INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal, Hyderabad - 500 043

Program &Class : B.Tech (CSE) & E – Section **Sem & Year : I & I**

Course code & Course Name : ACSD01 - Object Oriented Programming

Faculty : Dr.S.Sathees Kumar (IARE10907)

ANSWERS FOR TUTORIAL QUESTION BANK

MODULE -II

CLASSES AND OBJECTS

PART A-PROBLEM SOLVING AND CRITICAL THINKING QUESTIONS

1. What is the need for object-oriented programming?

- Object-oriented programming (OOP) is a programming paradigm that uses objects, which are instances of classes, for organizing and structuring code. The need for OOP arises from several advantages it offers in terms of code organization, reusability, modularity, and abstraction.
- **Modularity:**
    - OOP promotes modularity by breaking down a complex system into smaller, self-contained modules (classes). Each class represents a module with a specific responsibility, making it easier to understand, maintain, and modify code.
- **Reusability:**
    - OOP encourages the creation of reusable components in the form of classes and objects. Once a class is defined, it can be instantiated multiple times, and the same class can be used in different parts of an application or in different applications.
- **Abstraction:**
    - Abstraction allows developers to represent real-world entities and their interactions in a simplified manner. Classes abstract the essential characteristics of an entity, and objects provide a concrete instantiation of those characteristics.
- **Encapsulation:**
    - Encapsulation involves bundling the data (attributes) and the methods (functions) that operate on the data into a single unit called a class. This provides data hiding and protects the internal implementation details of a class, making it more maintainable.

- **Inheritance:**
    - Inheritance allows the creation of a new class (subclass) based on an existing class (superclass). This promotes code reuse and helps in modeling relationships between classes. Subclasses inherit the attributes and behaviors of their superclasses.
- **Polymorphism:**
    - Polymorphism enables a single interface to be used for different types of objects. This allows code to be more flexible and adaptable, as methods can be designed to work with objects of different classes that share a common interface.
- **Ease of Maintenance:**
    - OOP promotes code organization and structuring, making it easier to maintain and update code. Changes to one part of the system do not necessarily affect other parts if encapsulation is properly implemented.
- **Real-world Modelling:**
    - OOP allows developers to model real-world entities and their relationships directly in the code. This makes it easier to understand and translate real-world scenarios into code, leading to more intuitive and natural code design.
- **Collaborative Development:**
    - OOP facilitates collaborative development by allowing multiple developers to work on different parts of a system simultaneously. Each developer can focus on a specific class or module without interfering with others, promoting parallel development.
- **Code Extensibility:**
    - OOP provides a framework for building extensible systems. New classes and functionality can be added without modifying existing code, making it easier to extend the capabilities of a system.

2. is it always necessary to create objects from class?

In object-oriented programming (OOP), it is not always necessary to create objects from a class. The necessity of creating objects from a class depends on the design and purpose of the class and how it fits into your overall program or system.

a. Static Members and Methods: A class can contain static members (fields and methods) that are associated with the class itself rather than with instances of the class. We can access static members and methods using the class name without creating objects.

```
class MathUtils {

    public static int add(int a, int b) {

        return a + b;

    }

}

// Usage without creating an object

int result = MathUtils.add(3, 5);
```

b. Singleton Pattern: In some cases, you may design a class as a Singleton to ensure that only one instance of the class exists throughout the application's lifecycle. Singleton classes provide a single point of access to that instance.

```
class Singleton {

    private static final Singleton instance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {

        return instance;

    }

}

// Access the single instance

Singleton singleInstance = Singleton.getInstance();
```

c. Factory Methods: Classes can have factory methods that create and return instances of objects. These methods encapsulate the object creation process.

```
class Car {

    private String model;
    private Car(String model) {

        this.model = model;

    }

    public static Car createCar(String model) {

        return new Car(model);

    }

}

// Create a Car object using a factory method

Car myCar = Car.createCar("Toyota");
```

d. Abstract Classes and Interfaces: Classes can be designed as abstract classes or interfaces, which are not meant to be instantiated directly. Instead, they serve as blueprints for concrete classes to inherit from or implement.

e. Stateless Utility Classes: You can create utility classes with only static methods and constants. These classes are not intended to be instantiated and provide utility functions that can be called directly from the class itself.

3. Are class and structure the same? If not, what is the difference between a class and a structure?

Structure is a value type, while classes are reference types.

Class :

- It is defined using 'class' keyword.
- When data is defined in a class, it is stored in memory as a reference.
- It gets memory allocated only when an object of that class is created.
- The reference type (before creating an object) is allocated on heap memory.
- They can have constructors and destructors.
- It can use inheritance to inherit properties from base class.
- The 'protected' access modifier can be used with the data members defined inside the class.

Structure :

- The 'struct' keyword is used to define a structure.
- Every member in the structure is provided with a unique memory location.

- When the value of one data member is changed, it doesn't affect other data members in structure.
- It helps initialize multiple members at once.
- Total size of the structure is equivalent to the sum of the size of every data member.
- It is used to store various data types.
- It takes memory for every member which is present within the structure.
- A member can be retrieved at a time.
- It supports flexible arrays.
- Its instance can be created without a keyword.
- It doesn't support protected access modifier.
- It doesn't support inheritance.
- It doesn't have a constructor or destructor.
- The values allocated to structures are stored in stack memory.

4. Can a class extend itself.

No, a class cannot extend itself.

```
public class Main extends Main
{
    String name = "Jai";
    int rollNo = 11;
    void show(){
        System.out.println(name);
        System.out.println(rollNo);
    }
    public static void main(String[] args)
    {
        Main object = new Main();
        object.show();
    }
}
```

Output

```
Main.java:8: error: cyclic inheritance involving Main
public class Main extends Main
```

5. What is the difference between a class and object?

Class	Object
• A class is a template for creating objects in program.	• The object is an instance of a class.
• A class is a logical entity	• Object is a physical entity
• A class does not allocate memory space when it is created.	• Object allocates memory space whenever they are created.
• You can declare class only once.	• You can create more than one object using a class.
• Example: Car.	• Example: Jaguar, BMW, Tesla, etc.
• Class generates objects	• Objects provide life to the class.
• Classes can't be manipulated as they are not available in memory.	• They can be manipulated.
• It doesn't have any values which are associated with the fields.	• Each and every object has its own values, which are associated with the fields.
• You can create class using "class" keyword.	• You can create object using "new" keyword in Java

6. What is the need of access specifiers?

1. **Encapsulation:**
    - Access specifiers help in achieving encapsulation, one of the fundamental principles of object-oriented programming (OOP).
    - By specifying access levels, you control which parts of your code are accessible from outside and which are hidden. This promotes data hiding and protects the internal implementation of a class.

2. **Security:**
    - Access specifiers enhance the security of a program by restricting access to sensitive data and methods.
    - For example, private members can only be accessed within the same class, preventing external classes from directly modifying or accessing critical data.
3. **Modularity:**
    - Access specifiers aid in creating modular and maintainable code. They allow you to hide the implementation details of a class from the outside world, reducing dependencies between different parts of your program.

4. **Code Organization:**
    - Public, private, and protected members provide a clear structure to your code, making it more readable and understandable.
    - Public members define the interface of a class, while private members are hidden and can be modified only within the class.
5. **Abstraction:**
    - Access specifiers support the abstraction of complex systems by exposing only essential functionalities to external components.
    - Abstraction allows developers to focus on using the class without worrying about its internal complexities.
6. **Preventing Unintended Modifications:**
    - By marking certain members as private, you prevent unintended modifications by external classes.
    - This helps in avoiding bugs and unintended consequences that may arise from changes made to internal implementations.
7. **Inheritance and Polymorphism:**
    - Access specifiers are essential in inheritance. They determine how members of a base class are inherited by derived classes.
    - They also play a role in polymorphism, allowing you to override methods in derived classes while maintaining proper access levels.

7. What will happen if a class extends two interfaces and they both have a method with same name and signature?

Yes, a class can implement two interfaces with the same method signature but that method should be implemented only once in the class.

```
interface Test1 {
    void show();
}
interface Test2 {
    void show();
}
public class Main
{
    void show(){
        System.out.println("Implemented method.");
    }

    public static void main(String[] args)
    {
        Main object = new Main();
        object.show();
    }
}
```

Output

```
Implemented method.
```

8. Which access specifier should be used in a class where the instances can't be created?

All the constructors must be made private. This will restrict the instance of class to be made anywhere in the program. Since the constructors are private, no instance will be able to call them and hence won't be allocated with any memory space. Private constructors **allow us to restrict the instantiation of a class**. Simply put, they prevent the creation of class instances in any place other than the class itself.

Public and private constructors, used together, allow control over how we wish to instantiate our classes – this is known as constructor delegation.

Using Private Constructors to Create Uninstantiable Classes

Uninstantiable classes are classes that we cannot instantiate. In this example, we'll create **a class that simply contains a collection of static methods**:

```
public class StringUtils {
private StringUtils() { // this class cannot be instantiated
  }
Public static String toUpperCase(String s) {
return s.toUpperCase();
  }
public static String `LowerCase(String s) {
return s.toLowerCase();
  }
}
```

The ***StringUtils*** class contains a couple of static utility methods and can't be instantiated due to the private constructor.

Really, there's no need to allow object instantiation since static methods don't require an object instance to be used.

On which specifier's data, does the size of a class's object depend?

The size of a class's object does not directly depend on the access specifiers used for its members. The primary factors influencing the size of an object in Java include

1. **Data Members:**
   - The size of each data member (fields or variables) within the class contributes to the overall size of the object. The size of each data type (e.g., int, double, Object) is platform-dependent.
2. **Object Header:**
   - Each object in Java has an object header that contains information such as the object's identity hash code, a reference to the object's class, and any synchronization-related flags.
   - The size of the object header is platform-dependent.
3. **Padding and Alignment:**
   - Similar to other languages, Java may introduce padding between data members for alignment purposes. The alignment is done to improve memory access efficiency.
   - For example, an object might be aligned to a multiple of 8 bytes.
4. **Reference Size:**
   - If the class has reference-type members (references to other objects), the size of those references contributes to the overall size of the object.
   - The size of a reference is platform-dependent. In a 32-bit JVM, a reference typically takes 4 bytes, while in a 64-bit JVM, it takes 8 bytes.
5. **Internal Overheads:**
   - There might be internal overhead associated with the garbage collection and memory management system of the Java Virtual Machine (JVM), but these details are implementation-specific.
6. **Inheritance:**
   - In the case of inheritance, the size of a subclass object includes the size of its superclass. The exact layout is influenced by factors like access modifiers (public, private, protected) and the presence of static and final members.
7. **Virtual Methods and Interface Methods:**
   - If the class has virtual methods (methods marked as final or abstract), there may be additional space for method tables or other information related to method dispatch.
   - Interface methods may contribute to the size of the object, depending on the specifics of the JVM implementation.

10. What is the difference between local variable and data member?

Data member/ Instance Variable	**Local Variable**
• They are defined in class but outside the body of methods.	• They are defined as a type of variable declared within programming blocks or subroutines.

Data member/ Instance Variable	Local Variable
• These variables are created when an object is instantiated and are accessible to all constructors, methods, or blocks in class.	• These variables are created when a block, method or constructor is started and the variable will be destroyed once it exits the block, method, or constructor.
• These variables are destroyed when the object is destroyed.	• These variables are destroyed when the constructor or method is exited.
• It can be accessed throughout the class.	• Its access is limited to the method in which it is declared.
• They are used to reserving memory for data that the class needs and that too for the lifetime of the object.	• They are used to decreasing dependencies between components I.e., the complexity of code is decreased.
• These variables are given a default value if it is not assigned by code.	• These variables do not always have some value, so there must be a value assigned by code.
• It is not compulsory to initialize instance variables before use.	• It is important to initialize local variables before use.
• It includes access modifiers such as private, public, protected, etc.	• It does not include any access modifiers such as private, public, protected, etc.

PART B-LONG ANSWER QUESTIONS

1. Define class and object with suitable example. How members of class can be accessed?

Class:

In object-oriented programming, a class is a blueprint or a template for creating objects. It defines a set of attributes (data members) and behaviors (methods) that the objects created from the class will have. A class serves as a model for objects, encapsulating their properties and functionalities.

Example of a Class:

```
// Defining a simple class named "Car"
public class Car {
   // Data members (attributes)
   String model;
   String color;
   int year;

   // Methods (behaviors)
   public void start() {
      System.out.println("The car is starting.");
   }

   public void drive() {
      System.out.println("The car is in motion.");
   }

   public void stop() {
      System.out.println("The car has stopped.");
   }
}
```

In this example, the **Car** class has data members **(model, color, year)** representing the car's attributes, and it has methods **(start, drive, stop)** representing the car's behaviors.

Object:

An object is an instance of a class. It is a tangible entity that is created based on the structure defined by the class. Objects have their own set of data members (attributes), and they can perform actions through the methods defined in the class.

Example of Creating and Using Objects:

```
public class Main {

   public static void main(String[] args) {

      // Creating objects of the Car class

      Car myCar = new Car();

      Car friendCar = new Car();

      // Setting attributes for myCar

      myCar.model = "Toyota Camry";

      myCar.color = "Blue";

      myCar.year = 2022;

      // Setting attributes for friendCar

      friendCar.model = "Honda Civic";

      friendCar.color = "Red";

      friendCar.year = 2021;

      // Accessing methods to perform actions

      myCar.start();

      myCar.drive();

      myCar.stop();
```

```
        friendCar.start();

        friendCar.drive();

        friendCar.stop();

    }

}
```

In this example, **myCar** and **friendCar** are objects created from the Car class. Each object has its own set of attributes, and the methods can be invoked to perform actions specific to each object.

Accessing Members of a Class:

Members of a class (both data members and methods) can be accessed using the dot notation **('.')** For example:

```
// Accessing data members

myCar.model = "Toyota Camry";

System.out.println("Car model: " + myCar.model);

// Accessing methods

myCar.start();
```

The dot notation allows you to access and manipulate the attributes and behaviors of an object created from a class. The access modifiers (public, private, protected) can be used to control the visibility and accessibility of class members from outside the class.

2. What is a method? How a method is defined?

Method

In programming, a method is a set of code that performs a specific task or action. It is a block of reusable code that is defined within a class or an object and can be called to perform a particular operation. Methods are also referred to as functions or procedures in other programming languages.

How a Method is Defined?

The definition of a method typically includes the following components:

1. **Method Signature:**
    - The method signature is the declaration of the method, including its name, return type, and parameters (if any).

The syntax for a method signature is as follows:

```
returnType methodName(parameter1Type parameter1Name, parameter2Type
parameter2Name, ...) {
   // Method body
}
```

2. **Return Type:**
    - The return type specifies the type of value that the method will return after its execution. It can be a primitive data type (e.g., int, double) or an object type (e.g., String, CustomObject).
    - If a method does not return any value, the return type is specified as void.
3. **Method Name:**
    - The method name is a unique identifier for the method within the class or object. It is used to call or invoke the method.
    - Method names follow the same rules as variable names, such as starting with a letter, being case-sensitive, and avoiding reserved keywords.
4. **Parameters:**
    - Parameters (also known as arguments) are values that are passed into the method for it to use during its execution. A method can have zero or more parameters.
    - Each parameter consists of a data type and a parameter name.
5. **Method Body:**
    - The method body is enclosed in curly braces {} and contains the code that defines the behaviour of the method.
    - This is where you write the instructions that the method will execute when called.

Example of a Method Definition:

```
public class MathOperations {

   // Method definition with parameters and a return type

   public static int add(int num1, int num2) {

      int sum = num1 + num2;

      return sum; // Returning the result

   }
```

```
    // Method definition with no parameters and a return type
    public static void displayMessage() {
        System.out.println("Hello, this is a simple message.");
    }

    public static void main(String[] args) {
        // Calling the methods
        int result = add(5, 7);
        System.out.println("Sum: " + result);

        displayMessage();
    }
}
```

In this example, the **MathOperations** class defines two methods:

1. The **add** method takes two parameters (**num1** and **num2**), adds them, and returns the sum.
2. The displayMessage method has no parameters, does not return a value (**void**), and simply prints a message.

When calling methods, you use their names along with the appropriate arguments if required. In the **main** method, **add(5, 7)** and **displayMessage()** are examples of method calls.

3. What are access specifiers and what is their significance in OOP?

Access specifiers (also known as access modifiers) in object-oriented programming (OOP) are keywords that define the visibility and accessibility of class members (fields, methods, and nested classes) from outside the class. These specifiers play a crucial role in encapsulation and control the level of exposure of class internals to the external world. In most OOP languages, there are typically three main access specifiers:

1. **Public (public):**
    - Members declared as public are accessible from any other class.

    - There are no restrictions on accessing public members.
2. **Private (private):**
    - Members declared as private are accessible only within the class that declares them.
    - They are not visible to other classes, even if those classes are in the same package.
3. **Protected (protected):**
    - Members declared as protected are accessible within the class, its subclasses, and classes in the same package.
    - They are not accessible from classes outside the package that are not subclasses.

Significance of Access Specifiers in OOP:

1. **Encapsulation:**
    - Access specifiers facilitate encapsulation by controlling the visibility of class members.
    - Encapsulation is the concept of bundling data and methods that operate on that data within a single unit (class), and restricting direct access to the internal details of the class.
2. **Information Hiding:**
    - Private members are hidden from outside classes, providing a level of information hiding.
    - This prevents external classes from directly manipulating or accessing the internal state of an object, reducing the risk of unintended side effects.
3. **Modularity:**
    - Access specifiers help in creating modular and maintainable code.
    - By designating certain members as private, a class can expose only the necessary interfaces to the external world, promoting a clear separation between the implementation details and the public interface.
4. **Security:**
    - Access specifiers contribute to the security of an application by restricting unauthorized access to sensitive data or methods.
    - For example, sensitive data can be kept private, and controlled access to it can be provided through public methods.
5. **Inheritance and Polymorphism:**
    - Access specifiers play a role in inheritance and polymorphism, influencing the visibility of members in derived classes.
    - Protected members are often used to provide access to derived classes without exposing them to the entire world.

Example:

```
public class Example {
    private int privateField;
    public int publicField;
    protected int protectedField;
    private void privateMethod() {
        // implementation
    }
    public void publicMethod() {
        // implementation
    }
    protected void protectedMethod() {
        // implementation
    }
}
```

In this example:

- **privateField** and **privateMethod** are accessible only within the Example class.
- **publicField** and **publicMethod** are accessible from any class.
- **protectedField** and **protectedMethod** are accessible within the class, its subclasses, and classes in the same package.

4. What is the difference between public, private and protected access modifiers?

In object-oriented programming, public, private, and protected are access modifiers used to control the visibility and accessibility of class members (fields, methods, and nested classes). Here's a summary of the differences between these access modifiers:

1. **Public (public):**
    - **Visibility:** Public members are accessible from any other class.
    - **Scope:** There are no restrictions on accessing public members.

Example

```
public class Example {

   public int publicField;

   public void publicMethod() {

      // implementation

   }

}
```

Private (private):

- **Visibility:** Private members are accessible only within the class that declares them.
- **Scope:** They are not visible to other classes, even if those classes are in the same package.

Example

```
    public class Example {
       private int privateField;
       private void privateMethod() {
          // implementation

       }
    }
```

Protected (protected):

- **Visibility**: Protected members are accessible within the class, its subclasses, and classes in the same package.
- **Scope:** They are not accessible from classes outside the package that are not subclasses.

Example

```
public class Example {

   protected int protectedField;

   protected void protectedMethod() {

      // implementation

   }

}
```

Comparison:

- **Public:**
  - Provides the highest level of visibility.
  - Suitable for members that need to be accessed from any part of the program.
  - Often used for class interfaces.
- **Private:**
  - Provides the highest level of encapsulation and information hiding.
  - Restricts access to the declaring class only.
  - Used to protect sensitive data and hide implementation details.
- **Protected:**
  - Offers a compromise between public and private.
  - Accessible within the class, its subclasses, and classes in the same package.
  - Used when you want to provide access to subclasses but not to the entire world.

Example

```
public class Vehicle {

   // Public member

   public String model;


   // Private member

   private int speed;


   // Protected member

   protected void accelerate() {

      // implementation

   }

}


public class Car extends Vehicle {

   public void increaseSpeed() {

      // Accessing protected member from the superclass

      accelerate();
```

```
    }
}
```

In this example, **model** is public and can be accessed from any class. **speed** is private and is only accessible within the **Vehicle** class. The **accelerate** method is protected and can be accessed by subclasses like **Car** but not by unrelated classes outside the package.

5. Create a real scenario where static data members are useful. Explain with suitable example.

Let's consider a scenario where you want to keep track of the total sales made by a retail store. Each time a sale is made, you want to update the total sales for all instances of the **PointOfSale** class. A static data member can be used to achieve this.

```
public class PointOfSale {
    // Static data member to store the total sales for all instances
    private static double totalSales = 0;

    // Instance data members
    private String productName;
    private double saleAmount;
    // Constructor
    public PointOfSale(String productName, double saleAmount) {
        this.productName = productName;
        this.saleAmount = saleAmount;

        // Increment the totalSales every time a sale is made
        totalSales += saleAmount;
    }
    // Static method to get the total sales
    public static double getTotalSales() {
        return totalSales;
    }

    // Instance method to display sale information

    public void displaySaleInfo() {

        System.out.println("Product: " + productName + ", Sale Amount: $" +
saleAmount);

    }

}
```

In this example:

- **totalSales** is a static data member that keeps track of the total sales made by all instances of the **PointOfSale** class.
- The constructor increments **totalSales** every time a new sale is made.
- The **getTotalSales** static method allows you to retrieve the total sales.

Now, let's use this PointOfSale class in a simple program:

```
public class RetailStore {

    public static void main(String[] args) {

        // Creating PointOfSale instances (representing sales)

        PointOfSale sale1 = new PointOfSale("Laptop", 1200.50);

        PointOfSale sale2 = new PointOfSale("Smartphone", 699.99);

        PointOfSale sale3 = new PointOfSale("Headphones", 89.99);

        // Displaying sale information

        sale1.displaySaleInfo();

        sale2.displaySaleInfo();

        sale3.displaySaleInfo();

        // Retrieve the total sales using the static method

        double totalSales = PointOfSale.getTotalSales();

        System.out.println("Total Sales: $" + totalSales);

    }

}
```

In this scenario, the static data member **totalSales** is shared among all instances of the **PointOfSale** class. Each time a sale is made, the total sales are updated and can be accessed globally using the static method **getTotalSales()**. This allows you to keep a centralized record of the overall sales made by the retail store.

6. Explain the scope of access specifiers.

Access specifiers in object-oriented programming determine the scope or visibility of class members (fields, methods, and nested classes) within a class or between classes. The scope defines where a particular member can be accessed and modified. The three main access specifiers are public, private, and protected. Let's explore the scope of each:

Public (public):

- **Scope:**
  - Public members are accessible from any other class.
  - They can be accessed globally, both within and outside the package.

Example:

```
public class Example {

   public int publicField;

   public void publicMethod() {

      // implementation

   }

}
```

Private (private):

- **Scope:**
  - Private members are accessible only within the class that declares them.
  - They are not visible to other classes, even if those classes are in the same package.

Example:

```
public class Example {

   private int privateField;

   private void privateMethod() {

      // implementation

   } }
```

Protected (protected):

- **Scope:**
    - Protected members are accessible within the class, its subclasses, and classes in the same package.
    - They are not accessible from classes outside the package that are not subclasses.

Example:

```
public class Example {

    protected int protectedField;

    protected void protectedMethod() {

        // implementation

    }

}

public class Example {

    protected int protectedField;

    protected void protectedMethod() {

        // implementation

    }

}
```

Scope Summary:

- **Within the Same Class:**
    - Public, private, and protected members are accessible within the class itself.
- **Within the Same Package:**
    - Public and protected members are accessible from other classes in the same package.
    - Private members are not accessible from other classes in the same package.
- **Outside the Package (Global):**
    - Public members are accessible globally, both within and outside the package.
    - Protected members are accessible if the accessing class is a subclass, regardless of the package.

- Private members are not accessible from outside the class.

- **Access Specifiers and Inheritance:**
  - Public and protected members support inheritance, meaning they can be inherited by subclasses.
  - Private members are not directly inherited, but they can indirectly affect subclasses through public or protected methods.

7. Draw a class diagram for hospital management system which consists of classes, their attributes, operations (or methods), and relationship among objects. The main classes are hospitals, patients, doctors, nurses, appointments, and medicines.

Classes of hospital management system class diagram

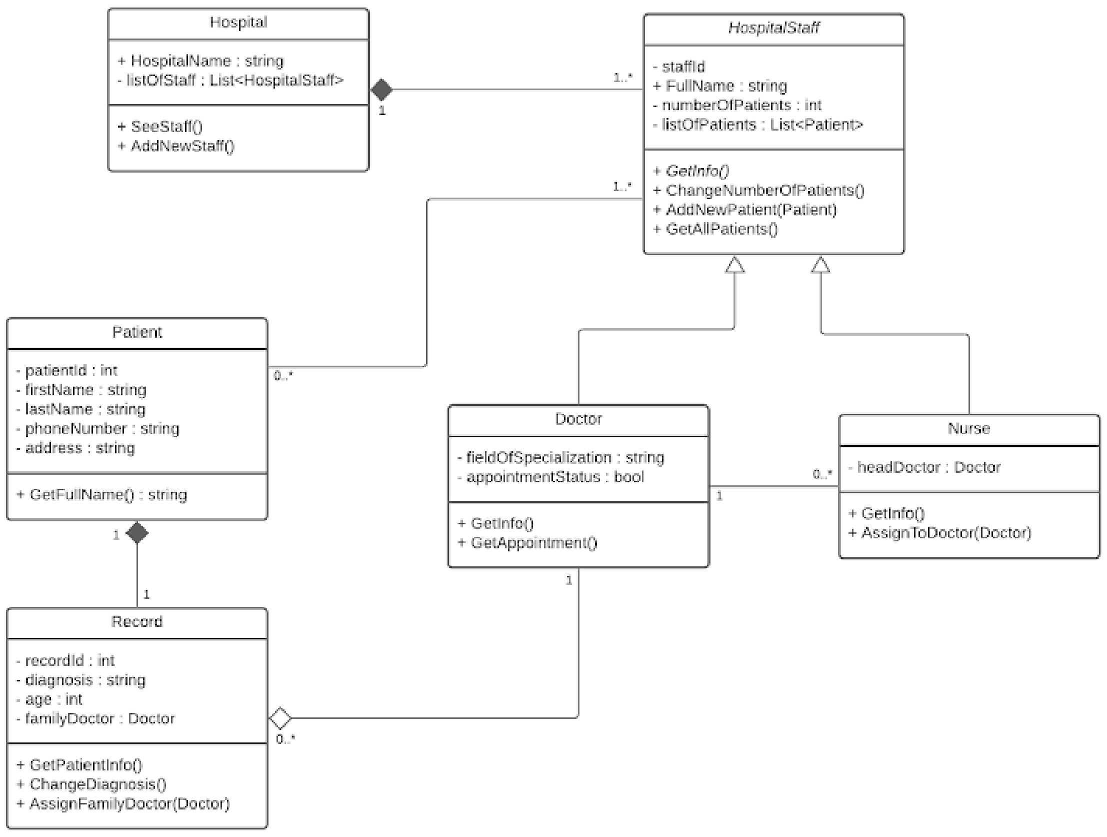
- **Hospitals Class** : Manage all the operations of Hospitals
- **Patient Class** : Manage all the operations of Patient
- **Doctors Class** : Manage all the operations of Doctors
- **Nurses Class** : Manage all the operations of Nurses
- **Appointments Class** : Manage all the operations of Appointments
- **Medicines Class** : Manage all the operations of Medicines

Classes and attributes of hospital management system class diagram

- **Hospitals Attributes** : hospital_id, hospital_doctor_id, hospital_name, hospital_place, hospital_type, hospital_description, hospital_address
- **Patient Attributes** : patient_id, patient_name, patient_mobile, patient_email, patient_username, patient_password, patient_address, patient_blood_group
- **Doctors Attributes** : doctor_id, doctor_name, doctor_specialist, doctor_mobile, doctor_email, doctor_username, doctor_password, doctor_address
- **Nurses Attributes** : nurse_id, nurse_name, nurse_duty_hour, nurse_mobile, nurse_email, nurse_username, nurse_password, nurse_address,
- **Appointments Attributes** : appointment_id, appointment_doctor_id, appointment_number, appointment_type, appointment_date, appointment_description
- **Medicines Attributes** : medicine_id, medicine_name, medicine_company, medicine_composition, medicine_cost, medicine_type, medicine_dose, medicine_description

Classes and attributes of hospital management system class diagram

- **Hospitals Methods** : addHospitals(), editHospitals(), deleteHospitals(), updateHospitals(), saveHospitals(), searchHospitals()
- **Patient Methods** : addPatient(), editPatient(), deletePatient(), updatePatient(), savePatient(), searchPatient()
- **Doctors Methods** : addDoctors(), editDoctors(), deleteDoctors(), updateDoctors(), saveDoctors(), searchDoctors()
- **Nurses Methods** : addNurses(), editNurses(), deleteNurses(), updateNurses(), saveNurses(), searchNurses()
- **Appointments Methods** : addAppointments(), editAppointments(), deleteAppointments(), updateAppointments(), saveAppointments(), searchAppointments()
- **Medicines Methods** : addMedicines(), editMedicines(), deleteMedicines(), updateMedicines(), saveMedicines(), searchMedicines()



8. Draw a class diagram for Library Management System. This diagram must describe the structure of the system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

Classes of Library Management System :

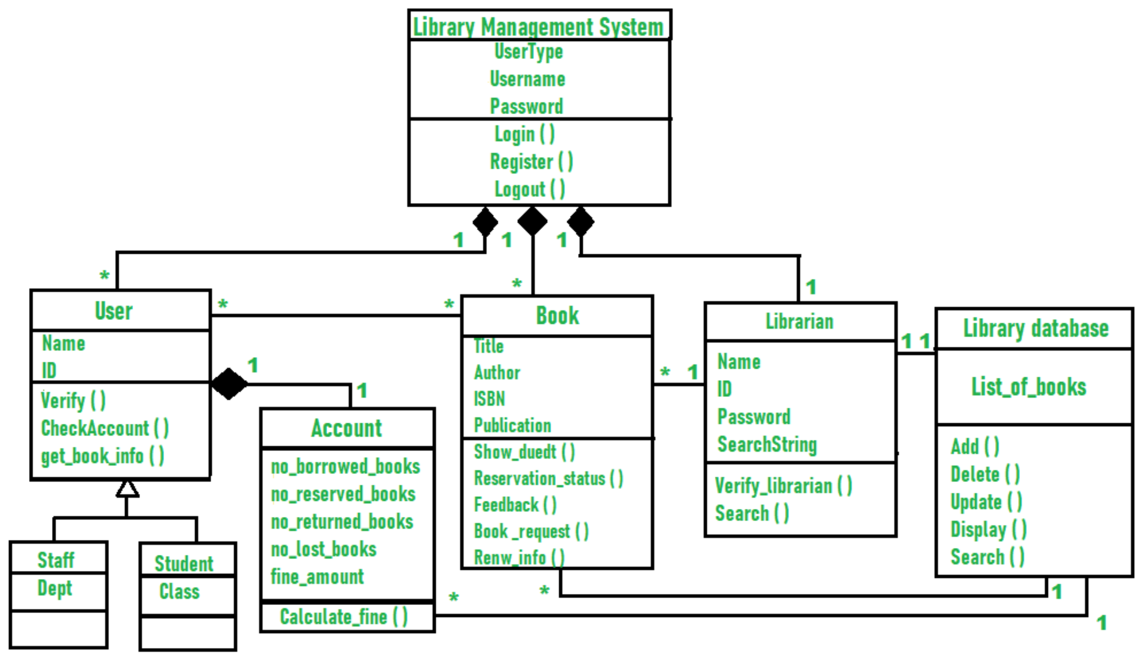
- **Library Management System class –**
  It manages all operations of Library Management System. It is central part of organization for which software is being designed.
- **User Class –**
  It manages all operations of user.
- **Librarian Class –** It manages all operations of Librarian.
- **Book Class –**
  It manages all operations of books. It is basic building block of system.
- **Account Class –**
  It manages all operations of account.
- **Library database Class –**
  It manages all operations of library database.
- **Staff Class –**
  It manages all operations of staff.
- **Student Class –**
  It manages all operations of student.

Attributes of Library Management System :

- **Library Management System Attributes –**
  UserType, Username, Password
- **User Attributes –**
  Name, Id
- **Librarian Attributes –**
  Name, Id, Password, SearchString
- **Book Attributes –**
  Title, Author, ISBN, Publication
- **Account Attributes –**
  no_borrowed_books, no_reserved_books, no_returned_books, no_lost_books fine_amount
- **Library database Attributes –**
  List_of_books
- **Staff Class Attributes –**
  Dept
- **Student Class Attributes –**
  Class

Methods of Library Management System :

- **Library Management System Methods –**
  Login(), Register(), Logout()
- **User Methods –**
  Verify(), CheckAccount(), get_book_info()
- **Librarian Methods –**
  Verify_librarian(), Search()
- **Book Methods –**
  Show_duedt(), Reservation_status(), Feedback(), Book_request(), Renew_info()
- **Account Methods –**
  Calculate_fine()
- **Library database Methods –**
  Add(), Delete(), Update(), Display(), Search()



9. What is the major use of data members and member functions as static? Explain it with an example.

The major use of static data members and member functions in a class is to associate properties or behaviors with the class itself rather than with individual instances of the class. Static members are shared among all instances of the class and belong to the class, not to any specific object.

Example: Utility Class with Static Methods

Let's consider a scenario where you need utility methods related to mathematical operations and you don't want to create instances of a class for using these methods. In such cases, static methods can be useful.

```
public class MathUtility {

   // Static method to calculate the square of a number

   public static double square(double number) {

      return number * number;

   }

   // Static method to calculate the cube of a number

   public static double cube(double number) {

      return number * number * number;
```

```
    }

    // Static method to check if a number is even

    public static boolean isEven(int number) {

        return number % 2 == 0;

    }

}
```

In this example:

- **square, cube,** and **isEven** are static methods that perform mathematical operations.
- These methods are not tied to any specific instance of the class and can be called directly on the class itself.

Now, let's use this utility class in a program:

```
public class Main {

    public static void main(String[] args) {

        double result1 = MathUtility.square(5.0);

        System.out.println("Square of 5: " + result1);

        double result2 = MathUtility.cube(3.0);

        System.out.println("Cube of 3: " + result2);

        int number = 8;

        boolean isEven = MathUtility.isEven(number);

        System.out.println(number + " is even: " + isEven);

    }

}
```

In this scenario, the **MathUtility** class acts as a utility class with static methods for performing common mathematical operations. You don't need to create an instance of the class; you can directly call the static methods on the class itself.

Benefits:

- No need to create instances: Static methods can be called without creating instances of the class, making them convenient for utility classes.
- Shared state: Static data members can be used for maintaining shared state or information across all instances of a class.
- Global accessibility: Static members can be accessed globally, making them suitable for operations that are not specific to any particular instance.

10. In object-oriented programming, what is the difference between accessing a class's public member and a private member?

In object-oriented programming, the difference between accessing a class's public member and a private member lies in the visibility and accessibility of these members from outside the class. Access modifiers, such as public and private, control the scope of class members. Here's a breakdown of the differences:

Public Member:

- **Visibility:**
  - Public members are visible and accessible from outside the class.
- **Access Scope:**
  - They can be accessed globally, both within and outside the package.

Example:

```
public class Example {
   public int publicField;
   public void publicMethod() {
      // implementation
   }
}
```

- **Accessing from Outside the Class:**

```
Example obj = new Example();

obj.publicField = 42;        // Accessing public field

obj.publicMethod();          // Calling public method
```

Private Member:

- **Visibility:**
  - Private members are not visible from outside the class.

- **Access Scope:**
    - They are limited to access within the class only.

<u>Example:</u>

```
public class Example {
    private int privateField;
    private void privateMethod() {
        // implementation
    }
}
```

- **Accessing from Outside the Class (Not Allowed):**

```
Example obj = new Example();
// obj.privateField = 42;    // Compilation error - private field not accessible
// obj.privateMethod();      // Compilation error - private method not accessible
```

Key Points:

- **Encapsulation:** The concept of encapsulation involves bundling the data (fields) and methods that operate on the data within a single unit (a class) and controlling access to that unit. Public and private access modifiers play a crucial role in encapsulation.
- **Information Hiding:** Private members are often used for information hiding, ensuring that the internal details of a class are not exposed to external code. This enhances security and allows the class to maintain control over its own state.
- **Public Interface:** Public members define the interface to the outside world, providing a way for external code to interact with and utilize the functionality of a class.

11. Explain object behaviors with examples.

In object-oriented programming, object behaviors refer to the actions or operations that objects can perform. Behaviors are typically defined by methods (functions) associated with an object's class. Let's explore this concept with examples:

<u>Example</u>: **Dog Class with Behaviors**

```
public class Dog {
    // Properties or state
```

```
        private String name;

        private int age;

        // Constructor

        public Dog(String name, int age) {

            this.name = name;

            this.age = age;

        }

        // Behavior: Bark

        public void bark() {

            System.out.println(name + " is barking!");

        }

        // Behavior: Fetch

        public void fetch() {

            System.out.println(name + " is fetching the ball.");

        }

        // Behavior: Sleep

        public void sleep() {

            System.out.println(name + " is sleeping.");

        }

        // Getter method for age

        public int getAge() {

            return age;

        }

    }
```

In this example:
The **Dog** class has properties (name and age) that represent the state of a dog.
It has behaviors (bark, fetch, sleep) that represent actions a dog can perform.

Key Points:

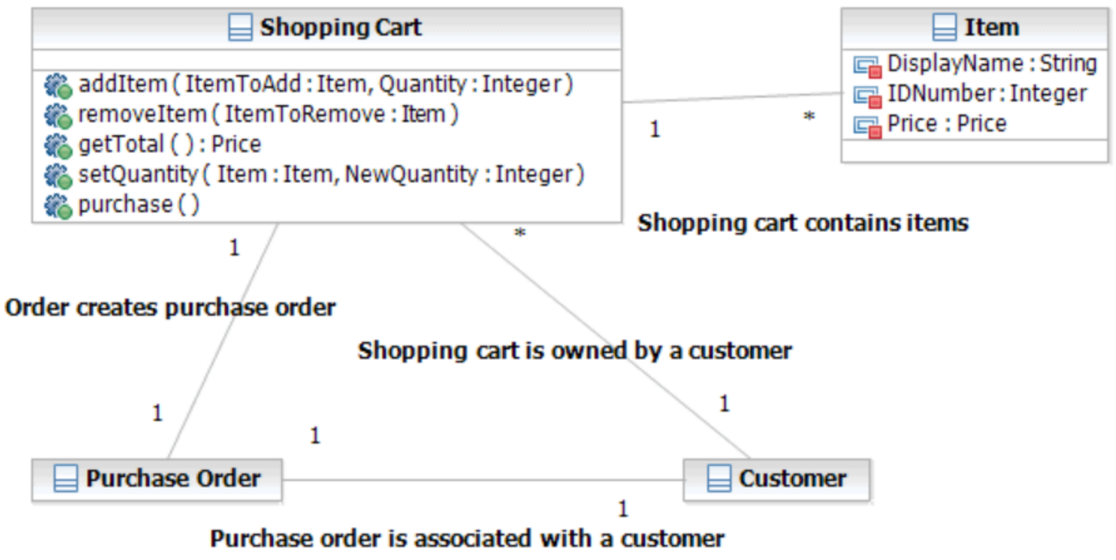
- Object behaviors are defined by methods in a class.
- These behaviors represent actions or operations that an object can perform.
- Methods provide an interface through which external code can interact with and instruct objects.

12. Explain the basic elements of a class diagram with an example.

In UML (Unified Modeling Language), class diagrams are one of six types of structural diagram. Class diagrams are fundamental to the object modeling process and model the static structure of a system. Depending on the complexity of a system, you can use a single class diagram to model an entire system, or you can use several class diagrams to model the components of a system.

Class diagrams are the blueprints of your system or subsystem. You can use class diagrams to model the objects that make up the system, to display the relationships between the objects, and to describe what those objects do and the services that they provide.

The following figure is an example of a simple class diagram. This diagram shows how a class that represents a shopping cart relates to classes that represent **customers, purchase orders, and items for sale.**


Shopping Cart
addItem (ItemToAdd : Item, Quantity : Integer)
removeItem (ItemToRemove : Item)
getTotal ( ) : Price
setQuantity (Item : Item, NewQuantity : Integer)
purchase ( )
Item
DisplayName : String
IDNumber : Integer
Price : Price
1
*
Shopping cart contains items
1
*
Order creates purchase order
Shopping cart is owned by a customer
1
1
1
Purchase Order
Customer
1
Purchase order is associated with a customer

Model elements in class diagrams

- **Classes**
  In UML, a *class* represents an object or a set of objects that share a common structure and behavior. Classes, or instances of classes, are common model elements in UML diagrams.

- **Objects**
  In UML models, *objects* are model elements that represent instances of a class or of classes. You can add objects to your model to represent concrete and prototypical instances. A concrete instance represents an actual person or thing in the real world. For example, a concrete instance of a Customer class represents an actual customer. A prototypical instance of a Customer class contains data that represents a typical customer.
- **Packages**
  Packages group related model elements of all types, including other packages.
- **Signals**
  In UML models, *signals* are model elements that are independent of the classifiers that handle them. Signals specify one-way, asynchronous communications between active objects.
- Enumerations
  In UML models, *enumerations* are model elements in class diagrams that represent user-defined data types. Enumerations contain sets of named identifiers that represent the values of the enumeration. These values are called enumeration literals.
- **Data** types
  In UML diagrams, *data types* are model elements that define data values. You typically use data types to represent primitive types, such as integer or string types, and enumerations, such as user-defined data types.
- **Artifacts**
  In UML models, *artifacts* are model elements that represent the physical entities in a software system. Artifacts represent physical implementation units, such as executable files, libraries, software components, documents, and databases.
- **Relationships in class diagrams**
  In UML, a relationship is a connection between model elements. A UML relationship is a type of model element that adds semantics to a model by defining the structure and behavior between model elements.
- **Qualifiers on association ends**
  In UML, *qualifiers* are properties of binary associations and are an optional part of association ends. A qualifier holds a list of association attributes, each with a name and a type. Association attributes model the keys that are used to index a subset of relationship instances.

13. Why static members are used in a class, and what advantages do they offer in terms of memory and performance?

Static members in a class are used to represent properties or behaviors that are shared among all instances of the class rather than being tied to a specific instance. They are associated with the class itself, not with individual objects created from the class. Static members include static variables (fields) and static methods.

Advantages of Using Static Members:

1. **Memory Efficiency:**
   - Static members are stored in a single memory location, regardless of the number of instances of the class.
   - Non-static members, on the other hand, are duplicated for each instance, leading to potential redundancy in memory usage.
2. **Global Accessibility:**
   - Static members can be accessed globally, both within and outside the class, without the need to create an instance of the class.
   - This allows for a convenient and centralized way to access shared resources or perform common operations.
3. **Shared State:**
   - Static variables allow the sharing of state or information among all instances of a class.
   - Changes to the static variable are reflected across all instances, providing a centralized way to maintain shared data.
4. **Performance Improvements:**
   - Static methods can be called directly on the class, without the need to create an instance. This can lead to a slight performance improvement since there is no need to instantiate an object.
   - Static methods are often used for utility functions or operations that don't depend on instance-specific state.
5. **Initialization Control:**
   - Static variables are initialized only once when the class is loaded into memory, providing control over when shared resources are initialized.
   - This can be useful for scenarios where you want to perform one-time initialization or set default values for shared resources.

14. Explain the difference between a static method and an instance method in terms of access and usage.

Static Method vs. Instance Method:

1. **Definition:**
   - **Static Method:**
     - A static method belongs to the class itself, not to any specific instance of the class.
     - It is declared using the static keyword.
     - Static methods can be called directly on the class without creating an instance.

- **Instance Method:**
    - An instance method is associated with an instance of the class.
    - It operates on the instance's state and is called on an object of the class.
    - Instance methods don't use the static keyword.

2. **Access:**
    - **Static Method:**
        - Can access only static members (variables or methods) of the class.
        - Cannot directly access instance-level members unless they are explicitly provided with an instance.
    - **Instance Method:**
        - Can access both static and instance members of the class.
        - Has access to the instance's state and other instance methods.
3. **Usage:**
    - **Static Method:**
        - Useful for operations that don't depend on instance-specific state.
        - Often used for utility methods or operations that apply to the class as a whole.
        - Called directly on the class (e.g., ClassName.staticMethod()).
    - **Instance Method:**
        - Operates on the instance's state and can access instance-specific data.
        - Used when the method needs to interact with the object's state.
        - Called on an instance of the class (e.g., object.instanceMethod()).
4. **Invocation:**
    - **Static Method:**
        - Invoked using the class name, not requiring an instance.
        - Example: ClassName.staticMethod().
    - **Instance Method:**
        - Invoked on an object of the class.
        - Example: object.instanceMethod().
5. **Memory Allocation:**
    - **Static Method:**
        - Exists in a single memory location and is shared among all instances of the class.
    - **Instance Method:**
        - Each instance has its own copy of instance methods, and memory is allocated per object.

15. In object-oriented programming, what is the significance of the static keyword when applied to members and methods?

In object-oriented programming, the static keyword is used to define members (variables and methods) that are associated with the class itself rather than with instances of the class. The significance of the static keyword lies in its impact on the behavior, accessibility, and memory allocation of the members to which it is applied.

1. Static Variables:

- **Significance:**
  - A static variable is shared among all instances of a class.
  - It exists in a single memory location, regardless of the number of instances.
  - Changes to a static variable are reflected across all instances.

Example:

```
public class Example {
   static int staticVariable = 0;
   int instanceVariable = 0;
}
```

staticVariable is shared among all instances, while each instance has its own copy of **instanceVariable.**

2. Static Methods:

- **Significance:**
  - A static method belongs to the class itself, not to any specific instance.
  - It can be called directly on the class without creating an instance.
  - Static methods cannot access instance-level members directly unless provided with an instance.

Example:

```
public class Example {
      static void staticMethod() {
      System.out.println("Static method called.");
   }

   void instanceMethod() {
      System.out.println("Instance method called.");
```

```
    }
  }
```

staticMethod() can be called as **Example.staticMethod()** without creating an instance.

3. Memory Allocation:

- **Significance:**
  - Static members are allocated memory once, shared among all instances.
  - Instance members are allocated memory for each instance separately.

  Example:

```
public class Example {
   static int staticVariable = 0;
   int instanceVariable = 0;
}
```

Memory for **staticVariable** is shared among all instances, while each instance gets its own memory for **instanceVariable.**

4. Global Accessibility:

- **Significance:**
  - Static members can be accessed globally without creating instances.
  - Useful for providing a centralized way to access shared resources or perform common operations

  Example:

```
public class Utility {
   public static void performTask() {
      // Task implementation
   }
}
```

Utility.performTask() can be called from anywhere in the code.

16. Give an example of a real-world problem where static members and methods are useful for solving the problem efficiently.

Real-World Example: Configuration Management

Consider a scenario where an application needs to manage configuration settings that are common across the entire application. Static members and methods can be employed to create a simple configuration manager that allows easy access to configuration settings without the need to create multiple instances.

```
public class ConfigurationManager {
   // Static variables for configuration settings
   private static String databaseUrl;
   private static String username;
   private static String password;

   // Static method to initialize configuration settings
   public static void initializeConfiguration(String dbUrl, String user,
     String pass) {
      databaseUrl = dbUrl;
      username = user;
      password = pass;
   }

   // Static method to get the database URL
   public static String getDatabaseUrl() {
      return databaseUrl;
   }

   // Static method to get the username
   public static String getUsername() {
      return username;
   }

   // Static method to get the password
   public static String getPassword() {
      return password;
   }
}
```

In this example:

- Static variables (**databaseUrl, username, and password**) hold configuration settings.
- The **initializeConfiguration** method is used to set the configuration settings.
- Static methods (**getDatabaseUrl, getUsername, and getPassword**) provide access to the configuration settings.

Benefits:

- Static members allow a single instance of the configuration manager to be shared across the entire application.
- Static methods provide a global and convenient way to access configuration settings from any part of the code.
- The static configuration manager is initialized only once, reducing redundancy and improving memory efficiency.
- The global accessibility of static methods ensures a consistent and centralized approach to configuration management.

This example illustrates how static members and methods can efficiently handle the management of shared configuration settings in a real-world scenario.

17. What are some potential drawbacks or limitations of using static members and methods, and when should you be cautious about their use?

While static members and methods offer several benefits, there are also potential drawbacks and limitations associated with their use. It's essential to be cautious and consider these factors:

1. **Global State:**
    - **Drawback**: Static members introduce a global state that can be accessed and modified from anywhere in the code. This can lead to unintended interactions and make it challenging to reason about the state of an application.
    - **Caution:** Avoid excessive use of mutable static variables that represent global state. Prefer immutability or encapsulation where possible.
2. **Testing Challenges:**
    - **Drawback:** Code with heavy use of static members can be challenging to test in isolation. Static state may persist between test cases, leading to dependencies and affecting test outcomes.
    - **Caution:** When designing for testability, consider dependency injection or other patterns that allow for more straightforward testing of individual components.
3. **Thread Safety:**
    - **Drawback:** Static members can lead to thread safety issues in a multi-threaded environment. If shared mutable state is not properly synchronized, it may result in race conditions and data inconsistencies.
    - **Caution**: Use synchronization mechanisms or consider alternatives like thread-local variables when dealing with mutable static state in a multi-threaded context.

4. **Hard to Mock:**
   - **Drawback:** Static methods are often challenging to mock or replace during unit testing, making it difficult to isolate and test components independently.
   - **Caution:** When dependency injection or inversion of control is not an option, consider using frameworks or techniques that enable the mocking of static methods.
5. **Inflexibility:**
   - **Drawback:** Static members are less flexible than instance members, as they are associated with the class itself. This can limit certain design patterns and make it harder to adapt to changes.
   - **Caution:** When designing for flexibility and extensibility, prefer instance methods and variables, especially when dealing with polymorphism and inheritance.
6. **Dependency on Order of Execution:**
   - **Drawback:** Static blocks and initialization of static variables depend on the order in which classes are loaded. This can lead to subtle bugs and issues in cases where the order of execution is not well-understood.
   - **Caution:** Avoid complex dependencies between static blocks or variables. Ensure that the order of execution is clear and well-defined.
7. **Global Accessibility:**
   - **Drawback:** The global accessibility of static members can lead to tight coupling between different parts of the code, making it harder to maintain and refactor the codebase.
   - **Caution:** Use static members judiciously and consider encapsulation and modularization to reduce dependencies between different components.

18. What is memory allocation, and why is it important in computer systems?

Memory Allocation:

Memory allocation is the process of assigning portions of a computer's memory to specific programs or processes. In a computer system, memory refers to the storage space used by the operating system and applications to store data and execute instructions. Memory allocation is crucial for managing the limited physical and virtual memory resources efficiently.

Importance of Memory Allocation in Computer Systems:

1. Resource Management:
   - Efficient Utilization: Memory allocation ensures that each program or process receives the necessary memory resources to execute its tasks efficiently. It helps in utilizing the available memory effectively.

2. **Program Execution:**
    - Storage for Instructions and Data: Programs need memory to store their executable instructions and data. Memory allocation ensures that the instructions and data required for program execution are available in a designated memory space.
3. **Dynamic Memory Usage:**
    - Dynamic Data Structures: Many programs use dynamic data structures like arrays, linked lists, and trees. Memory allocation allows these structures to expand or shrink as needed during program execution.
4. **Prevention of Conflicts:**
    - Isolation of Processes: Memory allocation prevents conflicts between different processes or programs running concurrently. Each process is allocated its own memory space, preventing interference with the memory of other processes.
5. **Security and Protection:**
    - Isolation of Memory: Memory allocation contributes to the security of a system by isolating the memory spaces of different processes. This prevents unauthorized access and protects the integrity of data.
6. **Optimization of Performance:**
    - Caching and Speed: Proper memory allocation strategies contribute to optimizing the performance of a computer system. Well-allocated memory facilitates efficient caching and helps in speeding up data access.
7. **Virtual Memory Management:**
    - Swapping and Paging: Memory allocation is crucial in the context of virtual memory management. Techniques like swapping and paging involve moving data between physical and virtual memory spaces to ensure smooth operation when physical memory is limited.
8. **Error Handling:**
    - Out-of-Memory Handling: Memory allocation allows for proper error handling when a program or system encounters out-of-memory situations. Systems can implement mechanisms to handle such errors gracefully.
9. **Memory Leak Prevention:**
    - Deallocation of Unused Memory: Proper memory allocation includes deallocating memory that is no longer needed. This prevents memory leaks, where memory is allocated but not released, leading to inefficient use of resources.
10. **Adaptability to System Changes:**
    - Dynamic Environment: Memory allocation allows systems to adapt to dynamic changes, such as the addition or removal of programs. It provides the flexibility to allocate and deallocate memory as needed.

19. Describe the concept of dynamic memory allocation and its advantages in program flexibility.

Dynamic memory allocation refers to the process of allocating memory during the execution of a program, allowing the program to request and release memory as needed. In contrast to static memory allocation, where memory is allocated at compile-time and has a fixed size, dynamic memory allocation enables programs to adapt to varying data requirements at runtime.

Advantages of Dynamic Memory Allocation:

1. **Adaptability and Flexibility:**
    - **Variable Memory Needs**: Dynamic memory allocation allows programs to adapt to varying memory requirements during execution. It enables the creation and resizing of data structures based on actual runtime needs.
2. **Dynamic Data Structures:**
    - **Flexible Structures:** Dynamic memory allocation is essential for implementing dynamic data structures like linked lists, trees, and resizable arrays. These structures can grow or shrink as needed without the need for a predefined size.
3. **Efficient Resource Usage:**
    - **Optimal Memory Utilization:** Dynamic memory allocation facilitates optimal utilization of memory resources. Memory can be allocated only when needed, reducing wastage and improving efficiency.
4. **Run-time Flexibility:**
    - On-the-Fly Changes: Dynamic memory allocation allows for on-the-fly changes to memory requirements. Programs can allocate or deallocate memory based on user inputs, data processing needs, or other dynamic factors.
5. **Avoidance of Fixed Size Limitations:**
    - No Fixed Size Constraints: Unlike static memory allocation, dynamic allocation avoids fixed size constraints. Programs are not limited by a predetermined amount of memory, and they can acquire more as required.
6. **Memory Leak Prevention:**
    - Deallocation Capability: Dynamic memory allocation enables the deallocation of memory that is no longer needed. Proper deallocation helps prevent memory leaks, ensuring efficient use of resources.
7. **Run-time Data Loading:**
    - Loading Data Dynamically: Dynamic memory allocation allows programs to load data into memory at runtime. This is particularly useful when dealing with data sets of unknown or variable size.

8. **Efficient String Handling:**
    - String Manipulation: Dynamic memory allocation is commonly used in string manipulation, allowing strings to be created, concatenated, and resized without limitations imposed by fixed-size buffers.
9. **Increased Program Modularity:**
    - Modular and Scalable Code: Dynamic memory allocation promotes modularity and scalability in code. It allows developers to design programs with a focus on functionality rather than being constrained by fixed memory limits.
10. **Support for Recursive Data Structures:**
    - Recursion and Pointers: Dynamic memory allocation supports the creation of recursive data structures, where structures contain references to other instances of the same structure, forming a dynamic and interconnected hierarchy.

20. What is virtual memory, and how does it extend physical memory allocation to improve system performance?

Virtual Memory:

Virtual memory is a memory management technique used by operating systems to provide an illusion to the running processes that each one has its own dedicated piece of contiguous address space, which may exceed the actual physical RAM available in the system. It allows programs to use more memory than is physically available by using a combination of RAM and disk space.

In a system with virtual memory, each process has its own virtual address space, starting from address 0 and going up to the maximum address supported by the architecture. This address space is divided into sections, including the code section, data section, and stack.

How Virtual Memory Works:

1. **Address Translation:**
    - The operating system uses a data structure called a page table to map virtual addresses to physical addresses.
    - When a process accesses a location in its virtual memory, the operating system translates the virtual address to the corresponding physical address using the page table.
2. **Page Faults:**
    - Not all of the virtual memory is initially loaded into physical RAM. Only the portions of the program that are actively being used are loaded.

    - If a process attempts to access a part of its virtual memory that is not in RAM (a page fault), the operating system loads the required portion from the disk into RAM.
3. **Swapping:**
    - When physical RAM is full, the operating system may move pages that are not actively in use to a special area on the disk called the swap space.
    - This process is known as swapping, and it allows the system to free up space in RAM for pages that are actively used.
4. **Demand Paging:**
    - The concept of demand paging is used, where only the pages that are needed are brought into memory.
    - This reduces the amount of time and resources required to load a program into memory initially.

Advantages of Virtual Memory:

1. **Program Size Exceeds RAM:**
    - Virtual memory allows programs to be larger than the physical RAM available in the system. This is particularly important for running large applications or multiple applications concurrently.
2. **Isolation of Processes:**
    - Each process has its own virtual address space, ensuring isolation from other processes. A process cannot directly access the memory of another process.
3. **Ease of Memory Management:**
    - Virtual memory simplifies memory management for both the operating system and application developers. Developers can work with a larger, contiguous address space, and the operating system can manage the physical-to-virtual address mappings.
4. **Improved Multi-tasking:**
    - Virtual memory supports efficient multitasking by allowing multiple processes to run concurrently, even if the total memory requirements of these processes exceed physical RAM.
5. **Reduced Fragmentation:**
    - Virtual memory reduces external fragmentation because the operating system can move pages around in the physical memory or swap them in and out of disk storage.
6. **Flexible Memory Allocation:**
    - It enables flexible memory allocation by allowing the operating system to allocate and deallocate memory dynamically based on the needs of running processes.

Improving System Performance:

1. **Optimal Resource Utilization:**
    - Virtual memory ensures optimal utilization of physical RAM and disk space by dynamically swapping pages in and out as needed.
2. **Efficient Use of RAM:**
    - Processes that are not actively using their entire address space can share physical RAM efficiently. Only the actively used portions need to be loaded.
3. **Effective Multi-tasking:**
    - Virtual memory enables effective multitasking by allowing multiple processes to run concurrently, each with its own isolated address space.
4. **Improved Responsiveness:**
    - Processes can be paged in and out as needed, allowing the operating system to respond to varying workloads and priorities efficiently.

PART C- SHORT ANSWER QUESTIONS

1. **What is a class?**

In object-oriented programming, a class is a template definition of the methods and variables in a particular kind of object. Thus, an object is a specific instance of a class; it contains real values instead of variables.

2. **What is an object?**

An object is a single instance of class, which contains data and methods working on that data. So, an object consists of three things:

- Name: This is the variable name that represents the object.
- Member data: The data that describes the object.
- Member methods: Behaviour that describes the object.

3. **What is an interface?**

An interface is a programming structure/syntax that allows the computer to enforce certain properties on an object (class).

4. **What is method hiding?**

Method hiding can be defined as, "if a subclass defines a static method with the same signature as a static method in the super class, in such a case, the method in the subclass hides the one in the superclass."

5. **How many types of access specifiers are provided in OOP?**

In object-oriented programming (OOP), there are typically **three main types** and **one additional type of access specifiers** that control the visibility and accessibility of class members (fields, methods, inner classes) within a class hierarchy.

1.Public, 2.Private, 3.protected and 4.Default (package-private) access specifier.

6. **What is an instance of a class?**

In object-oriented programming (OOP), an instance of a class refers to a specific occurrence or object created based on the blueprint defined by that class.

7. **Define class diagram.**

The class diagram depicts a static view of an application. It represents the types of objects residing in the system and the relationships between them. A class consists of its objects, and also it may inherit from other classes. A class diagram is used to visualize, describe, document various different aspects of the system, and also construct executable software code.

8. **List out the vital components of a class diagram.**

A class diagram is used to visualize, describe, document various different aspects of the system, and also construct executable software code. It shows the attributes, classes, functions, and relationships to give an overview of the software system.

- **Upper Section:** The upper section encompasses the name of the class. A class is a representation of similar objects that shares the same relationships, attributes, operations, and semantics. Some of the following rules that should be taken into account while representing a class are given below:
    a. Capitalize the initial letter of the class name.
    b. Place the class name in the center of the upper section.
    c. A class name must be written in bold format.
    d. The name of the abstract class should be written in italics format.

- **Middle Section:** The middle section constitutes the attributes, which describe the quality of the class. The attributes have the following characteristics:
    a. The attributes are written along with its visibility factors, which are public (+), private (-), protected (#), and package (~).
    b. The accessibility of an attribute class is illustrated by the visibility factors.
    c. A meaningful name should be assigned to the attribute, which will explain its usage inside the class.
- **Lower Section:** The lower section contain methods or operations. The methods are represented in the form of a list, where each method is written in a single line. It demonstrates how a class interacts with data.

9. What is the purpose of a class diagram?

- It analyses and designs a static view of an application.
- It describes the major responsibilities of a system.
- It is a base for component and deployment diagrams.
- It incorporates forward and reverse engineering.

10. What are the benefits of class diagrams?

- It can represent the object model for complex systems.
- It reduces the maintenance time by providing an overview of how an application is structured before coding.
- It provides a general schematic of an application for better understanding.
- It represents a detailed chart by highlighting the desired code, which is to be programmed.
- It is helpful for the stakeholders and the developers.

11. Define public access specifier.

Public means all class members declared public might be available to everyone. The data members and member functions said other classes could access the public. Hence, public members can be called from any part of the program.

(OR)

- **Public:** When defined in public, then it is accessible anywhere in the complete program. The programmer can call the member from anywhere with the help of an object of the class. In simple words, you can say it is globally accessible in the program.

12. Define private access specifier.

- **Private:** When the class members are defined under a private specifier, then it is accessible only inside the class. The programmer is not allowed to use outside the class.

13. Define protected access specifier.

- **Protected**: The protected member of the parent class is accessible only in the child classes.

14. What is the difference between default and protected access specifier?

Default: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

Protected: The access level of a protected modifier is within the package and outside the package through child class.

15. Define aggregation.

Aggregation is a specialized form of association between two or more objects in which each object has its own life cycle but there exists an ownership as well.

16. Define generalization.

Generalization is the process of taking out common properties and functionalities from two or more classes and combining them together into another class which acts as the parent class of those classes.

17. Define association.

Association defines a relationship between two separate classes, which is established with the help of objects.

18. What is a static member?

Static members are data members (variables) or methods that belong to a static or a non static class itself, rather than to objects of the class. Static members always remain the same, regardless of where and how they are used.

19. What is a static method?

A static method in object-oriented programming (OOP) is a method that belongs to a class rather than to instances of that class. Unlike instance methods, which operate on an instance of the class and can access instance-specific data, static methods are associated with the class itself and can only access static (class-level) members.

20. What is the need for static members.

A typical use of static members is for recording data common to all objects of a class. For example, if we use a static data member as a counter to store the number of objects of a particular class type that are created.