Object-Oriented Programming (OOPS) Overview:

1. OOPS Definition:

Object-Oriented Programming is a paradigm that models the real-world entities through the use of objects.

2. Object Definition:

Objects represent real-world entities like Car, Bike, or Person.

3. OOPS vs Procedural Programming:

Procedural divides the program into functions; OOPS divides it into objects.

Procedural lacks data hiding; OOPS emphasizes data hiding through objects.

Procedural Programming Object-Oriented Programming

Functions-oriented Object-oriented

Limited data hiding Emphasizes data hiding

No overloading Supports overloading

No inheritance Supports inheritance

Examples:

Procedural: C, Fortran, Pascal

OOPS: C++, Java, C#, Python

Objects & Classes:

1. Object States:

Objects have properties (states) and behaviors (functions).

2. Class and Object Relationship:

Objects are instances of classes, acting as blueprints/templates.

Multiple objects can be created from one class.

3. Class Creation:

Classes are created using the "class" keyword.

4. Example:

Example: Dog is an object with properties like Age, Bark, Breed, and Color.

Pillars of OOPS:

1. Data Abstraction:

Hides internal implementation, exposing essential functionality.

Achieved through interfaces and abstract classes.

Advantage: Increased security and confidentiality.

2. Data Encapsulation:

Bundles data and code into a single unit (class).

Involves private variables and necessary setters/getters.

Advantages: Loosely coupled code, better access control, and security.

3. Inheritance:

Class capability to inherit properties and behaviors from a parent class.

Achieved using "extends" and "implements" keywords.

Types: Single, Multilevel, Hierarchical, Multiple (solves diamond problem).

Advantages: Code reusability, supports polymorphism.

4. Polymorphism:

A method's ability to behave differently in different situations.

Types: Compile Time (Method Overloading), Runtime (Method Overriding).

Example: A person can be a father, husband, employee, etc.

Objects Relationships:

1. Is-a Relationship:

Achieved through interfaces.

Example: Dog is-an animal.

2. Has-a Relationship:

Objects used in another class.

One-to-one, one-to-many, many-to-many relationships.

Examples: School has students, Bike has an engine, School has classes.

Associations:

Relationship between two different objects.

Aggregation:

Objects of different classes survive individually.

Composition:

Ending one object ends another (object is composed of another).

Example: A house is composed of bricks.

Diamond Problem –

The "diamond problem" is a term used in the context of object-oriented programming, particularly in languages that support multiple inheritance. Java, although supporting interfaces with multiple inheritance, doesn't suffer from the diamond problem in the same way as some other languages like C++ do.

The diamond problem occurs when a class inherits from two classes that have a common ancestor. If both parent classes provide an implementation for the same method (a method with the same signature), and the child class does not override that method, there is ambiguity about which implementation should be inherited.

Here's a simple illustration of the diamond problem:

class A {

void doSomething() {

System.out.println("A's implementation");

}

}

class B extends A {

// Inherits doSomething from class A

}

class C extends A {

// Inherits doSomething from class A

}

class D extends B, C {

// This creates ambiguity, as class D inherits doSomething from both B and C

}

// In languages like C++, calling D's doSomething would lead to ambiguity issues.

In Java, this problem is avoided because Java doesn't support multiple inheritance for classes. However, Java does support multiple inheritance through interfaces, and it has its mechanisms to handle this kind of situation. If a class implements two interfaces that declare the same method, the implementing class must provide its implementation for that method, resolving any ambiguity.

Here's an example with interfaces in Java:

interface A {

void doSomething();

}

interface B extends A {

// Inherits doSomething from interface A

}

interface C extends A {

// Inherits doSomething from interface A

}

class D implements B, C {

// The implementing class D must provide its own implementation of doSomething

@Override

public void doSomething() {

System.out.println("D's implementation");

}

}

In this way, Java's approach to multiple inheritance through interfaces avoids the diamond problem by requiring explicit implementation in the class that implements multiple interfaces with conflicting method signatures.

Java -

Java is a high-level, object-oriented programming language known for its platform independence, portability, and versatility. Developed by Sun Microsystems (now owned by Oracle Corporation), Java is designed to be a general-purpose language suitable for a wide range of application domains.

Key Features:

Platform Independence:

Java applications are compiled into an intermediate bytecode, which can be executed on any device with a Java Virtual Machine (JVM). This "write once, run anywhere" approach enhances portability.

Object-Oriented Nature:

Java follows the principles of object-oriented programming (OOP), emphasizing the use of classes and objects for modular and reusable code design. Concepts such as inheritance, polymorphism, encapsulation, and abstraction are integral to Java's OOP paradigm.

Memory Management:

Java features automatic memory management through garbage collection. This simplifies memory allocation and deallocation processes, reducing the likelihood of memory-related errors.

Rich Standard Library:

Java includes a comprehensive standard library that provides pre-built functionalities for tasks such as input/output operations, networking, and data structures. This extensive library accelerates development by minimizing the need for low-level coding.

Multi-Threading Support:

Java supports multithreading, enabling the concurrent execution of multiple threads within a program. This is essential for optimizing performance in applications that require parallel processing.

Exception Handling:

Robust exception handling mechanisms allow developers to detect and manage errors effectively, contributing to the creation of resilient and reliable software.

Security Features:

Java incorporates security features such as the sandboxing model, which restricts the actions of Java applications for enhanced security. This is particularly crucial for web-based applications.

Frameworks and Ecosystem:

Java has a rich ecosystem of frameworks and libraries that facilitate the development of various types of applications. Frameworks like Spring and Hibernate are widely used for enterprise-level development and data access.

Community and Longevity:

Java boasts a large and active developer community, contributing to a wealth of resources, forums, and third-party tools. Its enduring popularity in the industry ensures the availability of skilled developers and ongoing support.

Enterprise Adoption:

Java is widely adopted in enterprise environments for building scalable and robust systems. Its stability, performance, and support for modular architecture make it a preferred choice for large-scale applications.

In summary, Java is a versatile and reliable programming language known for its cross-platform compatibility, object-oriented principles, and extensive ecosystem, making it suitable for a broad spectrum of software development projects.

JDK, JRE, JVM -

JDK :

- Java Development Kit.

- Flow -

- A kit that allows user to :

- Develop a Java Program.

- Run it finally.

- Top Level Component.

JDK = JRE + Development Tools

JRE :

- Java Runtime Enviroment.

- The required component to run a java program.

- So, the client who is supposed to run the java program, he has to install JRE.

- Middle Level Component.

JRE = JVM + Library Classes

JVM :

- Java Virtual Machine.

- It reads the program line by line and executes it.

- It is an interpreter to run the program.

- Low Level Component.

- It helps to run the java byte code into current os executable machine language.

JVM = JIT + Interpreter

JIT :

- Just in time compiler.

- Responsible to help interpreter to run java byte code fast.

- Its mainely to increase performance.

How to talk -

1. Places, which exists where.

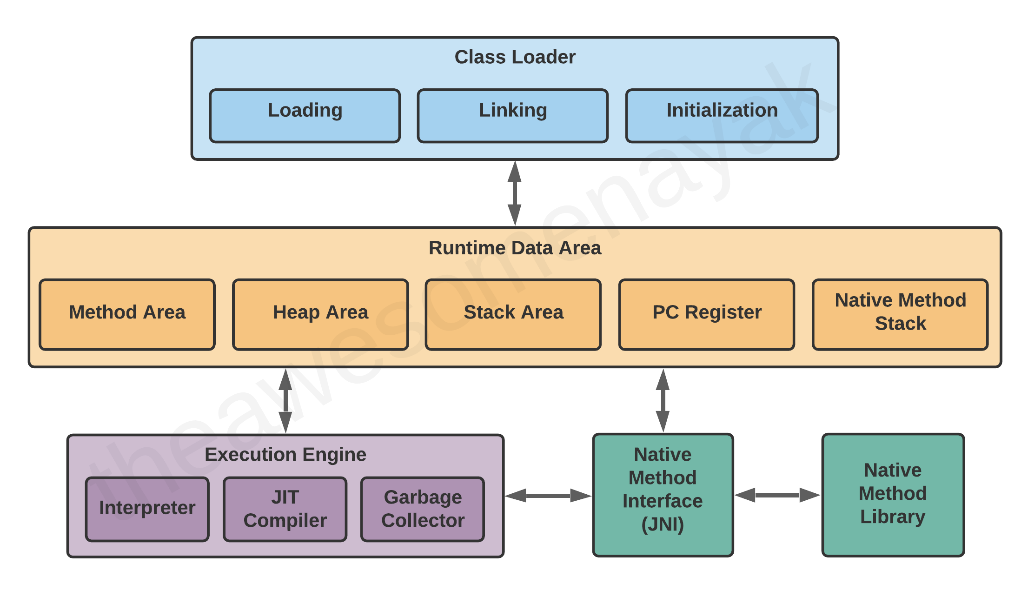
2. JDK, its abbreviation, what is does and its responsibilites.

3. JRE, same.

4. JVM, same.

5. Finally JIT.

JVM Architecture –



Class Loader –

* Bootstrap C.L :
  + Responsible to load classes from bootstrap class path -> rt.jar.
* Extension C.L :
  + Classes present inside ext folder are loaded by extension C.L.
* Application C.L :
  + Application level class path classes are loaded by Application C.L.

Note : Bootstrap C.L takes higher priority to load a class.

Runtime Data Area –

Method Area – Class Level Data

Heap Area – Object Data

Stack Area –

* For every thread, a separate runtime area will be created.
* Each entry in stack is called – Stack Frame.
* Stack Frame : Local Variable Array, Operand Stack, Frame Data.

PC Register –

* For every thread, pc registers are created.
* Its responsible to hold address of every executing instruction.

Native Method Stack –

* Holds native method information.

Execution Engine :

* Interpreter :
* JIT Compiler :
  + Intermediate Code Generator
  + Code Optimiser
  + Target Code Generator -> Generates Machine Code / Native Code
  + Profiler Component
* Garbage Collector :
* Security Manager :

Native Method Libraries :

* Java native interface is responsible to provide native method libraries.

Flow :

1. Once the class is loaded by the class loaders.
2. Immediately byte code verifier starts to verify the generated byte code.
3. Preparation starts here –
   1. For static variables, memory will be assigned with default values.
   2. And default values will be placed inside initializer.
4. Resolve starts finally –
   1. All symbolic references are resolved using resolvers and the references are kept inside initializers.

Why Java has only one public class in a single file ?

In Java, a single file can have only one public class, and the name of the file must match the name of that public class. This rule is imposed by the Java compiler for several reasons, including simplicity, predictability, and maintainability. Here are some key reasons behind this design choice:

File Naming Conventions:

Java enforces a convention where the filename should match the name of the public class declared in the file. This helps developers locate the source code for a particular class easily.

Accessibility:

The public modifier means that the class is accessible from outside the package. Limiting the number of public classes per file ensures that there is a clear association between the file name and the public class's name for users of the class.

EntryPoint in Java Programs:

In Java, the public static void main(String[] args) method is the entry point for the execution of a Java program. This method must be in a public class, and it helps to have a clear association between the filename and the entry point class.

Compilation Units:

In Java, a source file is considered a compilation unit. Having a one-to-one relationship between the public class and the file simplifies the compilation process. The Java compiler can compile each file independently without ambiguity.

Class Loaders:

Java class loaders play a crucial role in loading classes at runtime. The one-to-one relationship between the file name and the public class simplifies class loading, as the class loader can predict the location of the class file based on the class name.

Code Organization:

Encouraging a one-public-class-per-file approach helps maintain a clean and organized codebase. It makes it easier for developers to locate, understand, and manage classes within a project

Methods in Java –

In Java, methods are blocks of code that perform a specific task and are defined within a class. There are several types of methods in Java, including:

Instance Methods:

* These are non-static methods that operate on an instance of a class.
* They can access instance variables and other instance methods directly.

public class MyClass {

public void instanceMethod() {

// Code for instance method

}

}

Static Methods:

* These methods belong to the class rather than to any instance of the class.
* They can be called using the class name and don't require an instance to be created.

public class MyClass {

public static void staticMethod() {

// Code for static method

}

}

Getter and Setter Methods:

* These methods are commonly used for accessing and modifying the values of private instance variables.

public class MyClass {

private int myVariable;

public int getMyVariable() {

return myVariable;

}

public void setMyVariable(int value) {

this.myVariable = value;

}

}

Constructor:

* Constructors are special methods used for initializing objects when they are created.
* They have the same name as the class and do not have a return type.

public class MyClass {

public MyClass() {

// Code for constructor

}

}

Method Overloading:

* Java supports method overloading, which allows multiple methods with the same name but different parameters.

public class MyClass {

public void myMethod(int x) {

// Code for method with one parameter

}

public void myMethod(int x, int y) {

// Code for method with two parameters

}

}

Varargs (Variable-Length Argument Lists):

* Java allows you to pass a variable number of arguments to a method using varargs.

public class MyClass {

public void myMethod(int... numbers) {

// Code for method with variable arguments

}

}

Abstract Methods:

* These methods are declared without an implementation in an abstract class, and subclasses must provide the implementation.

public abstract class MyAbstractClass {

public abstract void abstractMethod();

}

Final Methods:

* When a method is declared as final, it cannot be overridden by subclasses.

public class MyClass {

public final void finalMethod() {

// Code for final method

}

}

Constructors in Java –

In Java, a constructor is a special type of method that is used for initializing objects. It has the same name as the class and does not have a return type, not even void. Constructors are called when an object is created using the new keyword. They are used to set initial values for the object's attributes or perform any other necessary setup.

Rules of Constructors:

Constructor Name:

The constructor name must be the same as the class name.

Return Type:

Constructors do not have a return type, not even void.

Accessibility:

Constructors can have different access modifiers (e.g., public, private, protected), controlling their visibility.

Overloading:

Like other methods, constructors can be overloaded by having different parameter lists.

Types of Constructors:

Default Constructor:

If a class does not explicitly define any constructor, Java provides a default constructor.

It initializes member variables to their default values (e.g., 0 for numeric types, null for objects).

public class MyClass {

// Default constructor provided by Java if not explicitly defined

}

Parameterized Constructor:

Constructors that take parameters to initialize the object with specific values.

public class MyClass {

private int value;

// Parameterized constructor

public MyClass(int value) {

this.value = value;

}

}

Copy Constructor:

A constructor that takes an object of the same class as a parameter, creating a copy of that object.

public class MyClass {

private int value;

// Copy constructor

public MyClass(MyClass original) {

this.value = original.value;

}

}

Constructor Overloading:

Having multiple constructors in a class with different parameter lists.

public class MyClass {

private int value;

// Parameterized constructor

public MyClass(int value) {

this.value = value;

}

// Default constructor

public MyClass() {

// Default initialization

}

}

Chaining Constructors (Using this):

Constructors can call another constructor in the same class using this().

public class MyClass {

private int value;

// Parameterized constructor

public MyClass(int value) {

this.value = value;

}

// Default constructor calling the parameterized constructor

public MyClass() {

this(0); // Calls the parameterized constructor with a default value

}

}

Constructors play a crucial role in object-oriented programming by ensuring that objects are properly initialized when they are created. They help establish the initial state of an object, making it ready for use.

Questions on Constructors –

Why constructors don't have any return type?

Constructors don't have a return type because their main purpose is to initialize objects, and the creation of an object is automatically signaled by the new keyword. Having a return type for constructors would go against the concept of object creation and initialization. The responsibility of a constructor is to set up the initial state of the object, not to return a value.

Why constructors can't be final?

Constructors can't be final because the final keyword is used to indicate that a method or class cannot be overridden or extended. Since constructors are not inherited like regular methods, marking them as final doesn't make sense. Additionally, constructors are automatically called when an object is created, and the concept of inheritance is not applicable to constructors.

Why constructors can't be abstract?

Constructors can't be abstract because abstract methods are meant to be implemented by subclasses, but constructors are automatically called during the instantiation of an object. If a constructor were abstract, it would be incomplete and could not be invoked to create an object. Abstract classes can have abstract methods, but constructors are not part of that abstraction.

Why constructors can't be static?

Constructors can't be static because the static keyword is used to define methods or fields that belong to the class rather than instances of the class. Constructors, on the other hand, are responsible for initializing instance variables and creating object-specific state. They are called on an instance and are not associated with the class itself.

Can we define constructors in an interface?

No, constructors cannot be defined in interfaces. Interfaces in Java are meant to declare abstract methods that implementing classes must provide. Since constructors are automatically called during the instantiation of an object and interfaces cannot be instantiated, it doesn't make sense to have constructors in interfaces.

Why is the constructor name the same as the class name?

The constructor name is the same as the class name to provide a clear and unambiguous way to identify the constructor associated with a class. When you create an object using the new keyword, Java looks for a constructor with the same name as the class. Using the class name for the constructor ensures that it is easily recognizable and invoked when an object is being created, making the code more readable and intuitive.

Java Memory Management & Garbage Collector –

In Java, the garbage collector is a part of the Java Virtual Machine (JVM) responsible for automatic memory management. Its primary purpose is to identify and reclaim memory that is no longer in use by the program. Java uses automatic memory management to handle memory allocation and deallocation, relieving the programmer from explicitly managing memory resources.

Here's how the garbage collector works:

Object Allocation:

When objects are created in a Java program using the new keyword, memory is allocated for those objects in the heap memory area.

Reference Tracking:

The JVM keeps track of references to objects. If an object is referenced by any variable, it is considered reachable.

Mark and Sweep:

Periodically, or when the JVM determines it is necessary, the garbage collector performs a "mark and sweep" operation.

The "mark" phase identifies all reachable objects by starting from the roots (e.g., global variables, method parameters) and marking all objects that can be reached through reference chains.

Sweep:

In the "sweep" phase, the garbage collector identifies and reclaims memory occupied by objects that are not marked as reachable.

This involves freeing up the memory occupied by objects that are no longer accessible or referenced.

Compact:

Some garbage collectors may include a "compaction" phase where memory is compacted, reducing fragmentation and improving the efficiency of memory allocation.

Finalization:

Before an object is garbage collected, the finalize() method (if overridden) is called on the object. This method can be used for cleanup operations before the object is reclaimed.

The use of a garbage collector in Java simplifies memory management for developers, as they do not need to explicitly free up memory or worry about memory leaks. However, it's essential to be mindful of object lifecycles and avoid unintentional strong references that could lead to objects being retained longer than necessary.

Different implementations of the garbage collector exist in Java, and the JVM may use various garbage collection algorithms based on the selected garbage collector and its configuration. Some common garbage collectors in Java include the Serial, Parallel, CMS (Concurrent Mark-Sweep), G1 (Garbage-First), and ZGC (Z Garbage Collector). The choice of garbage collector depends on factors such as application requirements, system resources, and performance considerations.

How to make Objects Eligible for GC –

In Java, objects become eligible for garbage collection when they are no longer reachable or accessible by the program. The garbage collector identifies and reclaims memory occupied by objects that cannot be accessed by the program. Here are some ways to make an object eligible for garbage collection:

Nullifying References:

Set all references to the object to null. Once there are no more references pointing to an object, it becomes unreachable and can be collected by the garbage collector.

MyClass obj = new MyClass();

// ...

obj = null; // Object is now eligible for garbage collection

Method Scope:

If an object is created within a method, and there are no references to it outside the method, the object becomes eligible for garbage collection when the method completes.

void someMethod() {

MyClass obj = new MyClass();

// Object is eligible for garbage collection when someMethod() completes

}

Assignment to Another Object:

If an object is reassigned to another reference, the original reference no longer points to the object, making it eligible for garbage collection.

MyClass obj1 = new MyClass();

MyClass obj2 = new MyClass();

// ...

obj1 = obj2; // obj1 no longer references the original object and is eligible for garbage collection

Object Goes Out of Scope:

If an object is declared within a block of code and the block finishes execution, the object becomes unreachable and eligible for garbage collection.

{

MyClass obj = new MyClass();

// Object is eligible for garbage collection when the block completes

}

Circular References:

Circular references between objects can prevent them from being garbage collected. Break the circular reference or set one of the references to null to make the objects eligible for collection.

class MyClass {

MyClass anotherObj;

public void setAnotherObj(MyClass obj) {

this.anotherObj = obj;

}

}

MyClass obj1 = new MyClass();

MyClass obj2 = new MyClass();

obj1.setAnotherObj(obj2);

obj2.setAnotherObj(obj1);

// Break the circular reference or set one reference to null to make objects eligible for garbage collection

Explicitly Invoking System.gc():

While it's generally not recommended, you can explicitly suggest garbage collection by calling System.gc(). Note that the JVM may or may not honor this request.

System.gc(); // Suggest garbage collection