an interface that must be implemented for creating any Servlet.
- Servlet is a class that extends the capabilities of the servers and responds to the incoming requests. It can respond to any requests.
- Servlet is a web component that is deployed on the server to create a dynamic web page.

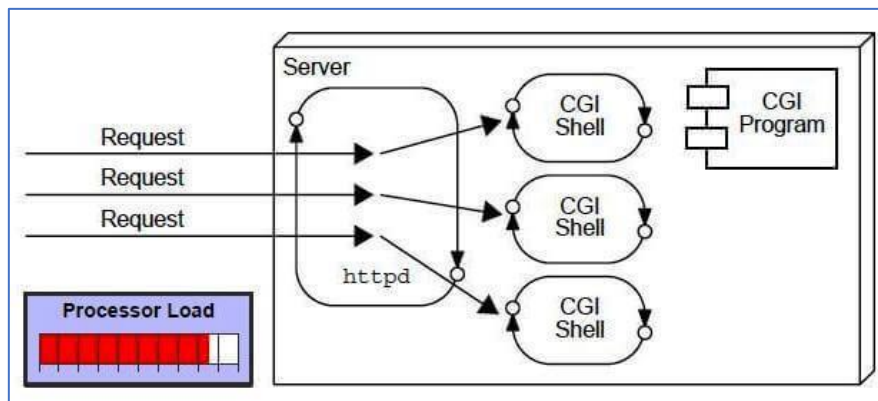


What is a web application?

A web application is an application accessible from the web. A web application is composed of web components like Servlet, JSP, Filter, etc. and other elements such as HTML, CSS, and JavaScript. The web components typically execute in Web Server and respond to the HTTP request.

CGI (Common Gateway Interface)

CGI technology enables the web server to call an external program and pass HTTP request information to the external program to process the request. For each request, it starts a new process.

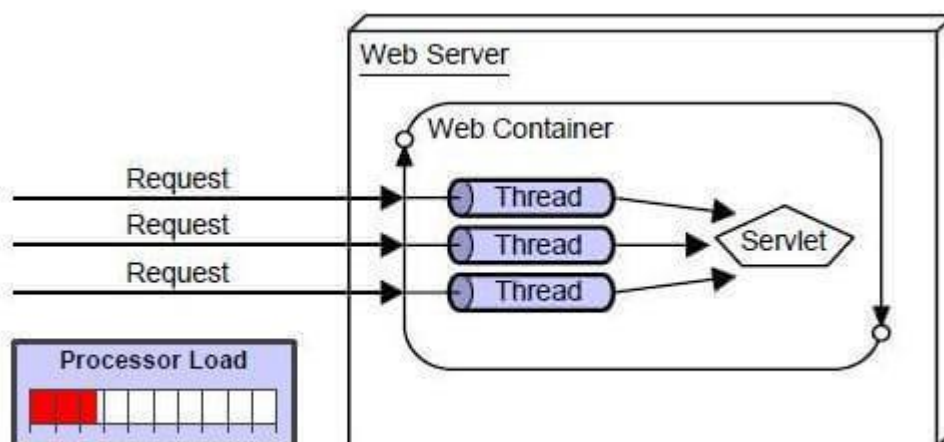


Disadvantages of CGI

There are many problems in CGI technology:

1. If the number of clients increases, it takes more time for sending the response.
2. For each request, it starts a process, and the web server is limited to start processes.
3. It uses platform dependent language e.g. [C](#), [C++](#), [perl](#).

Advantages of Servlet



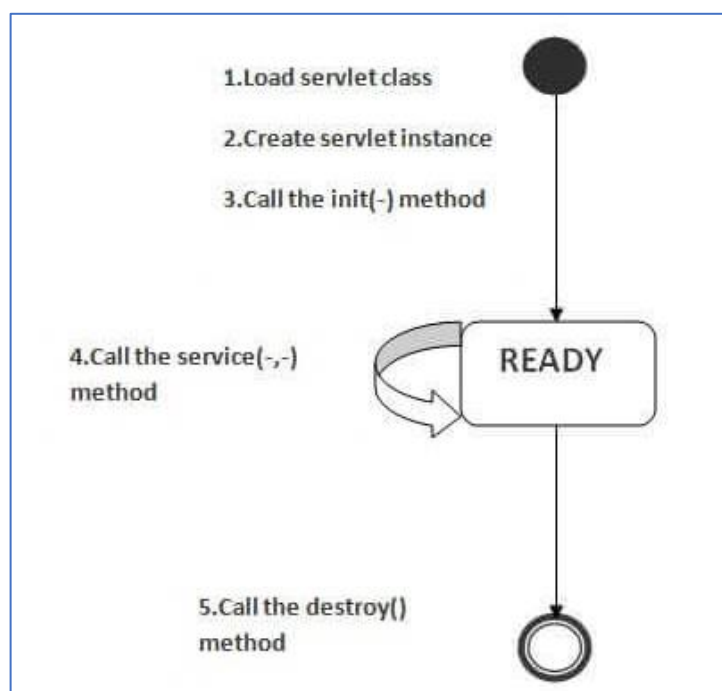
There are many advantages of Servlet over CGI. The web container creates threads for handling the multiple requests to the Servlet. Threads have many benefits over the Processes such as they share a common memory area, lightweight, cost of communication between the threads are low. The advantages of Servlet are as follows:

1. **Better performance:** because it creates a thread for each request, not process.
2. **Portability:** because it uses Java language.
3. **Robust:** JVM manages Servlets, so we don't need to worry about the memory leak, garbage collection, etc.
4. **Secure:** because it uses java language.

Servlet life cycle:

The web container maintains the life cycle of a servlet instance. Let's see the life cycle of the servlet:

1. **Servlet class is loaded.**
2. **Servlet instance is created.**
3. **init method is invoked.**
4. **service method is invoked.**
5. **destroy method is invoked.**



1. Servlet class is loaded:

The servlet class is loaded by the classloader when the first request for the servlet is received.

2. Servlet instance is created:

After loading the servlet class, the web container creates an instance of the servlet. This instance is created only once during the servlet's life cycle.

3. init method is invoked:

The init method is called by the web container after creating the servlet instance. This method is used for servlet initialization and is invoked only once in the servlet's life cycle.

Syntax: public void init(ServletConfig config) throws ServletException

4. service method is invoked:

The service method is called each time a request for the servlet is received. If the servlet is not initialized, it goes through the initialization steps and then calls the service method. If the servlet is already initialized, it directly calls the service method.

Syntax: public void service(ServletRequest request, ServletResponse response) throws ServletException, IOException

5. destroy method is invoked:

The destroy method is called by the web container before removing the servlet instance from service. This method gives the servlet an opportunity to clean up any resources, such as memory or threads.

Syntax: public void destroy()

The servlet life cycle consists of these key steps: loading, instantiation, initialization, request processing, and destruction. Understanding the servlet life cycle is essential for effective servlet development, as it allows developers to manage resources and perform necessary setup and cleanup tasks.

Servlet API

The Servlet API (Application Programming Interface) in Java provides a set of classes and interfaces that define the contract between a servlet class and the runtime environment provided by the web container. Servlets are Java programs that run on a web server and respond to client requests.

Here are some key components of the Servlet API:

javax.servlet package:

- This package contains the core classes and interfaces for servlets.
- Important classes and interfaces include Servlet, GenericServlet, HttpServlet, ServletRequest, ServletResponse, ServletConfig, ServletContext, and more.

Servlet Interface:

- The Servlet interface is the foundation of the servlet API. It declares methods that servlet classes must implement.
- Methods include `init(ServletConfig config)`, `service(ServletRequest req, ServletResponse res)`, and `destroy()`.

GenericServlet Class:

- The GenericServlet class is an abstract class that implements the Servlet interface. It provides a generic, protocol-independent servlet base class.
- Servlets can extend this class and override the necessary methods.

HttpServlet Class:

- The HttpServlet class is an abstract class that extends GenericServlet and provides specific support for HTTP protocol-related features.
- Servlets that handle HTTP requests often extend this class and override methods like `doGet()`, `doPost()`, etc.

ServletRequest and ServletResponse Interfaces:

- ServletRequest represents the client's request to the servlet, providing methods to access parameters, headers, and input streams.
- ServletResponse represents the servlet's response to the client, providing methods to set content type, status, and write output streams.

ServletConfig Interface:

- ServletConfig provides an object through which a servlet can get information about its initialization parameters.
- Servlets can access configuration details defined in the web deployment descriptor (web.xml) through this interface.

ServletContext Interface:

- ServletContext provides a way for servlets to communicate with the web container. It allows servlets to get information about the web application and manage resources.
- Servlets can use the context to obtain parameters, attributes, and resources.

Annotation Support:

- Java EE 5 and later versions introduced annotation-based configuration for servlets. Annotations like `@WebServlet` can be used to define servlet mappings and other configurations directly in the servlet class.

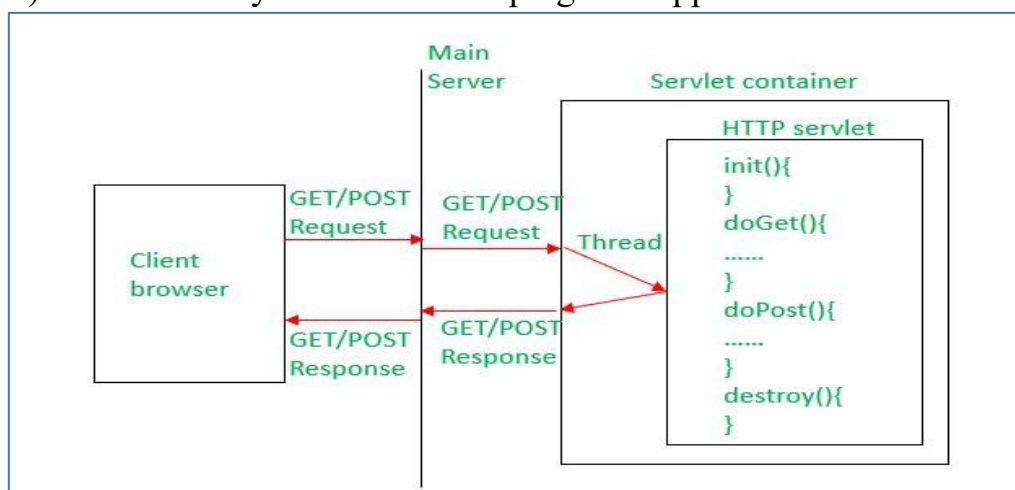
Session Management:

- The Servlet API includes features for session management through the `HttpSession` interface. Sessions allow servlets to maintain user-specific data across multiple requests.

Event Handling:

- Servlets can handle various events related to the lifecycle of the servlet, such as initialization, request processing, and destruction.

The Servlet API provides a robust framework for building dynamic, server-side web applications in Java. Servlets are a key component of Java EE (Enterprise Edition) and are widely used for developing web applications and services.



Cookies:

Cookies are small pieces of data that a web server sends to a user's web browser during their visit to a website. These cookies are stored on the user's device and are sent back to the server with subsequent requests. Cookies are used to track and maintain information about the user's activity on a website. Key points about cookies include:

Purpose: Cookies serve various purposes, such as session management, personalization, tracking user behavior, and storing user preferences.

Content: A cookie typically contains information like a unique identifier, expiration date, and some user-specific data. This data is often encrypted for security purposes.

Types: There are two main types of cookies: session cookies and persistent cookies.

- **Session Cookies:** These are temporary and are stored only during a user's session. They are usually deleted when the browser is closed.
- **Persistent Cookies:** These are stored on the user's device for a longer period. They have an expiration date and remain on the device even after the browser is closed.

Security Concerns: While cookies are widely used, there are privacy and security concerns associated with them. Users can control cookie settings in their browsers, and websites should handle user data responsibly.

HTTP Cookies: Cookies are often referred to as HTTP cookies because they operate within the context of the HTTP protocol used for web communication.

Session Tracking:

Session tracking is a mechanism used by web applications to maintain state information about a user across multiple requests. In a stateless protocol like HTTP, where each request is independent, session tracking helps in associating multiple requests from the same user as part of a single session. Common techniques for session tracking include:

- **Cookies:** As mentioned earlier, cookies can be used for session tracking by storing a session identifier on the user's device. This identifier is sent back to the server with each request, allowing the server to associate requests with the same session.

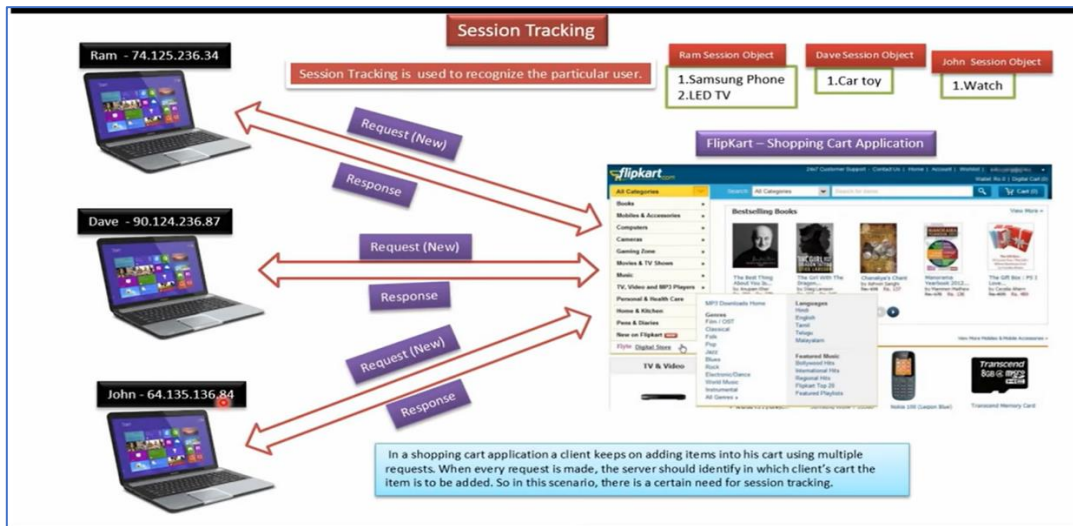
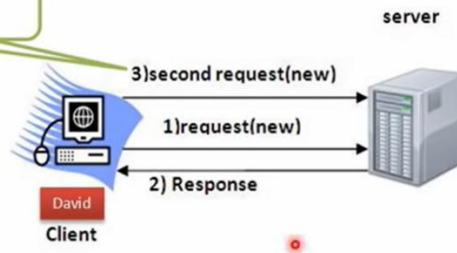
- **URL Rewriting:** Session information can be encoded in URLs as parameters. This is achieved by appending a session ID to URLs, and the server extracts this ID to identify the session.
- **Hidden Form Fields:** Session information can be stored in hidden form fields within HTML forms. When the form is submitted, the session data is sent back to the server.
- **HTTP Session Object:** In Java web applications, the Servlet API provides an HttpSession object that can store session-specific data. This object is stored on the server, and a session ID is used for client-server communication.
- **Security Considerations:** Implementing secure session tracking is crucial to prevent unauthorized access or tampering. This includes using secure connections (HTTPS), encrypting session data, and validating session IDs.

Session tracking is essential for maintaining user-specific information across different pages or requests, enabling features like user authentication, shopping carts in e-commerce sites, and personalized user experiences on websites.

Session Tracking

- ✓ **Session Tracking** is a way to maintain state (data) of an user. It is also known as **session management** in servlet.
- ✓ Session tracking is mechanism of tracking the client provided data and making it available to the next request from the same client. And this process is continued until the user choose to Logout or terminate the session.
- ✓ Http protocol is a stateless so we need to maintain state using session tracking techniques. Each time user requests to the server, server treats the request as the new request. So we need to maintain the state of an user to recognize to particular user.
- ✓ Session tracking is a mechanism that servlets use to maintain state about a series of requests from the same user (that is, requests originating from the same browser) across some period of time.

HTTP is stateless that means each request is considered as the new request.



What is Session Tracking? | Servlets

Session Tracking

Session tracking methods:

- User authorization
- Hidden fields
- URL rewriting
- Cookies
- Session tracking API

The first four methods are traditionally used for session tracking in all the server-side technologies. The session tracking API method is provided by the underlying technology (java servlet or PHP or likewise). Session tracking API is built on top of the first four methods.

Session tracking API

- ✓ Session tracking API is built on top of the below four methods.
 1. User authorization
 2. Hidden fields
 3. URL rewriting
 4. Cookies
- ✓ Lets take the java servlet example. Then, the servlet container manages the session tracking task and the user need not do it explicitly using the java servlets. This is the best of all methods, because all the management and errors related to session tracking will be taken care of by the container itself.

Session Tracking API | Servlets

Session tracking API

- ✓ Every client of the server will be mapped with a `javax.servlet.http.HttpSession` object. Java servlets can use the session object to store and retrieve java objects across the session. Session tracking is at the best when it is implemented using session tracking api.

The diagram shows two clients, Client 1 (labeled 'Ram') and Client 2 (labeled 'David'), each with a computer icon. Client 1 sends a request labeled 'Session ID 1' to a central server icon. The server then points to a cloud labeled 'Session 1' with 'Ram' written inside. Client 2 sends a request labeled 'Session ID 2' to the same server, which points to a cloud labeled 'Session 2' with 'David' written inside.

What are Cookies in servlet? | Servlets

Cookies in Servlet

- ✓ Cookies are the mostly used technology for session tracking. Cookie is a key value pair of information, sent by the server to the browser. This should be saved by the browser in its space in the client computer. Whenever the browser sends a request to that server it sends the cookie along with it. Then the server can identify the client using the cookie.
- ✓ Session tracking is easy to implement and maintain using the cookies. Disadvantage is that, the users can opt to disable cookies using their browser preferences. In such case, the browser will not save the cookie at client computer and session tracking fails.
- ✓ Cookies are small files which are stored on a user's computer. They are designed to hold a modest amount of data specific to a particular client and website.
- ✓ Cookies are small files that websites put on your computer hard disk drive when you first visit. Think of a cookie as an identification card that's uniquely yours. Its job is to notify the site when you've returned.

The diagram shows a client machine (labeled 'User ID = 7456') and a server (labeled 'Server'). The client sends a request labeled '1)request' to the server. The server responds with a response labeled '2)response+cookie'. A callout box points to the client machine saying 'Cookies saved by the browser in the client machine'. The client then sends a request labeled '3)request+cookie' back to the server. The server identifies the user as 'User ID = 7456'.

```
Cookie cookie = new Cookie("userID", "7456");
res.addCookie(cookie);
```

What are Cookies in servlet? | Servlets Cookies in Servlet ServletRequestAttributeListener Introduction 1 Servlets

Advantage of Cookies

1. Simplest technique of maintaining the state.
2. Cookies are maintained at client side.

Disadvantage of Cookies

1. It will not work if cookie is disabled from the browser.
2. Only textual information can be set in Cookie object.

2:45 / 4:10 • What are Cookies

What are Cookies in servlet? | Servlets Cookies in Servlet

Types of Cookie

There are 2 types of cookies in servlets.

1. Non-persistent cookie
2. Persistent cookie

Non-persistent cookie

It is **valid for single session** only. It is removed each time when user closes the browser.

Persistent cookie

It is **valid for multiple session**. It is not removed each time when user closes the browser. It is removed only if user logout or signout.

3:26 / 4:10 • Types of Cookies

Cookie Class

- ✓ `javax.servlet.http.Cookie` class provides the functionality of using cookies. It provides a lot of useful methods for cookies.
- ✓ Creates a cookie, a small amount of information sent by a servlet to a Web browser, saved by the browser, and later sent back to the server. A cookie's value can uniquely identify a client, so cookies are commonly used for session management.
- ✓ A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number
- ✓ The servlet sends cookies to the browser by using the `HttpServletResponse#addCookie` method, which adds fields to HTTP response headers to send cookies to the browser, one at a time.
- ✓ The browser returns cookies to the servlet by adding fields to HTTP request headers. Cookies can be retrieved from a request by using the `HttpServletRequest#getCookies` method. Several cookies might have the same name but different path attributes.

```

graph LR
    Browser[Browser] -- "1) request" --> Server((Server))
    Server -- "2) response+cookie" --> Browser
    Browser -- "3) request+cookie" --> Server
  
```

Constructor Summary

`Cookie(java.lang.String name, java.lang.String value)`
Constructs a cookie with the specified name and value.

Method Summary

java.lang.Object	<code>clone()</code>	Overrides the standard java.lang.Object.clone method to return a copy of this Cookie.
java.lang.String	<code>getComment()</code>	Returns the comment describing the purpose of this cookie, or null if the cookie has no comment.
java.lang.String	<code>getDomain()</code>	Gets the domain name of this Cookie.
int	<code>getMaxAge()</code>	Gets the maximum age in seconds of this Cookie.
java.lang.String	<code>getName()</code>	Returns the name of the cookie.
java.lang.String	<code>getPath()</code>	Returns the path on the server to which the browser returns this cookie.
boolean	<code>getSecure()</code>	Returns true if the browser is sending cookies only over a secure protocol, or false if the browser can send cookies using any protocol.
java.lang.String	<code>getValue()</code>	Gets the current value of this Cookie.

Cookie Class

For adding cookie or getting the value from the cookie, we need some methods provided by other interfaces.

HttpServletResponse interface**addCookie**

`void addCookie(Cookie cookie)`

Adds the specified cookie to the response. This method can be called multiple times to set more than one cookie.

Parameters:

`cookie` - the Cookie to return to the client

HttpServletRequest interface**getCookies**

`Cookie[] getCookies()`

Returns an array containing all of the Cookie objects the client sent with this request. This method returns null if no cookies were sent.

Returns:

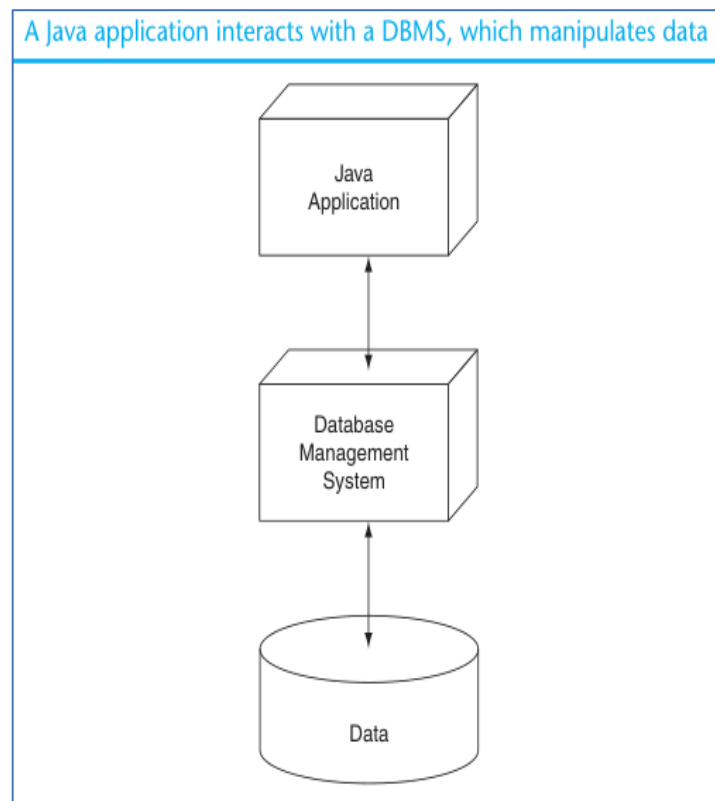
an array of all the Cookies included with this request, or null if the request has no cookies

UNIT-V

2.Introduction to Database Management Systems - Tables, Rows, and Columns - Introduction to the SQL SELECT Statement – Inserting Rows - Updating and Deleting Existing Rows - Creating and Deleting Tables - Creating a New Database with JDBC - Scrollable Result Sets.

Introduction to Database Management Systems:

- An application needs to store only a small amount of data, text and binary files work well. These types of files, however, are not practical when a large amount of data must be stored and manipulated.
- When developing applications that work with an extensive amount of data, most developers prefer to use a database management system.
- A database management system (DBMS) is software specifically designed to store, retrieve, and manipulate large amounts of data in an organized and efficient manner.



- The topmost layer of software, which in this case is written in Java, interacts with the user. It also sends instructions to the next layer of software, the DBMS. The DBMS works directly with the data, and sends the results of operations back to the application.
- For example, suppose that a company keeps all of its product records in a database. The company has a Java application that allows the user to look up information on any product by entering its product ID number.
- The Java application instructs the DBMS to retrieve the record for the product with the specified product ID number. The DBMS retrieves the product record and sends the data back to the Java application. The Java application displays the data to the user.

Connecting to the CoffeeDB Database:

- After installing Java DB and creating the CoffeeDB database, you should attempt to connect to the database with a Java program.
- A program can call the static JDBC method `DriverManager.getConnection` to get a connection to a database. There are overloaded versions of this method, but the simplest one has the following general format:

DriverManager.getConnection(DatabaseURL);

- A simple database URL has the following general format:

protocol:subprotocol:databaseName

- Let's take a closer look at each one.
 1. **protocol** is the database protocol. When using JDBC, protocol will always be `jdbc`.
 2. The value for **subprotocol** will be dependent upon the particular type of DBMS you are connecting to. If you are using Java DB, the subprotocol is `derby`.
 3. **databaseName** is the name of the database you are connecting to.
- If we are using Java DB, the URL for the CoffeeDB database is:

jdbc:derby:CoffeeDB

Tables, Rows, and Columns

- A database management system stores data in a database.
- Your first step in learning to use a DBMS is to learn how data is organized inside a database.
- The data that is stored in a database is organized into one or more tables. Each table holds a collection of related data.
- The data that is stored in a table is then organized into rows and columns. A row is a complete set of information about a single item.
- The data that is stored in a row is divided into columns. Each column is an individual piece of information about the item.

Table 17-1 The Coffee database table

Description	ProdNum	Price
Bolivian Dark	14-001	8.95
Bolivian Medium	14-002	8.95
Brazilian Dark	15-001	7.95
Brazilian Medium	15-002	7.95
Brazilian Decaf	15-003	8.55
Central American Dark	16-001	9.95
Central American Medium	16-002	9.95
Sumatra Dark	17-001	7.95
Sumatra Decaf	17-002	8.95
Sumatra Medium	17-003	7.95

(table continues)

Column Data Types

The columns in a database table are assigned a data type. Notice that the Description and ProdNum columns in the Coffee table hold strings, and the Price column holds floatingpoint numbers. The data types of the columns are not Java data types, however. Instead, they are SQL data types. Table 17-2 lists a few of the standard SQL data types, and shows the Java data type that each is generally compatible with.

Table 17-2 A few of the SQL data types

SQL Data Type	Description	Corresponding Java Data Type
INTEGER or INT	An integer number	int
CHARACTER(<i>n</i>) or CHAR(<i>n</i>)	A fixed-length string with a length of <i>n</i> characters	String
VARCHAR(<i>n</i>)	A variable-length string with a maximum length of <i>n</i> characters	String
REAL	A single-precision floating-point number	float
DOUBLE	A double-precision floating-point number	double
DECIMAL(<i>t</i> , <i>d</i>)	A decimal value with <i>t</i> total digits and <i>d</i> digits appearing after the decimal point	java.math.BigDecimal
DATE	A date	java.sql.Date

Primary Keys

- Most database tables have a primary key, which is a column that can be used to identify a specific row in a table.
- The column that is designated as the primary key holds a unique value for each row. If you try to store duplicate data in the primary key column, an error will occur.

Introduction to the SQL SELECT Statement

The fundamental building block of querying a database is the SELECT statement. Its primary purpose is to retrieve specific rows from a table. As the name implies, SELECT allows for the extraction of targeted data, and we'll begin with a basic structure:

SELECT Columns FROM Table;

For instance:

SELECT Description FROM Coffee;

The process of interacting with a database through SQL involves the following steps:

- **Establish a Database Connection:**
 - Obtain a connection to the database.
- **Send SQL Statement to DBMS:**
 - Pass an SQL statement as a string to the Database Management System (DBMS).
 - If the statement produces results, a ResultSet object is returned.
- **Process ResultSet:**
 - Handle the contents of the ResultSet object, if applicable.
- **Close the Connection:**
 - Conclude the database interaction by closing the connection.

To send an SQL statement, acquire a Statement object from the Connection object:

```
Statement stmt = conn.createStatement();
```

For executing SELECT queries, use the executeQuery method, which returns a ResultSet object:

```
String sqlStatement = "SELECT Description FROM Coffee";  
ResultSet result = stmt.executeQuery(sqlStatement);
```

Specifying Search Criteria with WHERE Clause:

The WHERE clause allows for targeted data retrieval based on specified conditions:

```
SELECT * FROM Coffee WHERE Price > 12.00;
```

String Comparisons and Case Sensitivity:

String comparisons are case-sensitive, but functions like UPPER() and LOWER() can be employed:

```
SELECT * FROM Coffee WHERE UPPER(Description) =  
'FRENCH ROAST DARK';
```


Using the LIKE Operator:

For flexible string matching, use the LIKE operator with wildcard characters:

```
SELECT * FROM Coffee WHERE Description LIKE '%Decaf%';
```

Combining Conditions with AND and OR:

Logical operators AND and OR facilitate combining multiple conditions:

```
SELECT * FROM Coffee WHERE Price > 10.00 AND Price < 14.00;
```

Sorting Results with ORDER BY:

To arrange results in a specific order, utilize the ORDER BY clause:

```
SELECT * FROM Coffee ORDER BY Price;
```

Including WHERE and ORDER BY together:

```
SELECT * FROM Coffee WHERE Price > 9.95 ORDER BY Price;
```

Descending order sorting:

```
SELECT * FROM Coffee WHERE Price > 9.95 ORDER BY Price  
DESC;
```

Mathematical Functions:

Perform calculations on data using SQL functions:

```
SELECT AVG(Price) FROM Coffee;
```

```
SELECT SUM(Price) FROM Coffee;
```

```
SELECT MIN(Price) FROM Coffee;
```

```
SELECT COUNT(*) FROM Coffee;
```

These SQL concepts form the foundation for extracting and manipulating data, providing a robust toolkit for database interactions.

Inserting, Updating, and Deleting Rows in SQL:

Inserting Rows:

- In SQL, the INSERT statement is vital for adding new records to a table. The basic syntax is as follows:
 - INSERT INTO TableName VALUES (Value1, Value2, etc...);
- To specify column names and values explicitly:
 - INSERT INTO TableName (ColumnName1, ColumnName2, etc...) VALUES (Value1, Value2, etc...);
- Example:
 - INSERT INTO Coffee VALUES ('Honduran Dark', '22-001', 8.65);

Inserting Rows with JDBC:

- In JDBC, utilize a Statement object to execute an INSERT statement. The executeUpdate method returns the number of inserted rows.
- Example:
 - String sqlStatement = "INSERT INTO Coffee (ProdNum, Price, Description) VALUES ('22-001', 8.65, 'Honduran Dark')";
 - int rows = stmt.executeUpdate(sqlStatement);

Updating Rows:

- To modify existing records, use the UPDATE statement:
 - UPDATE Table SET Column = Value WHERE Criteria;
- Example:
 - UPDATE Coffee SET Price = 9.95 WHERE Description = 'Brazilian Decaf';

Updating Rows with JDBC:

- Updating rows in JDBC is similar to inserting. Get a Statement object and use executeUpdate. It returns the number of affected rows.
- Example:
 - String sqlStatement = "UPDATE Coffee SET Price = 9.95 WHERE Description = 'Brazilian Decaf'";
 - int rows = stmt.executeUpdate(sqlStatement);

Deleting Rows:

- To remove rows, employ the DELETE statement:
 - `DELETE FROM Table WHERE Criteria;`
- Example:
 - `DELETE FROM Coffee WHERE ProdNum = '20-001';`

Deleting Rows with JDBC:

- Similar to updating, use a Statement object with `executeUpdate` to delete rows. It returns the number of deleted rows.
- Example:
 - `String sqlStatement = "DELETE FROM Coffee WHERE ProdNum = '20-001'";`
 - `int rows = stmt.executeUpdate(sqlStatement);`

Understanding these SQL operations and their JDBC counterparts is crucial for effective data manipulation and maintenance within a relational database system.

Creating and Deleting Tables in SQL:

Creating Tables:

In SQL, the `CREATE TABLE` statement is used to define a new table. The basic syntax is as follows:

```
CREATE TABLE TableName (  
    Column1 DataType,  
    Column2 DataType,  
    ...  
);
```

Example:

```
CREATE TABLE Coffee (  
    ProdNum VARCHAR(10),  
    Description VARCHAR(255),  
    Price DECIMAL(5, 2)  
);
```

Creating Tables with Constraints:

You can add constraints to enforce rules on the table. For instance, to set a primary key:

```
CREATE TABLE Coffee (  
    ProdNum VARCHAR(10) PRIMARY KEY,  
    Description VARCHAR(255),  
    Price DECIMAL(5, 2)  
);
```

Deleting Tables:

To remove an existing table, use the DROP TABLE statement:

```
DROP TABLE TableName;
```

Example:

```
DROP TABLE Coffee;
```

Deleting Tables with Conditions:

You can use conditions to drop a table only if it exists:

```
DROP TABLE IF EXISTS TableName;
```

Example:

```
DROP TABLE IF EXISTS Coffee;
```

Understanding these SQL statements is essential for database schema management, allowing you to create and delete tables to organize and maintain your data effectively. Always exercise caution when using DROP TABLE, as it permanently removes the entire table and its data.

Creating a new Database with JDBC:

Java Database Connectivity (JDBC) is a Java-based API that allows Java applications to interact with relational databases. It provides a standard interface for connecting to databases, sending SQL queries, and processing the results. Here's a basic illustration of how JDBC works, along with an explanation of how SQL statements are written and executed in Java:

In JDBC, creating a new database involves the following steps:

1. Establish a Connection to the Database Server:

- a. Connect to the server where you want to create the new database. Typically, this involves connecting to a specific database management system (DBMS) such as MySQL, PostgreSQL, or Oracle.

2. Create a Statement Object:

- a. Obtain a Statement object from the Connection to execute SQL statements.

3. Execute the SQL Statement to Create the Database:

- a. Use the executeUpdate method on the Statement object to execute the SQL statement that creates the new database.

4. Close the Connection:

- a. Conclude the database interaction by closing the connection.

Here is an example in Java using JDBC and MySQL:

```
import java.sql.*;
public class MyJdbc {
    public static void main(String[] args) {
        String jdbcUrl = "jdbc:mysql://localhost:3306/Mca"; // Update the
        database name if necessary
        String username = "root";
        String password = "Elan@27";
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            try (Connection con = DriverManager.getConnection(jdbcUrl,
                username, password);
                Statement stmt = con.createStatement();
                ResultSet rs = stmt.executeQuery("SELECT * FROM
                emp"))
            {
```

```

while (rs.next()) {
    System.out.println(rs.getInt("Id") + " " + rs.getString("Name") + "
    " + rs.getInt("Age"));
}
}
} catch (Exception e) {
    System.out.println(e);
}
}
}

```

```

C:\Users\Dell>mysql -u root -p
Enter password: *****(Mca@30)
mysql> CREATE DATABASE Mca;
Query OK, 1 row affected (0.01 sec)
mysql> use Mca;
Database changed
mysql> CREATE TABLE emp(id INT, name VARCHAR(40), age
INT);
Query OK, 0 rows affected (0.02 sec)
mysql> INSERT INTO emp (id, name, age) VALUES (1,
'Elanchezhian M', 21);
Query OK, 1 row affected (0.00 sec)
mysql> exit
Bye

```

Output:

Id:1 Name: Elanchezhian M Age:21

In this example:

- Change the url, user, and password variables to match your MySQL server configuration.
- The CREATE DATABASE IF NOT EXISTS statement ensures that the database is created only if it does not already exist.

Remember to include the JDBC driver for your specific database in your project's dependencies. The example uses MySQL, so you would need the MySQL JDBC driver.

This code can serve as a starting point for creating a new database with JDBC, and you can adapt it to suit the specifics of your database server and authentication credentials.

Scrollable Result Sets in JDBC:

In JDBC, a scrollable result set allows you to navigate back and forth through the rows of a query result. This is particularly useful when you want to traverse the result set in a non-sequential manner, such as moving both forward and backward.

Here are the steps to create a scrollable result set in JDBC:

1. Establish a Connection to the Database:

- a. Connect to the database as you would in a regular JDBC connection.

2. Create a Statement Object with Scrollability:

- a. When creating a Statement object, specify the scrollability type using the `ResultSet.TYPE_SCROLL_INSENSITIVE` or `ResultSet.TYPE_SCROLL_SENSITIVE` constant.

3. Execute the Query:

- a. Use the `executeQuery` method on the Statement object to execute the SQL query.

4. Navigate through the Result Set:

- a. Use methods like `next()`, `previous()`, `first()`, `last()`, `absolute()`, or `relative()` to move the cursor within the result set.

ResultSet Navigation Methods

Once you have created a scrollable result set, you can use the following `ResultSet` methods to move the cursor:

<code>first()</code>	Moves the cursor to the first row in the result set.
<code>last()</code>	Moves the cursor to the last row in the result set.
<code>next()</code>	Moves the cursor to the next row in the result set.
<code>previous()</code>	Moves the cursor to the previous row in the result set.
<code>relative(rows)</code>	Moves the cursor the number of rows specified by the argument <i>rows</i> , relative to the current row. For example, the call <code>relative(2)</code> will move the cursor 2 rows forward from the current row, and <code>relative(-1)</code> will move the cursor 1 row backward from the current row.
<code>absolute(row)</code>	Moves the cursor to the row specified by the integer <i>row</i> . Remember, row numbering begins at 1, so the call <code>absolute(1)</code> will move the cursor to the first row in the result set.

The following code shows a simple, yet practical use of some of these methods:

```
resultSet.last();           // Move to the last row
int numRows = resultSet.getRow(); // Get the current row number
resultSet.first();          // Move back to the first row
```

Here is an example in Java:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class ScrollableResultSetExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/yourDatabase";
        String user = "yourUsername";
        String password = "yourPassword";

        try (Connection conn = DriverManager.getConnection(url, user,
password);
            Statement stmt =
conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY)) {

            String query = "SELECT * FROM yourTable";
            ResultSet resultSet = stmt.executeQuery(query);

            // Move to the last row
            resultSet.last();

            // Print data from the last row
            System.out.println("Last Row: " +
resultSet.getString("columnName"));

            // Move to the first row
            resultSet.first();
```



```

        // Print data from the first row
        System.out.println("First Row: " +
resultSet.getString("columnName"));

        // Move to the third row
        resultSet.absolute(3);

        // Print data from the third row
        System.out.println("Third Row: " +
resultSet.getString("columnName"));

    } catch (SQLException e) {
        e.printStackTrace();
    }
}
}

```

Output:

Last Row: [data from the last row]
 First Row: [data from the first row]
 Third Row: [data from the third row]

In this example:

- Replace yourDatabase, yourUsername, yourPassword, and yourTable with your actual database details.
- The `ResultSet.TYPE_SCROLL_INSENSITIVE` constant creates a scrollable result set that is insensitive to changes made by others during its lifetime.
- Various methods are used to move the cursor to different positions within the result set.

Remember to include the JDBC driver for your specific database in your project's dependencies.

Result Set Metadata in JDBC:

Result Set Metadata provides information about the structure and characteristics of the result set returned by a query. It includes details such as the number of columns, column names, data types, and other properties of the result set. JDBC provides the `ResultSetMetaData` interface to access this information.

Table 17-6 A few `ResultSetMetaData` methods

Method	Description
<code>int getColumnCount()</code>	Returns the number of columns in the result set.
<code>String getColumnName(int col)</code>	Returns the name of the column specified by the integer <code>col</code> . The first column is column 1.
<code>String getColumnType(int col)</code>	Returns the name of the data type of the column specified by the integer <code>col</code> . The first column is column 1. The data type name returned is the database-specific SQL data type.
<code>int getColumnDisplaySize(int col)</code>	Returns the display width, in characters, of the column specified by the integer <code>col</code> . The first column is column 1.
<code>String getTableName(int col)</code>	Returns the name of the table associated with the column specified by the integer <code>col</code> . The first column is column 1.

Here's how you can retrieve and use Result Set Metadata in JDBC:

1. Execute a Query:

- Start by executing a query using a `Statement` object.

2. Retrieve Result Set Metadata:

- Obtain the `ResultSetMetaData` object from the `ResultSet` using the `getMetaData()` method.

3. Access Metadata Information:

- Use methods of `ResultSetMetaData` to retrieve various details about the result set, such as column names, data types, and column count.

Here's an example in Java:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;

public class ResultSetMetadataExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/yourDatabase";
        String user = "yourUsername";
        String password = "yourPassword";
```

```

try (Connection conn = DriverManager.getConnection(url, user, password);
    Statement stmt = conn.createStatement()) {

    String query = "SELECT * FROM yourTable";
    ResultSet resultSet = stmt.executeQuery(query);

    // Get ResultSetMetaData
    ResultSetMetaData metaData = resultSet.getMetaData();

    // Retrieve information about the result set
    int columnCount = metaData.getColumnCount();
    System.out.println("Number of columns: " + columnCount);

    // Print column names and data types
    for (int i = 1; i <= columnCount; i++) {
        System.out.println("Column Name: " +
            metaData.getColumnName(i));
        System.out.println("Data Type: " +
            metaData.getColumnTypeName(i));
        System.out.println("-----");
    }

} catch (SQLException e) {
    e.printStackTrace();
}
}

```

Output:

```

Number of columns: n
Column Name: column1
Data Type: datatype1
-----
Column Name: column2
Data Type: datatype2
-----
...
Column Name: columnn
Data Type: datatype_n
-----

```

In this example:

- Replace yourDatabase, yourUsername, yourPassword, and yourTable with your actual database details.
- The ResultSetMetaData object, obtained using getMetaData(), provides information about the result set.
- The loop prints details for each column, such as name and data type.

Understanding Result Set Metadata is crucial for dynamic handling of query results, especially when the structure of the result set is not known beforehand. It allows you to programmatically adapt to different result set characteristics.