CS305-Object Oriented Programming &Methodology Sem- 3 CSE RGPV

By:- Mr. Sonu Kumar

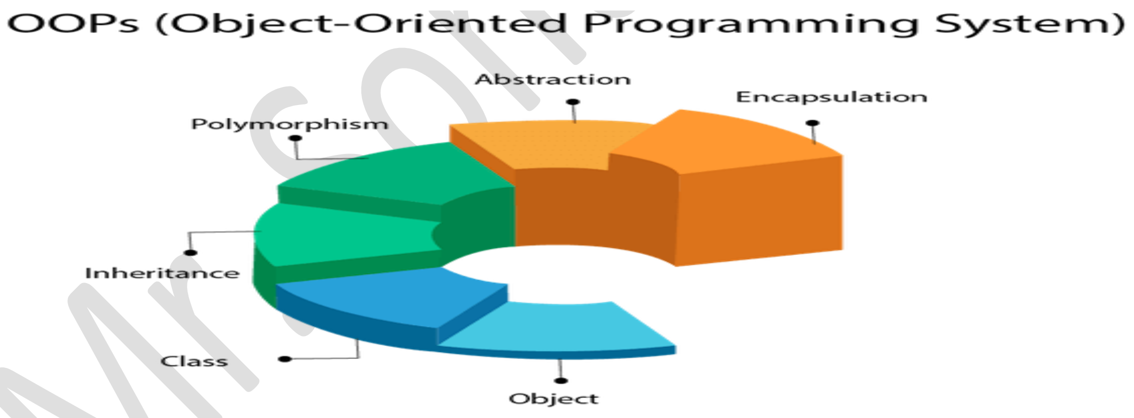
Module – 1: Introduction to OOPs

Java OOPs Concepts

- Object-Oriented Programming is a paradigm that provides many concepts, such as **inheritance**, **data binding**, **polymorphism**, etc.
- **Simula** is considered the first object-oriented programming language.
- The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.
- **Smalltalk** is considered the first truly object-oriented programming language.
- The popular object-oriented languages are Java, C#, PHP, Python, C++, etc.

OOPs (Object-Oriented Programming System)

- **Object** means a real-world entity such as a pen, chair, table, computer, watch, etc.
- **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects.
- It simplifies software development and maintenance by providing some concepts:



Object:-

- Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.
- An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory.

- Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

Class

- *Collection of objects* is called class. It is a logical entity.
- A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

Inheritance

- *When one object acquires all the properties and behaviors of a parent object*, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

- If *one task is performed in different ways*, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.
- In Java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

Abstraction

- *Hiding internal details and showing functionality* is known as abstraction. For example phone call, we don't know the internal processing.
- In Java, we use abstract class and interface to achieve abstraction.

Encapsulation

- *Binding (or wrapping) code and data together into a single unit are known as encapsulation*. For example, a capsule, it is wrapped with different medicines.
- A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Coupling

- Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other.
- If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field.
- You can use interfaces for the weaker coupling because there is no concrete implementation.

Cohesion

- Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method.
- The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface.

Association

- Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:
- One to One
- One to Many
- Many to One, and
- Many to Many.

Aggregation

- Aggregation is a way to achieve Association. Aggregation represents the relationship where one object contains other objects as a part of its state.
- It represents the weak relationship between objects. It is also termed as a *has-a* relationship in Java. Like, inheritance represents the *is-a* relationship.
- It is another way to reuse objects.

Composition

- The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state.
- There is a strong relationship between the containing object and the dependent object. It is the state where containing objects do not have an independent existence.
- If you delete the parent object, all the child objects will be deleted automatically.

What is Object-Oriented Programming:-

- **Object-oriented programming** has a sweeping impact because it appeals at multiple levels and promises faster and cheaper development and maintenance. It follows a bottom-up approach to develop applications.

Object-Oriented Programming

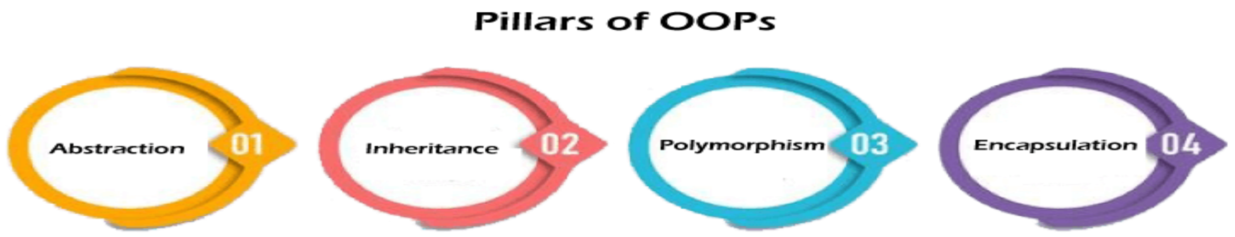
- The word **object-oriented** is the combination of two words i.e. **object** and **oriented**. The dictionary meaning of the object is an article or entity that exists in the real world.
- The meaning of oriented is interested in a particular kind of thing or entity. In layman's terms, it is a programming pattern that rounds around an object or entity are called **object-oriented programming.**
- The **object-oriented programming** is basically a computer programming design philosophy or methodology that organizes/ models software design around data, or objects rather than functions and logic.
- An object is referred to as a data field that has unique attributes and behavior. Everything in OOP is grouped as self-sustainable objects.
- It is the most popular programming model among developers. It is well suited for programs that are large, complex, and actively updated or maintained.
- It simplifies software development and maintenance by providing major concepts such as **abstraction, inheritance, polymorphism**, and **encapsulation**. These core concepts support OOP.
- A real-world example of OOP is the automobile. It more completely illustrates the power of object-oriented design.

Points to Remember

- Everything is an object
- Developer manipulates objects that uses message passing.
- Every object is an instance of a class.
- The class contains the attribute and behavior associated with an object.

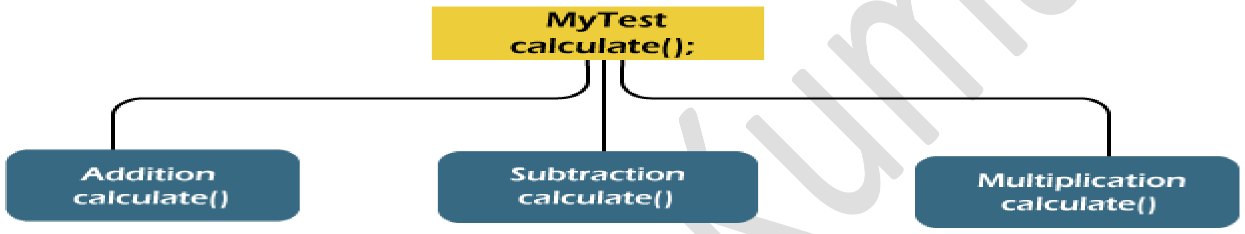
Pillars of OOPs

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism



Abstraction

- The concept allows us to hide the implementation from the user but shows only essential information to the user.
- Using the concept developer can easily make changes and added over time.

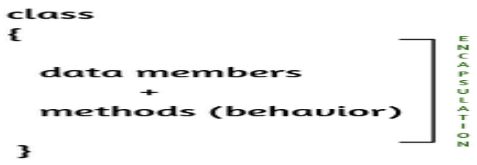


There are the following advantages of abstraction:

- It reduces complexity.
- It avoids delicacy.
- Eases the burden of maintenance
- Increase security and confidentially.

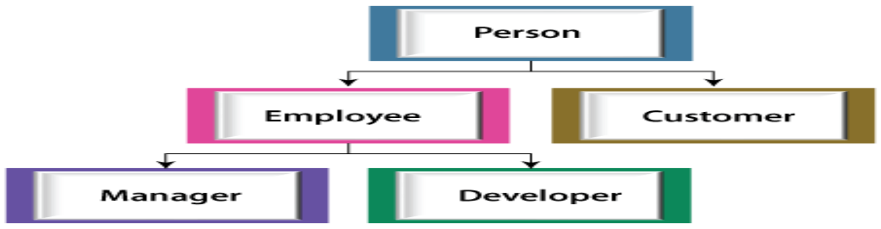
Encapsulation

- Encapsulation is a mechanism that allows us to bind data and functions of a class into an entity.
- It protects data and functions from outside interference and misuse. Therefore, it also provides security. A class is the best example of encapsulation.



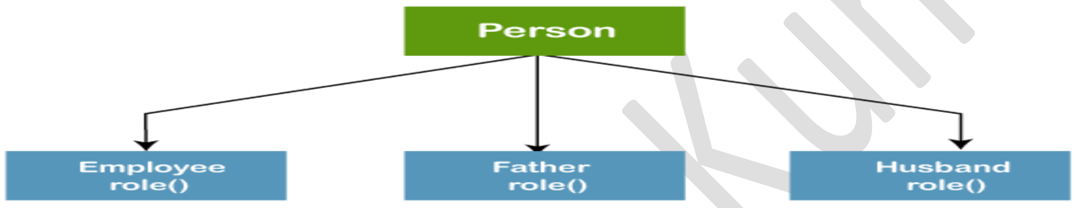
Inheritance

- The concept allows us to inherit or acquire the properties of an existing class (parent class) into a newly created class (child class).
- It is known as **inheritance**. It provides code reusability.



Polymorphism

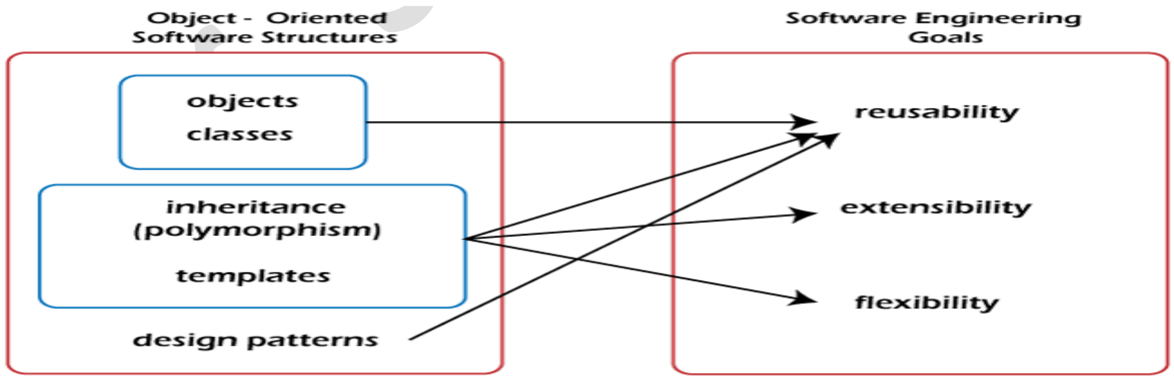
- The word **polymorphism** is derived from the two words i.e. **ploy** and **morphs**. Poly means many and morphs means forms.
- It allows us to create methods with the same name but different method signatures. It allows the developer to create clean, sensible, readable, and resilient code.



- The above figure best describes the concepts of polymorphism. A person plays an employee role in the office, father and husband role in the home.

Why should we use OOP?

- Object-oriented programming is an evolutionary development in software engineering. Using OOP in software development is a good habit because it accomplishes the three major software engineering goals, as we have shown in the following figure.


Object - Oriented Software Structures
objects
classes
inheritance (polymorphism)
templates
design patterns
Software Engineering Goals
reusability
extensibility
flexibility

Where it is used?

- OOP is often the best use when we are dealing with manufacturing and designing applications. It provides modularity in programming.

- It allows us to break down the software into chunks of small problems that we then can solve one object at a time.
- It should be used where the reusability of code and maintenance is a major concern. Because it makes development easy and we can easily append code without affecting other code blocks.
- It should be used where complex programming is a challenge.

Benefits of OOP

- Modular, scalable, extensible, reusable, and maintainable.
- It models the complex problem in a simple structure.
- Object can be used across the program.
- Code can be reused.
- We can easily modify, append code without affecting the other code blocs.
- Provides security through encapsulation and data hiding features.
- Beneficial to collaborative development in which a large project is divided into groups.
- Debugging is easy.

Limitations of OOP

- Requires intensive testing processes.
- Solving problems takes more time as compared to Procedure Oriented Programming.
- The size of the programs created using this approach may become larger than the programs written using the procedure-oriented programming approach.
- Software developed using this approach requires a substantial amount of pre-work and planning.
- OOP code is difficult to understand if you do not have the corresponding class documentation.
- In certain scenarios, these programs can consume a large amount of memory.
- Not suitable for small problems.
- Takes more time to solve problems.

Applications of OOPs

- Computer graphics applications
- Object-oriented database
- User-interface design such as windows
- Real-time systems
- Simulation and modeling
- Client-Server System
- Artificial Intelligence System
- CAD/CAM Software
- Office automation system

Introduction to Object-Oriented Thinking and Object-Oriented Programming (OOP):

- Object-Oriented Programming (OOP) is a programming paradigm that is based on the concept of "objects." It is a way of organizing and designing software in a more natural and intuitive manner, which closely mirrors the real-world entities and their interactions.
- OOP promotes the use of objects, which are instances of classes, to represent and manipulate data and the behaviors associated with that data.

Comparison with Procedural Programming:

- Procedural programming and Object-Oriented Programming are two different paradigms used to write computer programs:

Procedural Programming:

- In procedural programming, the program is organized around functions or procedures.
- Data is typically stored in data structures like arrays or structs, and functions operate on this data.
- It is less modular, making it harder to manage and scale as the program grows.
- Reusability of code is limited.
- Examples of procedural programming languages include C and Pascal.

Object-Oriented Programming:

- In OOP, the program is organized around objects that combine data and behaviors.
- Objects are instances of classes, and classes define the structure and behavior of objects.
- It promotes modularity and reusability, making it easier to manage and extend software.
- Encapsulation, inheritance, and polymorphism are key features of OOP.

Features of Object-Oriented Paradigm:

- **Classes and Objects:** The building blocks of OOP are classes and objects. Classes define the structure and behavior of objects, and objects are instances of classes.
- **Encapsulation:** Encapsulation is the concept of bundling data (attributes) and methods (functions) that operate on that data into a single unit, called a class. It hides the internal details of the class from the outside and exposes only what is necessary.
- **Inheritance:** Inheritance allows a class to inherit the properties and behaviors of another class. It promotes code reuse and hierarchy in class relationships.
- **Polymorphism:** Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables dynamic dispatch and flexibility in method calling.

- **Abstraction:** Abstraction is the process of simplifying complex systems by modeling classes based on the essential characteristics they possess. It focuses on what an object does, rather than how it does it.

Merits (Advantages) of OOP:

- **Modularity:** OOP promotes modular design, making it easier to manage, maintain, and extend software systems.
- **Reusability:** Code reusability is enhanced through the use of classes and inheritance.
- **Encapsulation:** Encapsulation provides data hiding, which improves security and data integrity.
- **Readability and Maintainability:** OOP code is often more readable and easier to maintain due to its natural representation of real-world entities.
- **Polymorphism:** Polymorphism allows for flexibility and adaptability in code design.

Demerits (Disadvantages) of OOP:

- **Complexity:** OOP can introduce additional complexity, especially for small programs that don't need the full range of OOP features.
- **Overhead:** There can be some overhead in terms of memory and processing when working with objects.
- **Learning Curve:** OOP can be more challenging for beginners to grasp compared to procedural programming.

Elements of OOP:

- **Class:** A blueprint or template for creating objects.
- **Object:** An instance of a class, representing a real-world entity with attributes and behaviors.
- **Inheritance:** The mechanism that allows a class to inherit properties and behaviors from another class.
- **Encapsulation:** The concept of bundling data and methods into a single unit, protecting data from external interference.
- **Polymorphism:** The ability of objects of different classes to be treated as objects of a common superclass.

Input/Output (I/O) Processing:

- Input/Output processing in programming involves the reading of data from external sources (input) and the writing of data to external destinations (output).
- It's a fundamental part of software development to interact with users, files, databases, and other systems. Here's an explanation of I/O processing in detail:

Input Processing:

- **Reading from Input Devices:** Input can be obtained from various sources, including keyboards, mice, sensors, and more.

- **Reading from Files:** Data can be read from files stored on the computer's storage devices.
- **Reading from Databases:** Data can be retrieved from databases using database management systems.
- **Reading from Network:** Data can be received from network connections, such as sockets or APIs.
- **Parsing and Validation:** Data from input sources often requires parsing and validation to ensure it is in the expected format and free from errors.

Output Processing:

- **Displaying to Output Devices:** Output can be presented to users through displays, monitors, printers, and other output devices.
- **Writing to Files:** Data can be written to files for storage or sharing.
- **Storing in Databases:** Data can be stored in databases for future retrieval.
- **Sending over Network:** Data can be transmitted to other systems over the network.
- **Formatting and Presentation:** Output data is often formatted and presented in a human-readable or machine-readable format, depending on the application's requirements.

Module – 2: Encapsulation and Data Abstraction

Introduction to Encapsulation and Data Abstraction:

- Encapsulation and data abstraction are fundamental concepts in Object-Oriented Programming (OOP) that help in organizing and structuring code to create more maintainable and understandable software.

Encapsulation:

- Encapsulation is the bundling of data (attributes) and the methods (functions) that operate on that data into a single unit called a class.
- It restricts direct access to some of the object's components, providing controlled access via methods. This is often achieved by using access modifiers like private and protected.
- Encapsulation helps hide the internal state of an object from the outside world and exposes only what is necessary, ensuring data integrity and security.

Data Abstraction:

- Data abstraction is the concept of displaying only essential features of an object while hiding the irrelevant details.
- It focuses on "what" an object does rather than "how" it does it.

- Abstraction allows you to create abstract data types, where the implementation details are hidden, making it easier to manage complex systems.

State, Behavior, and Identity of an Object:

- **State:** The state of an object represents the values of its attributes at a particular moment in time. For example, a "Car" object may have attributes like "color," "speed," and "fuel level."
- **Behavior:** The behavior of an object is defined by the methods or functions that can be applied to it. In the "Car" example, behaviors might include "accelerate," "brake," and "refuel."
- **Identity:** The identity of an object is a unique identifier that distinguishes it from other objects of the same class. This identity is often represented by a reference or memory address.

Classes:

- A class is a blueprint or template for creating objects. It defines the structure and behavior that objects of that class will have.
- Classes encapsulate both data (attributes) and methods (functions) that operate on that data.
    - For example, a "Car" class might define attributes like "color" and "speed" and methods like "accelerate" and "brake."

Identifying Classes and Candidates for Classes:

- Identifying classes is an important part of software design in OOP. To identify classes, consider the following:
- **Real-World Objects:** Look for objects and concepts in the problem domain that need to be represented in your software.
- **Responsibilities:** Identify what each class should do. Classes should have a clear and distinct purpose.
- **Attributes:** Determine the attributes (data) that each class needs to store.
- **Methods:** Identify the methods (behavior) that each class should have to interact with its attributes and perform its responsibilities.

Attributes and Services:

- Attributes are the data members of a class, representing the state of objects.
- Services are the methods or functions of a class, representing the behaviors of objects.

- For example, in a "Student" class, attributes might include "name," "age," and "student ID," while services might include methods like "enrollInCourse" and "submitAssignment."

Access Modifiers:

- Access modifiers control the visibility and accessibility of class members (attributes and methods). Common access modifiers include:
- **public:** Members are accessible from any part of the program.
- **private:** Members are only accessible within the class.
- **protected:** Members are accessible within the class and its subclasses.
- Access modifiers help enforce encapsulation and data hiding.

Static Members of a Class:

- Static members belong to the class itself, rather than to instances of the class.
- They are shared among all instances of the class and can be accessed using the class name.
- Common examples include static variables and static methods.

Instances:

- An instance is an object created from a class. It represents a specific realization of the class blueprint.
- Instances have their own unique state and identity while sharing the same behavior defined in the class.

Message Passing:

- In OOP, communication between objects is often done through message passing.
- Objects send messages to other objects to request them to perform specific actions.
- Messages typically involve calling methods on objects, which can result in changes to the object's state or the return of information.

Construction and Destruction of Objects:

- Construction of objects is the process of creating instances of a class using constructors. Constructors initialize the object's state and set it up for use.

- Destruction of objects is the process of releasing the resources associated with an object when it is no longer needed. In languages like C++, this is done using destructors.

Difference Between Abstraction and Encapsulation

Abstraction	**Encapsulation**
Abstraction is a feature of OOPs that hides the **unnecessary** detail but shows the essential information.	Encapsulation is also a feature of OOPs. It hides the code and data into a **single** entity or unit so that the data can be protected from the outside world.
It solves an issue at the **design** level.	Encapsulation solves an issue at **implementation** level.
It focuses on the **external** lookout.	It focuses on **internal** working.
It can be implemented using **abstract classes** and **interfaces**.	It can be implemented by using the **access modifiers** (private, public, protected).
It is the process of **gaining** information.	It is the process of **containing** the information.
In abstraction, we use **abstract classes** and **interfaces** to hide the code complexities.	We use the **getters** and **setters** methods to hide the data.
The objects are **encapsulated** that helps to perform abstraction.	The object need not to **abstract** that result in encapsulation.

Module – 3: Inheritance

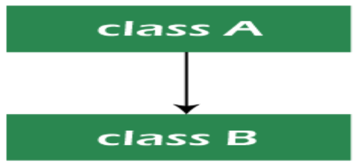
What is Inheritance:-

- The term 'Inheritance' is derived from the word "Inherit" which means, "to derive any quality, features or characteristics from family". Therefore,
- "***Inheritance is defined as a mechanism where the sub or child class inherits the properties and characteristics of the super class or other derived classes. It also supports additional features of extracting properties from the child class and using it into other derived classes.***"

- Inheritance is one of the most important concepts followed by Abstraction, Encapsulation and Polymorphism in the Object Oriented Programming (OOPS) Paradigm.

Types of Inheritance

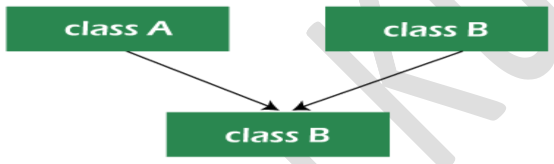
1. Single Inheritance:



Single Inheritance

- you will notice class One is the superclass, and class Two is the base class. Therefore, class Two inherits the properties and behaviour of the base class One.

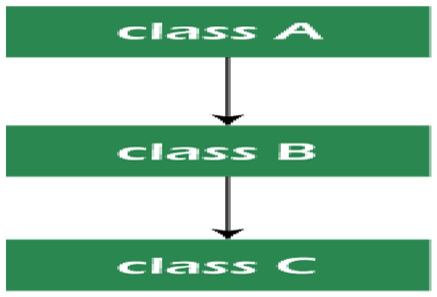
2. Multiple Inheritance:



Multiple Inheritance

- The child class inherits the properties and features from two or more parent classes with this type.
- In the below example, you will notice class Three inherits the features and behaviour of class Two. Further, class Two inherits the properties of class One. Therefore, we can conclude that class One is the base class of class Two, whereas class Two is the base class of class Three.
- Hence, class Three implicitly inherits the behaviour and properties of class One along with class Two, thereby creating a multiple inheritance.

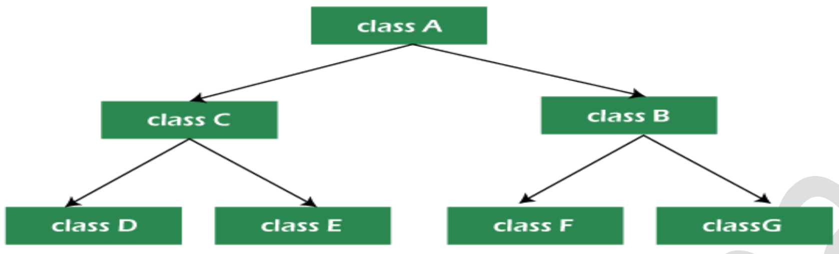
3. Multilevel Inheritance:



Multi - Level Inheritance

- class three inherits the properties, functions and objects for both class Two and class One at the same level. Therefore, both class One and class Two are the superclasses for class Three.

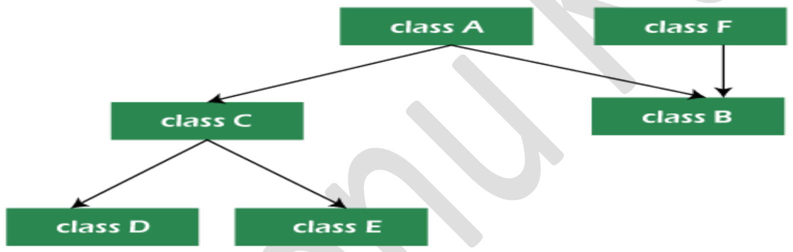
4. Hierarchal Inheritance:



Hierarchical Inheritance

(I) The base class One has two derived classes, i.e., class Two and class Three.
(II) Next, class Two acts as a base class for two child classes named Six and Seven.
(III) Further, class three is also a base class for class Four and class Five, respectively.

5. Hybrid Inheritance:



Hybrid Inheritance (a combination of Hierachical and multiple)

- With this type, the programmer can call a combination of two or more types of inheritance. Therefore the code can include a combination of Multiple and Hierarchical inheritance, Multilevel and Hierarchical inheritance or Hierarchical and Multipath inheritance, or it may be a combination of three of them,
- i.e., Multiple, Multilevel and Multiple Hierarchical inheritance.

Advantages of Inheritance.

- Recursive code is written once. Therefore, allowing code reusability.
- One base class can be used for one or more derived classes in a hierarchy.
- It saves time, as the programmer does not change the value in all the base classes; they change it in a parent class, and the base class inherits the change.
- OOPs Inheritance is used to create dominant objects.
- Inheritance prevents data duplicity and redundancy.
- Inheritance reduces space and time complexity.

What is the Need for Inheritance:-

1. Inheritance helps to prevent the rewriting of the same code again and again. The programmer can reuse the previously written code, override code (if necessary) and use it further in the program.
2. OOPs Inheritance mechanism helps to prevent code duplicity and data redundancy.
3. It also reduces the space and time complexity. Thus, making the program more efficient.
4. Inheritance makes the program accessible, as it involves a hierarchical programming paradigm.
5. The programmer can create any variable once and use the same variable multiple times within the scope of the code.
6. Programmers need Inheritance to create dominant data objects and program functions.

Relationship in Inheritance:

- Inheritance is a fundamental concept in Object-Oriented Programming (OOP) that defines a relationship between classes.
- It allows one class to inherit the attributes and behaviors of another class.
- Inheritance is used to model an "is-a" relationship, where a derived class (subclass or child class) "is a" specialized version of a base class (superclass or parent class).

Purpose of Inheritance:

- **Code Reusability:** Inheritance promotes code reuse. Instead of writing the same attributes and methods in multiple classes, you can define them in a base class and have multiple derived classes inherit those attributes and methods.
- **Specialization:** It allows you to create more specialized classes that build upon the general attributes and behaviors of a base class. This specialization helps model the "is-a" relationship between objects.
- **Hierarchy:** Inheritance helps create hierarchical relationships among classes, making it easier to organize and manage the structure of your code.

Types of Inheritance:

- **Single Inheritance:** In single inheritance, a class inherits from only one base class. For example, a class "Car" inheriting from a class "Vehicle."

- **Multiple Inheritance:** In multiple inheritance, a class can inherit from multiple base classes. Some programming languages, like C++, support multiple inheritance, while others, like Java, do not.
- **Multilevel Inheritance:** In multilevel inheritance, a class derives from a class that is itself derived from another class. It creates a chain of inheritance.
- **Hierarchical Inheritance:** In hierarchical inheritance, multiple classes inherit from a single base class. All derived classes share common attributes and behaviors from the same base class.
- **Hybrid Inheritance:** Hybrid inheritance combines multiple types of inheritance within a single program.

"Is-a" Relationship:

- Inheritance establishes an "is-a" relationship between a derived class and a base class.
- For example, if you have a base class "Animal," a derived class "Dog" can inherit from "Animal" because a dog "is an" animal.

Association:

- Association is a relationship in OOP where two or more classes are connected, but they are not part of each other.
- It can be a simple relationship, like one class using another as a parameter in a method, or a more complex relationship involving multiple classes interacting.

Aggregation:

- Aggregation is a specific form of association where one class contains or is composed of other classes. It represents a "whole-part" relationship.
- For example, a "University" class can have an aggregation relationship with a "Department" class because a university is composed of multiple departments.

Concept of Interfaces:

- An interface in OOP defines a contract of methods that a class must implement. It provides a way to achieve abstraction and multiple inheritance in languages that do not support multiple inheritance for classes (e.g., Java and C#). Key points about interfaces:
- Interfaces contain method signatures but no implementations.
- A class can implement multiple interfaces.
- Interfaces allow for polymorphism, where objects of different classes that implement the same interface can be treated interchangeably.

- Interfaces are used to define a common set of behaviors that various classes can adhere to.

Abstract Classes:

- An abstract class is a class that cannot be instantiated and is meant to be subclassed. Abstract classes are used when you want to provide a common base for a group of related classes. Key points about abstract classes:
- Abstract classes can have both abstract (unimplemented) methods and concrete (implemented) methods.
- Subclasses of an abstract class must provide implementations for all the abstract methods.
- Abstract classes are used to create a hierarchy of related classes with common attributes and behaviors.

Module – 4: Polymorphism

Polymorphism in Java

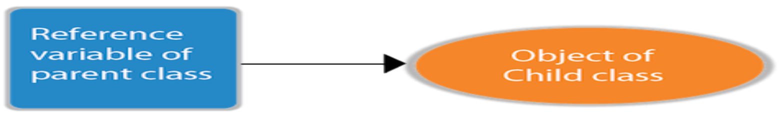
- **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs.
- The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- There are two types of polymorphism in Java: **compile-time polymorphism** and **runtime polymorphism.** We can perform polymorphism in java by method overloading and method overriding.
- If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

Runtime Polymorphism in Java

- **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
- an overridden method is called through the reference variable of a superclass.
- The determination of the method to be called is based on the object being referred to by the reference variable.
- Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

- If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:



```
class A{}
class B extends A{}
```

```
A a=new B();//upcasting
```

For upcasting, we can use the reference variable of class type or an interface type. For Example:

```
interface I{}
class A{}
class B extends A implements I{}
```

Here, the relationship of B class would be:

```
B IS-A A
B IS-A I
B IS-A Object
```

Since Object is the root class of all classes in Java, so we can write B IS-A Object.

Example of Java Runtime Polymorphism

- In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
class Bike{
  void run(){System.out.println("running");}
}
class Splendor extends Bike{
  void run(){System.out.println("running safely with 60km");}

  public static void main(String args[]){
    Bike b = new Splendor();//upcasting
    b.run();
  }
}
```

```
running safely with 60km.
```

Introduction to Polymorphism in OOP:

- Polymorphism is a fundamental concept in Object-Oriented Programming (OOP) that allows objects of different classes to be treated as objects of a common superclass.
- It provides a way to design more flexible and extensible software by enabling objects to exhibit different behaviors while sharing a common interface. Polymorphism simplifies code and enhances reusability.

There are two main forms of polymorphism in OOP:

- **Compile-Time (Static) Polymorphism:** This form of polymorphism is resolved at compile time, and it is related to method overloading.
- Method overloading allows multiple methods in the same class to have the same name but different parameter lists.
- **Run-Time (Dynamic) Polymorphism:** This form of polymorphism is resolved at run time, and it is related to method overriding.
- Method overriding allows a subclass to provide a specific implementation for a method that is already defined in its superclass.

Method Overloading:

- Method overloading is a form of static polymorphism where multiple methods in the same class have the same name but different parameters.
- Method overloading is based on the number, type, and order of parameters.

Key points about method overloading:

- Methods must have the same name.
- Methods must have different parameter lists (i.e., different types or number of parameters).
- Return types do not play a role in method overloading.

Example in Java:

```
public class Calculator {

   public int add(int a, int b) {

      return a + b;  {

   public double add(double a, double b) {

      return a + b;  } }
```

- In the above example, the add method is overloaded with different parameter types (int and double).

Method Overriding:

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

Key points about method overriding:

- The method in the subclass must have the same method signature (name, return type, parameters) as the method in the superclass.
- The method in the subclass should be annotated with the @Override annotation (in languages like Java) to indicate that it is intended to override a superclass method.

Example in Java:

```
class Shape {
   public void draw() {

      System.out.println("Drawing a shape");    }  }

class Circle extends Shape {

   @Override

   public void draw() {

      System.out.println("Drawing a circle");    } }
```

- the draw method in the Circle class overrides the draw method in the Shape class.

Static Polymorphism (Method Overloading):

- Static polymorphism, also known as compile-time polymorphism, occurs when the decision about which method to call is made at compile time.
- It is determined by the number and types of arguments passed to a method.
- The method to be executed is known at compile time, and there is no method resolution at runtime.

Run-Time Polymorphism (Method Overriding):

- Run-time polymorphism, also known as dynamic polymorphism, occurs when the decision about which method to call is made at runtime.
- It is determined by the actual type of object at runtime. The method to be executed is resolved at runtime, making it more flexible and adaptable.

Module – 5: String In OOPs

Java String

- In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string.

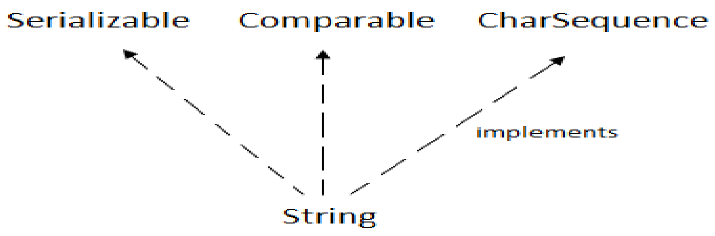
For example:

1. **char**[] ch={'j','a','v','a','t','p','o','i','n','t'};
2. String s=**new** String(ch);

is same as:

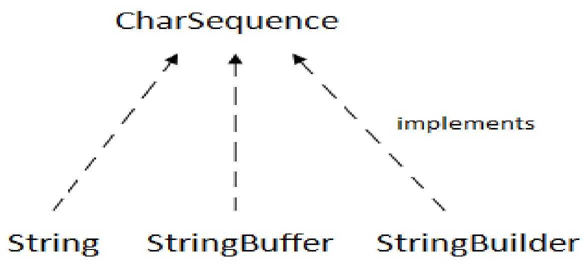
1. String s="javatpoint";

- **Java String** class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.



Char Sequence Interface

- The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it.
- It means, we can create strings in Java by using these three classes.



- The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use String Buffer and String Builder classes.
- We will discuss immutable string later. Let's first understand what String in Java is and how to create the String object.

What is String in Java:-

- Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters.
- The java.lang.String class is used to create a string object.

How to create a string object?

There are two ways to create String object:

1. By string literal
2. By new keyword

1) String Literal

- Java String literal is created by using double quotes.

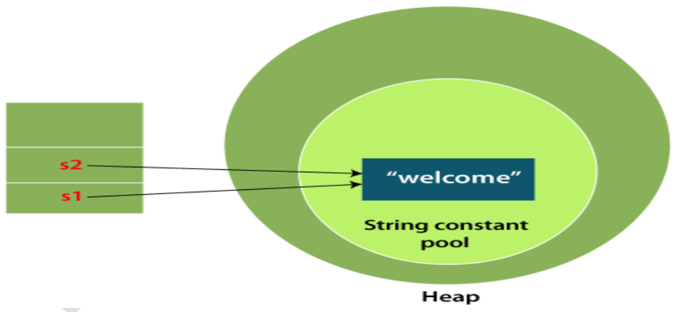
For Example:

```
1. String s="welcome";
```

- Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool.

For example:

```
1. String s1="Welcome";
2. String s2="Welcome";//It doesn't create a new instance
```



- In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool that is why it will create a new object.
- After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

Note: String objects are stored in a special memory area known as the "string constant pool".

Why Java uses the concept of String literal?

- To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

2) By new keyword

```
String s=new String("Welcome");//creates two objects and one reference variable
```

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

Java String Example

StringExample.java

```
public class StringExample{
public static void main(String args[]){
String s1="java";//creating string by Java string literal
char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);//converting char array to string
String s3=new String("example");//creating Java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

Test it Now

Output:

```
java
strings
example
```

- The above code, converts a ***char*** array into a **String** object. And displays the String objects ***s1, s2***, and ***s3*** on console using ***println()*** method.

Introduction to String:

- In Object-Oriented Programming (OOP), a string is a sequence of characters that represents text. Strings are a fundamental data type and are often represented as objects in OOP languages.
- Strings allow for the manipulation, storage, and processing of textual data. They provide various operations for concatenation, splitting, searching, and modification of text.

Example in Java:

String text = "Hello, World!";

String modifiedText = text.replace("Hello", "Hi"); // Returns a new string with "Hi, World!"

Exception Handling:

- Exception handling is a crucial aspect of OOP and programming in general. It is a mechanism that allows a program to deal with unexpected or erroneous situations that may occur during execution.
- Exceptions represent various types of errors or exceptional conditions.

Key components of exception handling:

- **Try-Catch Blocks:** Code that may raise an exception is enclosed within a try block. If an exception is raised, it is caught and handled in a catch block.
- **Throwing Exceptions:** Exceptions can be explicitly thrown using the throw statement when an error is detected.
- **Exception Types:** Programming languages define a hierarchy of exception types. Common exception types include NullPointerException, FileNotFoundException, ArithmeticException, and custom exception types created by programmers.
- **Exception Handling Strategies:** Handling exceptions can involve logging the error, providing alternative behavior, or gracefully terminating the program.

Example in Java:

```
try {

   int result = 10 / 0; // Division by zero raises an ArithmeticException

} catch (ArithmeticException ex) {

   System.out.println("An error occurred: " + ex.getMessage());  }
```

Introduction to Multi-Threading:

- Multi-threading is a technique that allows a program to execute multiple threads (smaller units of a program) concurrently, taking advantage of modern multi-core processors.
- Each thread represents an independent flow of execution within a program.
- Multi-threading is used to achieve parallelism, improve performance, and support responsive user interfaces.

Key concepts in multi-threading:

- **Thread:** A thread is the smallest unit of execution in a program. Multiple threads can run concurrently.
- **Concurrency:** Concurrency is the ability to run multiple threads simultaneously, which can lead to better resource utilization.

- **Thread Safety:** Ensuring that multiple threads can access shared data or resources without causing data corruption or inconsistencies is an essential aspect of multi-threading.
- **Synchronization:** Synchronization mechanisms like locks and semaphores are used to control access to shared resources and prevent race conditions.

Example in Java:

```
class MyThread extends Thread {
   public void run() {
      System.out.println("Thread is running.");   } }
public class Main {
   public static void main(String[] args) {
      MyThread thread1 = new MyThread();
      MyThread thread2 = new MyThread();
      thread1.start(); // Start the first thread
      thread2.start(); // Start the second thread   } }
```

Introduction to Data Collections:

- Data collections, often referred to as data structures, are fundamental in OOP and programming.
- They provide a way to organize and store data efficiently. Common data collections include arrays, lists, sets, maps, and queues.

Key types of data collections:

- **Arrays:** A fixed-size collection of elements of the same type.
- **Lists:** Dynamic collections that can grow or shrink. Examples include lists, linked lists, and ArrayLists.
- **Sets:** Collections of unique elements. Examples include sets and hash sets.
- **Maps:** Key-value pairs where keys are unique. Examples include dictionaries and hash maps.
- **Queues:** Collections that follow the "first-in, first-out" (FIFO) order. Examples include queues and priority queues.

Case Study - ATM System (Object-Oriented Approach):

Classes: ATM, Bank, Account

- **Attributes:** PIN, balance, account number, ATM ID, bank name, transaction history, etc.
- **Methods:** Withdraw, deposit, check balance, change PIN, log transaction, etc.
- **Inheritance:** The Account class can inherit from a base Bank class.

- **Polymorphism:** Different types of bank accounts (e.g., savings, checking) can exhibit different behaviors while sharing a common interface.
- **Encapsulation:** PIN and balance are private attributes, and methods provide controlled access to these attributes.
- **Exception Handling:** Handle cases like insufficient balance or incorrect PIN.

Case Study - Library Management System (Object-Oriented Approach):

Classes: Library, Book, Member

- **Attributes:** Book title, author, due date, member ID, book ID, fines, etc.
- **Methods:** Check out a book, return a book, calculate fines, list available books, etc.
- **Inheritance:** Different types of books (e.g., fiction, non-fiction) can inherit from a base Book class.
- **Polymorphism:** Different books and library members can exhibit different behaviors while sharing a common interface.
- **Encapsulation:** Private attributes like member ID and book ID, with controlled access via methods.
- **Exception Handling:** Handle cases like late returns or unavailable books.

CS305-Object Oriented Programming &Methodology Sem- 3 CSE RGPV

By:- Mr. Sonu Kumar

Contact Number :- 6200638476

Thank You !!!