Basic Of Java

Package Statement:-The package statement should be at the very top of each file, defining the namespace of your class.

Example:-
```
package com.example.application;
class{

}
```

Import Statements:-After the package statement, you add import statements for all the classes and libraries needed for the functionality.

Example:-
```
import java.util.List;
Class{
p.s.v.m(){
List li=new ArrayList();
}
}
```

Class Definition:-This defines a class, which can be a main class.

Example;- class{

}

Main Method:-The entry point of the application. The main method is typically found in the main application class.

Example:-
```
public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
```

Model Class/Entity class:- This is where you define your data model classes with properties, constructors, getters, setters, and any necessary annotations.

Example:-
```
@Entity
public class User {
@Id @GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
private String username;
private String password;
// Constructors, getters, and setters
}
```

Repositories:-Repository interfaces are used to handle database operations. Spring Data JPA will automatically implement these based on the methods you define.

Example:
```
public interface UserRepository extends JpaRepository<User, Long> {
User findByUsername(String username); //user-defined method to search user by username
}
```

Annotation:-Annotations are used throughout Java classes to provide metadata and configure behaviors.

- `@SpringBootApplication` - Indicates a Spring Boot application entry point.

- **`@Entity`** - Defines a class as an entity (for ORM purposes).
- **`@Id`, `@GeneratedValue`** - Used in model classes to mark the primary key and define generation strategy.
- **`@Repository`** - Marks a class as a repository.
- **`@Autowired`** - Used for dependency injection, allowing Spring to manage class dependencies.
- **`@RequestMapping`** - Maps HTTP requests to handler methods in controllers.

- **`@Component`**
  - Marks a class as a Spring component. It is a generic stereotype for any Spring-managed component. Spring will autodetect these classes for dependency injection.

- **`@Service`**
  - A specialization of `@Component`, used to annotate classes that provide business logic or services.

- **`@Controller`**
  - Marks a class as a Spring MVC controller, allowing it to handle HTTP requests.

- **`@RestController`**
  - A combination of `@Controller` and **`@ResponseBody`**, used for RESTful web services. It automatically serializes return values to JSON or XML.

@ResponseBody:- When you use `@ResponseBody` on a method, Spring will:

1. Serialize the returned object to JSON (or XML, depending on the configuration).

Loops

While Loop:- A `while` loop repeatedly executes a block of code as long as a given condition is true. It checks the condition before entering the loop body, so if the condition is false initially, the loop will not execute.

```
int i = 1;
while (i <= 5) {
    System.out.println("Count: " + i);
    i++;
}
o/p:-
Count: 1
Count: 2
```

```
Count: 3
Count: 4
Count: 5
```

For Loop:- A `for` loop is typically used when the number of iterations is known beforehand. It consists of three parts in the header: initialization, condition, and update.

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

Output:

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

Do-While Loop:- A `do-while` loop is similar to a `while` loop, except that it checks the condition *after* executing the loop body. This guarantees that the loop body runs at least once, even if the condition is initially false.

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

Output:

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

Nested For Loop:- A nested `for` loop is a `for` loop inside another `for` loop. It is often used to work with multidimensional data structures, like matrices.

```
for (int i = 1; i <= 3; i++) {       // Outer loop
    for (int j = 1; j <= 3; j++) {   // Inner loop
        System.out.print(i * j + " ");
    }
    System.out.println(); // Moves to the next line after each inner
loop
}
```

Output:

```
1 2 3
2 4 6
3 6 9
```

Conditional Statements

Simpile IF Statement:- A simple `if` statement executes a block of code if a condition is true. If the condition is false, the code block is skipped.

```
int number = 10;
if (number > 5) {
    System.out.println("The number is greater than 5.");
}
```

Output (if `number` is 10):

```
The number is greater than 5.
```

IF-Else Statement:- An `if-else` statement allows you to execute one block of code if a condition is true and another block if it is false.

```
int number = 3;
if (number > 5) {
    System.out.println("The number is greater than 5.");
} else {
    System.out.println("The number is 5 or less.");
}
```

Output (if `number` is 3):

```
The number is 5 or less.
```

Ladder If-Else Statement:- Also known as an "else-if" ladder, it is used when multiple conditions need to be checked in sequence. If one of the conditions is true, the corresponding block executes, and the rest are skipped.

```
int score = 85;
if (score >= 90) {
    System.out.println("Grade: A");
} else if (score >= 80) {
    System.out.println("Grade: B");
} else if (score >= 70) {
    System.out.println("Grade: C");
} else if (score >= 60) {
    System.out.println("Grade: D");
} else {
    System.out.println("Grade: F");
}
```

Output (if `score` is 85):

```
Grade: B
```

Nested If-Else Statement:- In a nested `if-else` statement, an `if` or `else` block contains another `if-else` statement. It’s often used for more complex conditions.

```
int age = 25;
int height = 170;

if (age > 18) {
    if (height > 160) {
        System.out.println("Eligible for the competition.");
    } else {
        System.out.println("Not tall enough for the competition.");
    }
} else {
    System.out.println("Too young for the competition.");
}
```

Output (if `age` is 25 and `height` is 170):

```
Eligible for the competition.
```

Switch Statement:- A `switch` statement allows you to select one of many code blocks to execute, based on the value of a variable. It's often used as an alternative to the if-else ladder for checking multiple conditions based on a single variable.

```
int day = 3;
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    case 4:
        System.out.println("Thursday");
        break;
    case 5:
        System.out.println("Friday");
        break;
    case 6:
        System.out.println("Saturday");
        break;
    case 7:
        System.out.println("Sunday");
        break;
    default:
        System.out.println("Invalid day");
        break;
}
```

Output (if `day` is 3):

```
Wednesday
```

Object Oriented Programming

Class:- A **class** is a blueprint for creating objects. It defines the properties (attributes) and behaviors (methods) that the objects created from it will have. In Java, a class is defined using the `class` keyword.

Object :- An **object** is an instance of a class. It represents a specific example of the class with real values for the attributes. Each object has its own identity, state (values for the attributes), and behavior (methods it can perform).

Static Keyword

Static Variables:- A **static variable** (also known as a class variable) is a variable that belongs to the class, not to instances of the class. There is only one copy of a static variable per class, shared among all instances.

Static Methods:- A **static method** is a method that belongs to the class rather than to instances of the class. Static methods can be called without creating an instance of the class. However, they can only access static variables and other static methods directly, as they do not have access to instance variables.

Inheritance

types of Inheritance:- Java supports several types of inheritance, although it doesn't support **multiple inheritance** (inheriting from more than one superclass) directly due to ambiguity issues. The types of inheritance are:

- **Single Inheritance**: A subclass inherits from only one superclass.

```
class A { }
class B extends A { }  // Single inheritance
```

- **Multilevel Inheritance**: A class is derived from a superclass, which is itself derived from another superclass.

```
class A { }
class B extends A { }
class C extends B { }  // Multilevel inheritance
```

- **Hierarchical Inheritance**: Multiple classes inherit from the same superclass.

```
class A { }
class B extends A { }
class C extends A { }  // Hierarchical inheritance
```

Java does **not support multiple inheritance** (where a subclass inherits from more than one superclass) to avoid **ambiguity** or the "diamond problem." However, multiple inheritance can be achieved through **interfaces**.

Constructor Chaining:- Constructor chaining is a process of calling one constructor from another constructor within the same class or from a superclass. It helps in setting up the object's initial state through inheritance. In Java, constructor chaining can be done in two ways:

- **Within the Same Class**: Using the `this()` keyword to call another constructor in the same class.
- **From a Superclass**: Using the `super()` keyword to call a superclass constructor.

Polymorphism

Method Overloading:- Method overloading is a form of compile-time polymorphism, where multiple methods have the same name but different parameters within the same class. The compiler differentiates them by their parameter lists, which include different numbers, types, or orders of parameters.

Method overriding:- Method overriding is a form of runtime polymorphism, where a subclass provides a specific implementation for a method that is already defined in its superclass. The method in the subclass has the same name, return type, and parameters as the method in the superclass, but provides a new implementation.

Dynamic Polymorphism:- Dynamic polymorphism is achieved through **method overriding** and **late binding**. In dynamic polymorphism, the method to be executed is determined at runtime rather than compile-time. This is often achieved through polymorphic references—using a superclass reference to refer to a subclass object.

Encapsulation

Getter& Setter:- Getters and **setters** are methods used to access and update the private fields (variables) of a class. This practice encapsulates the data, allowing controlled access to class properties while keeping the actual variables private. This control helps maintain data integrity and security by adding logic in these methods, such as validation.

Final Keyword:- The `final` keyword in Java is used to create constants and restrict certain operations. It can be applied to variables, methods, and classes, each with specific effects.

Collection Framework

List Interface:- The `List` interface represents an **ordered collection** (also called a sequence) that allows **duplicate elements**. A `List` maintains the order in which elements are inserted, and you can access the elements via an index. Some commonly used implementations of the `List` interface are `ArrayList`, `LinkedList`, and `Vector`.

- **Type of List**:-ArrayList,LinkedList,Vector(Legacy class)
- **Key Features of List**
  - Allows duplicates.
  - Maintains insertion order.
  - Elements can be accessed by their index.
  - Supports positional access and insertion.

Set Interface:- The `Set` interface represents a **collection** that does not allow **duplicate elements**. It is an unordered collection, meaning there is no guarantee of the order in which elements are stored. `Set` is often used when the uniqueness of the elements is more important than their order.

- **Key Features of Set**:
  - Does not allow duplicates.
  - Does not maintain any order (in most cases).
  - Elements are stored uniquely.

Map Interface:- The `Map` interface represents a collection of **key-value pairs**. A `Map` does not allow duplicate keys, but it can have duplicate values. It allows you to map a unique key to a specific value. Common implementations of the `Map` interface include **`HashMap`**, **`LinkedHashMap`**, and **`TreeMap`**.

Exception handling

Purpose of Exception Handling:- Exception handling is a mechanism in Java (and other programming languages) that deals with runtime errors, enabling the program to continue its execution smoothly. The main purposes of exception handling are:

Try-Catch block:- The `try-catch` block is used to handle exceptions in Java. Code that might throw an exception is placed inside the `try` block, and the exception handling code is written inside the `catch` block.

Finally keyword:- The `finally` block is used to execute important code (like resource cleanup) whether or not an exception occurs. It is optional and is always executed after the `try` and `catch` blocks, even if an exception is thrown or not.

Types Of Exception:- Exceptions in Java are categorized into two main types:

- **Checked Exceptions**:
  - Checked exceptions are exceptions that are checked at compile time.
  - These exceptions are subclasses of `Exception` (except `RuntimeException` and its subclasses).

- The programmer is required to handle checked exceptions using a `try-catch` block or declare them using the `throws` keyword.
- Example: `IOException`, `SQLException`, `ClassNotFoundException`.

- **Unchecked Exceptions**:
  - Unchecked exceptions are exceptions that are not checked at compile time.
  - These exceptions are subclasses of `RuntimeException`.
  - These exceptions do not need to be explicitly handled or declared.
  - Example: `ArithmeticException`, `NullPointerException`, `ArrayIndexOutOfBoundsException`.

Throw Keyword:- The `throw` keyword is used to explicitly throw an exception from a method or block of code. It is used when a certain condition is met, and you want to intentionally throw an exception, either predefined or custom.

Throws Keyword:- The `throws` keyword is used in a method signature to indicate that the method may throw one or more exceptions. It is used for **checked exceptions** that are not handled within the method but are passed on to the caller.

Abstraction

Abstract class:- An **abstract class** is a class that cannot be instantiated directly. It is designed to be inherited by other classes, providing a base for them to build upon. An abstract class can contain both **abstract methods** (methods without a body) and **concrete methods** (methods with a body). It allows you to define common behavior while leaving specific details for subclasses.

Interface:- An **interface** is a reference type in Java, similar to a class, that can contain only **abstract methods** (methods without a body) and **constant fields** (i.e., static final variables). Interfaces are used to define a contract that other classes can implement. Unlike abstract classes, interfaces do not provide any method implementation (unless using default methods, introduced in Java 8).