JAVA

WEEK 1 DAY 1

What is java?

Java is a general-purpose programming language that is class-based, object-oriented, and designed to have as few implementation dependencies as possible. It is intended to let application developers write once, run anywhere (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. JAVA was developed by Sun Microsystems Inc and first released in 1991, later acquired by Oracle Corporation.

Java Components

Java Virtual Machine (JVM)

Generally referred as JVM, it’s the primary function is to execute the bytecode produced by compiler.

Java Development Kit (JDK)

This is a complete java development kit that includes JRE (Java Runtime Environment), compilers and various tools like JavaDoc, Java debugger etc.

Java Runtime Environment (JRE)

JRE allows you to run java programs, it includes JVM, browser plugins and applets support. When you only need to run a java program on your computer, you would only need JRE.

Main Java features

Platform Independence:

Java is designed to be platform-independent, meaning that Java programs can run on any device with a compatible Java Virtual Machine (JVM).

Object-Oriented Programming (OOP):

Java is a fully object-oriented programming language, emphasizing concepts such as encapsulation, inheritance, and polymorphism.

Simple and Easy to Learn:

Java was designed to be straightforward and easy to learn for programmers. It eliminates complex features such as pointers, operator overloading, and multiple inheritance, making it more accessible for developers.

Robust and Secure:

Java incorporates features like strong memory management, exception handling, and a comprehensive set of APIs, which contribute to the robustness and reliability of Java applications.

Multithreading:

Java supports multithreading, allowing concurrent execution of multiple threads within a program.

Install Java Development Kit (JDK):

Visit the Oracle JDK download page or OpenJDK to download and install the latest version of the Java Development Kit (JDK). Follow the installation instructions for your operating system.

Set the JAVA\_HOME environment variable:

On Windows: Right-click on "This PC" or "Computer," select "Properties," click on "Advanced system settings," go to the "Advanced" tab, and click "Environment Variables." Add a new system variable with the name JAVA\_HOME and the path to your JDK installation (e.g., C:\Program Files\Java\jdk1.8.0\_291).

Typical Structure of a Java program

A typical structure of a Java program contains the following elements:

Package declaration

Import statements

Comments

Class definition

Attributes

Methods/Behaviours

**DAY 2**

**What is a variable?**

A variable is a name given to a memory location. It is the basic unit of storage in a program.

The value stored in a variable can be changed during program execution.

A variable is only a name given to a memory location; all the operations done on the variable effects that memory location.

In Java, all the variables must be declared before use.

**How to declare variables?**

type: Type of data that can be stored in this variable.

name: Name given to the variable.

It can be assigned values in two ways:

* Variable Initialization
* Assigning value by taking input.

Declaring and initialize variables:

* datatype: Type of data that can be stored in this variable.
* variable\_name: Name given to the variable.
* value: It is the initial value stored in the variable.
* Declaring variables examples
* float simpleInterest; - Declaring float variable
* int myAge = 19; - Declaring and Initializing integer variable
* char firstLetter = 'h'; - Declaring and Initializing character variable

**There are three types of variables in Java:**

1. Local Variables: A variable defined within a block or method or constructor is called local variable.
2. Instance Variables: Instance variables are non-static variables and are declared in a class outside any method, constructor or block.
3. Static Variables: Static variables are also known as Class variables.

**Data types in Java:**

1) Primitive data types - A primitive data type specifies the size and type of variable values, and it has no additional methods.

2) Non-primitive data types – Arrays and Strings are non-primitive data types, these are the datatypes which have instances like objects. Hence, they are called reference variables. They are primarily classes, arrays, strings or interfaces.

There are eight primitive data types in Java:

Byte

Short

Int

Long

Float

Double

Boolean

Char

**DAY 3**

## OOP

Certainly! Here's a rewritten version of the information about OOP, its advantages, core features, other features, and example Java classes:

### What is OOP?

Object-Oriented Programming (OOP) is a programming paradigm centered around the concept of "objects." Java, a widely used language, embodies OOP principles.

### Key Concepts in Java OOP:

1. \*\*Class:\*\*

- Blueprint or template defining attributes (fields) and behaviors (methods) of objects.

2. \*\*Object:\*\*

- Instance of a class representing a specific entity in the program.

3. \*\*Method:\*\*

- Function defined within a class representing object behaviors.

4. \*\*Instance Variable:\*\*

- Variable defined in a class representing an object's state or properties.

5. \*\*Constructor:\*\*

- Special method initializing objects upon creation, sharing the class's name.

### Advantages of OOP:

1. \*\*Modularity and Reusability:\*\* Encapsulation organizes code into self-contained modules for easy reuse.

2. \*\*Code Reusability:\*\* Inheritance permits new classes to inherit properties and behaviors, reducing redundancy.

3. \*\*Abstraction:\*\* Focuses on essential object properties, hiding unnecessary details.

4. \*\*Encapsulation:\*\* Restricts access to certain components, minimizing complexity and improving maintainability.

5. \*\*Flexibility and Extensibility:\*\* Polymorphism allows treating different objects as the same type, promoting generic code.

6. \*\*Modeling Real-World Concepts:\*\* Aligns closely with real-world entities and relationships for effective modeling.

7. \*\*Collaborative Development:\*\* Encourages well-defined interfaces, facilitating teamwork on different system parts.

8. \*\*GUI Development:\*\* Suited for creating graphical user interfaces by representing visual elements as objects.

### Core Features of OOP:

1. \*\*Encapsulation:\*\* Bundles data and methods into a class.

2. \*\*Polymorphism:\*\* Allows treating different types as a common type.

3. \*\*Inheritance:\*\* Enables a class to inherit properties and behaviors from another class.

4. \*\*Abstraction:\*\* Simplifies complex systems by focusing on essential object aspects.

### Other Features Related to OOP:

1. \*\*Coupling:\*\* Measures dependence between classes or modules.

2. \*\*Cohesion:\*\* Measures relationships within modules.

3. \*\*Association:\*\* Represents bidirectional relationships between classes.

4. \*\*Aggregation:\*\* "Whole-part" relationship, allowing parts to exist independently.

5. \*\*Composition:\*\* Stronger form of aggregation where parts cannot exist independently.

### Activity 3: Example Java Classes:

```java

// PurchaseItem Class

public class PurchaseItem {

private String name;

private double unitPrice;

public PurchaseItem(String name, double unitPrice)

{

this.name = name;

this.unitPrice = unitPrice;

}

public double getPrice()

{

return unitPrice;

}

}

// WeighedItem Class (Subclass of PurchaseItem)

public class WeighedItem extends PurchaseItem

{

private double weight;

public WeighedItem(String name, double unitPrice, double weight) {

super(name, unitPrice);

this.weight = weight;

}

@Override

public double getPrice()

{

return super.getPrice() \* weight;

}

}

// CountedItem Class (Subclass of PurchaseItem)

public class CountedItem extends PurchaseItem

{

private int quantity;

public CountedItem(String name, double unitPrice, int quantity)

{

super(name, unitPrice);

this.quantity = quantity;

}

@Override

public double getPrice()

{

return super.getPrice() \* quantity;

}

// Usage example

public class Main {

public static void main(String[] args)

{

WeighedItem banana = new WeighedItem("Banana", 3.00, 1.37);

System.out.println(banana.getName() + " @ " + banana.getPrice());

CountedItem pens = new CountedItem("Pens", 4.5, 10);

System.out.println(pens.getName() + " @ " + pens.getPrice());

}

}

```These classes demonstrate inheritance and overridden methods to calculate prices for weighed and counted items.

Additional OOP Concepts:

1. \*\*Coupling:\*\* Degree of dependence between classes or modules.

2. \*\*Cohesion:\*\* Level of relatedness within modules.

3. \*\*Association:\*\* Bidirectional relationship between classes.

4. \*\*Aggregation:\*\* "Whole-part" relationship allowing independent existence.

5. \*\*Composition:\*\* Stronger "whole-part" relationship with dependent existence.

Object-Oriented Programming (OOP) gained popularity over declarative and procedural programming paradigms due to several advantages it offers:

1. \*\*Modularity and Reusability:\*\*

- OOP promotes encapsulation, bundling data and functionalities into objects. This modularity enables easier code organization, maintenance, and reuse. Objects can be reused in different parts of the code or in entirely different projects, enhancing efficiency and reducing redundancy.

2. \*\*Code Reusability through Inheritance:\*\*

- Inheritance in OOP allows creating new classes based on existing ones, inheriting their attributes and behaviors. This feature minimizes redundant code, promotes code reuse, and helps build hierarchies of related classes.

3. \*\*Abstraction for Simplification:\*\*

- OOP emphasizes abstraction, focusing on essential properties and hiding complex implementation details. This approach simplifies the understanding of code and enhances manageability by representing real-world entities as objects with specific behaviors and attributes.

4. \*\*Encapsulation for Better Control:\*\*

- Encapsulation restricts access to certain components of an object, allowing developers to control and protect data integrity. By hiding implementation details, encapsulation reduces complexity, improves maintainability, and minimizes unintended interference with the internal workings of a class.

5. \*\*Flexibility and Extensibility through Polymorphism:\*\*

- Polymorphism allows objects of different types to be treated as objects of a common type, enabling the creation of generic and flexible code. This feature facilitates the handling of different types uniformly and enhances the adaptability of code to changing requirements.

6. \*\*Modeling Real-World Concepts:\*\*

- OOP aligns closely with how we conceptualize real-world entities and relationships, making it easier to map real-world problems into code. The concepts of objects, classes, inheritance, and relationships between objects resemble the relationships between real-world entities, making the code more intuitive and easier to design.

7. \*\*Collaborative Development:\*\*

- OOP encourages the creation of well-defined interfaces between different parts of a program. This facilitates teamwork, as different developers can work on distinct modules or classes independently, as long as they adhere to the specified interfaces.

8. \*\*Suitability for GUI Development:\*\*

- OOP aligns well with graphical user interface (GUI) development principles. Objects representing visual elements and their interactions in the GUI can be modeled effectively using OOP concepts.

While declarative and procedural programming paradigms have their strengths, the advantages offered by OOP, such as code organization, reusability, abstraction, flexibility, and modeling capabilities, have contributed significantly to its popularity and widespread adoption in software development.

OOP Core Features

Encapsulation is when object only exposes selected information

Inheritance one class acquires the methods and properties of another.

Polymorphism Entities can have more than one form.

Abstraction hides complex details to reduce complexity.

The core features of Object-Oriented Programming (OOP)

1. \*\*Encapsulation:\*\*

- Encapsulation is the bundling of data (attributes) and methods (functions) that operate on the data into a single unit known as a class. It allows the internal workings of an object to be hidden from the outside, providing controlled access to the object's state and behaviors. Encapsulation helps in data protection and promotes the principle of information hiding.

2. \*\*Polymorphism:\*\*

- Polymorphism refers to the ability of objects to take on multiple forms. In Java, polymorphism allows objects of different classes to be treated as objects of a common superclass through inheritance. There are two main types of polymorphism:

- \*\*Compile-time Polymorphism (Method Overloading):\*\* It involves having multiple methods in the same class with the same name but different parameters.

- \*\*Runtime Polymorphism (Method Overriding):\*\* It allows a subclass to provide a specific implementation of a method that is already defined in its superclass.

3. \*\*Inheritance:\*\*

- Inheritance is a mechanism that allows a class (subclass or derived class) to inherit properties (attributes and methods) and behaviors from another class (superclass or base class). The subclass can extend or override the functionalities of its superclass. It promotes code reuse, hierarchy establishment, and the creation of a more specialized class from a more generalized one.

4. \*\*Abstraction:\*\*

- Abstraction involves simplifying complex systems by modeling classes based on the essential properties and behaviors relevant to the problem. It allows programmers to focus on what an object does rather than how it achieves its functionality. Abstract classes and interfaces are used to achieve abstraction in Java. Abstraction enables the creation of generalized blueprints that can be extended for more specific implementations.

These core features provide the foundation for designing and implementing object-oriented systems.

They enable developers to create modular, reusable, and maintainable code by promoting concepts such as code organization, data protection, code reuse, and flexibility in handling various object interactions and hierarchies.

Features of OOP – Other features

Besides the core features of Object-Oriented Programming (OOP) such as encapsulation, polymorphism, inheritance, and abstraction, there are several other important concepts and principles related to OOP:

1. \*\*Coupling:\*\*

- \*\*Definition:\*\* Coupling measures the degree of dependence between classes or modules. It indicates how much one class knows about or relies on another. Low coupling is desirable as it leads to more maintainable and flexible systems.

- \*\*Types:\*\*

- \*\*Low Coupling:\*\* Classes have minimal knowledge of each other. Changes in one class have minimal impact on others.

- \*\*High Coupling:\*\* Classes are closely connected, and changes in one class may require changes in others.

2. \*\*Cohesion:\*\*

- \*\*Definition:\*\* Cohesion measures the relatedness and interdependency of elements within a module (like a class or method). High cohesion is desirable as it leads to more maintainable and understandable code.

- \*\*Types:\*\*

- \*\*High Cohesion:\*\* Elements within a module are closely related and work together towards a common goal.

- \*\*Low Cohesion:\*\* Elements within a module have weak relationships and may not contribute to a common goal.

3. \*\*Association:\*\*

- \*\*Definition:\*\* Association represents a bidirectional relationship between two classes. It can be a one-to-one, one-to-many, or many-to-many relationship.

- \*\*Example:\*\* In a university system, a `Student` class may be associated with a `Course` class through enrollment.

4. \*\*Aggregation:\*\*

- \*\*Definition:\*\* Aggregation represents a "whole-part" relationship between classes. It implies that one class (the whole) contains another class (the part), but the part can exist independently.

- \*\*Example:\*\* In a car example, a `Car` class may have an aggregation relationship with a `Wheel` class. A `Car` has wheels, and wheels can exist independently.

5. \*\*Composition:\*\*

- \*\*Definition:\*\* Composition is a stronger form of aggregation where the part cannot exist independently of the whole. If the whole is destroyed, all its parts are destroyed as well.

- \*\*Example:\*\* In a computer example, a `Computer` class may have a composition relationship with a `CPU` class. If the `Computer` is destroyed, the `CPU` is also destroyed.

Understanding these additional concepts in OOP is crucial for designing systems with maintainability, flexibility, and clarity in mind. Properly managing coupling and cohesion helps create systems that are easier to modify and extend, while association, aggregation, and composition allow for more accurate modeling of real-world relationships in code.

**Day 4**

**Introduction to Decision control**

* Decision control in Java refers to the programming constructs that allow you to control the flow of execution based on certain conditions or decisions.

There are mainly three decision-making structures in Java:

1. if statement:

   - The `if` statement evaluates a boolean expression and executes a block of code only if the condition is true.

   Syntax:

   ```java

   if (condition) {

       // Code to be executed if the condition is true

   }

   ```

2. \*\*if-else statement:\*\*

   - The `if-else` statement allows the program to execute one block of code if the condition is true and another block if the condition is false.

   Syntax:

   ```java

   if (condition) {

       // Code to be executed if the condition is true

   } else {

       // Code to be executed if the condition is false

   }

   ```

3. \*\*else-if ladder:\*\*

   - The `else-if` ladder allows checking multiple conditions one by one. If the condition is true, the corresponding block of code will be executed. If none of the conditions are true, the default else block (if present) will be executed.

   Syntax:

   ```java

   if (condition1) {

       // Code to be executed if condition1 is true

   } else if (condition2) {

       // Code to be executed if condition2 is true

   } else if (condition3) {

       // Code to be executed if condition3 is true

   } else {

       // Default code if no condition is true

   }

   ```

Example:

```java

int number = 10;

// Example of if statement

if (number > 0) {

    System.out.println("Number is positive");

}

// Example of if-else statement

if (number > 0) {

    System.out.println("Number is positive");

} else {

    System.out.println("Number is not positive");

}

// Example of else-if ladder

if (number > 0) {

    System.out.println("Number is positive");

} else if (number < 0) {

    System.out.println("Number is negative");

} else {

    System.out.println("Number is zero");

}

```

* These decision control structures help in writing programs that can make choices based on given conditions, allowing for dynamic behavior and more control over the flow of execution in Java programs.

**If Statement, IF ELSE statement, Nested If Statement**

`if` statement, `if-else` statement, and nested `if` statements in Java:

### 1. If Statement:

The `if` statement executes a block of code if the condition specified is true.

Syntax:

```java

if (condition) {

    // Code to execute if the condition is true

}

```

Example:

```java

int number = 10;

// Example of if statement

if (number > 0) {

    System.out.println("Number is positive");

}

```

### 2. If-Else Statement:

The `if-else` statement executes one block of code if the condition is true and another block if the condition is false.

Syntax:

```java

if (condition) {

    // Code to execute if the condition is true

} else {

    // Code to execute if the condition is false

}

```

Example:

```java

int number = 10;

// Example of if-else statement

if (number > 0) {

    System.out.println("Number is positive");

} else {

    System.out.println("Number is not positive");

}

```

### 3. Nested If Statement:

Nested `if` statements contain an `if` statement within another `if` statement. This allows for more complex conditional logic.

Syntax:

```java

if (condition1) {

    // Code to execute if condition1 is true

    if (condition2) {

        // Code to execute if condition2 is true

    }

}

```

Example:

```java

int x = 10;

int y = 5;

// Example of nested if statement

if (x > 0) {

    System.out.println("x is positive");

    if (y > 0) {

        System.out.println("y is also positive");

    }

}

```

* Nested `if` statements can have multiple levels, each level checking a specific condition.
* However, excessive nesting may lead to complex code and reduced readability, so it's essential to use them judiciously.

**Switch statement and the Tenary (?) operator**

 the `switch` statement and the ternary conditional operator (`? :`) in Java:

### 1. Switch Statement:

The `switch` statement is used to select one of many code blocks to be executed based on the value of a variable or expression.

\*\*Syntax:\*\*

```java

switch (expression) {

    case value1:

        // Code to execute if expression equals value1

        break;

    case value2:

        // Code to execute if expression equals value2

        break;

    // More cases...

    default:

        // Code to execute if expression doesn't match any case

}

```

\*\*Example:\*\*

```java

int day = 3;

String dayName;

// Example of switch statement to get the day name

switch (day) {

    case 1:

        dayName = "Monday";

        break;

    case 2:

        dayName = "Tuesday";

        break;

    case 3:

        dayName = "Wednesday";

        break;

    // More cases...

    default:

        dayName = "Invalid day";

}

System.out.println("The day is: " + dayName);

```

### 2. Ternary Operator (? :):

The ternary operator `? :` is a concise way to express conditional statements. It evaluates a boolean expression and returns one of two values depending on the result of the evaluation.

\*\*Syntax:\*\*

```java

result = (condition) ? valueIfTrue : valueIfFalse;

```

\*\*Example:\*\*

```java

int x = 10;

int y = 5;

// Example of using the ternary operator

String message = (x > y) ? "x is greater than y" : "y is greater than or equal to x";

System.out.println(message);

```

The ternary operator evaluates the condition `(x > y)`. If the condition is true, it returns `"x is greater than y"`, otherwise, it returns `"y is greater than or equal to x"`.

* The `switch` statement allows for multiple conditional cases based on the value of an expression, while the ternary operator provides a concise way to create conditional expressions that return different values based on a condition.
* Both are useful tools for controlling the flow of execution in Java.

**Day 5 Java Modifiers**

Java provides a number of access modifiers to set access levels for classes, variables, methods, and constructors.

The four access levels are:

Visible to the package, the default. No modifiers are needed. Visible to the class only (private). Visible to the world (public). Visible to the package and all subclasses (protected).

Default Access Modifier - No Keyword

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc. A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

Example

Variables and methods can be declared without any modifiers, as in the following an example:

String version = "1.5.1"; boolean processOrder() { return true; }

Private Access Modifier - Private

Methods, variables, and constructors that are declared private can only be accessed within the declared class itself.Private access modifier is the most restrictive access level. Class and interfaces cannot be private. Variables that are declared private can be accessed outside the class, if public getter methods are present in the class. Using the private modifier is the main way that an object encapsulates itself and hides data from the outside world.

Example

The following class uses private access control

public class Logger { private String format;

public String getFormat() { return this.format; }

public void setFormat(String format) { this.format = format; } } The format variable of the Logger class is private, so there's no way for other classes to retrieve or set its value directly. So, to make this variable available to the outside world, we defined two public methods: getFormat(), which returns the value of format, and setFormat(String), which sets its value.

Public Access Modifier - Public

A class, method, constructor, interface, etc. declared public can be accessed from any other class. Therefore, fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

However, if the public class we are trying to access is in a different package, then the public class still needs to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

Example

The following function uses public access control:

Public static void main(String[] arguments) { } The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.

Protected Access Modifier - Protected

Variables, methods, and constructors, which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class. The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected. Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

Example

The following parent class uses protected access control, to allow its child class override openSpeaker() method: class AudioPlayer { protected boolean openSpeaker(Speaker sp) { // implementation details } }

class StreamingAudioPlayer extends AudioPlayer { boolean openSpeaker(Speaker sp) { // implementation details } } If we define openSpeaker() method as private, then it would not be accessible from any other class other than AudioPlayer. If we define it as public, then it would become accessible to all the outside world. But our intention is to expose this method to its subclass only, that's why we have used protected modifier.

Access Control and Inheritance

The following rules for inherited methods are enforced:

1. Methods declared public in a superclass also must be public in all subclasses.
2. Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
3. Methods declared private are not inherited at all, so there is no rule for them.

Example In this example, we've created a class with a private variable age and a variable with default scope as name. Using setter/getter method, we're updating age and getting value and name is updated directly.

public class Puppy { private int age; String name;

public Puppy() {   
}

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

public int getAge( ) { return age; }

public static void main(String []args) { Puppy myPuppy = new Puppy();

// update age variable using method call   
myPuppy.setAge( 2 );   
   
// update name directly   
myPuppy.name = "Tommy";   
System.out.println("Age: " + myPuppy.getAge() +", name: " + myPuppy.name ); 

} } Output