|  |  |
| --- | --- |
| Week2 | Java Fundamentals-II -- OOPS concepts ,non-access modifiers  inner classes, generics, wrapper classes |

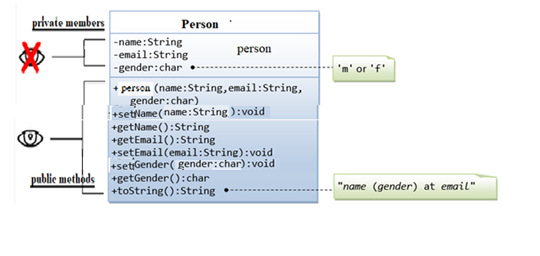
1. Encapsulation
2. Abstraction
3. Polymorphism
4. Inheritance

**Encapsulation**

**To achieve encapsulation in Java**−

* Declare the variables of a class as private.
* Provide public setter and getter methods to modify and view the variables values.

Encapsulation gives maintainability, flexibility and extensibility to our code.



**NOTE:- Private data members are acces through public member functions.**

The Person class has three fields – name , email & gender. All are declared as private, meaning they can not be accessed directly outside the class.

The getter and setter methods : getName, setName,…ect. are declared public. Those methods are exposed to “outsiders” and can be used to change and retrieve data from the Person object.

**toString() method** makes it easier to output debugging traces, or makes better logging messages, since you can use the object's string representation provided by toString() directly; you don't have to manually build a string that gives the information needed on the object.

**Abstraction**

**Abstraction** is a process of hiding the complexity from end the user.

## **abstract method,** method with no implementation

## **concrete method,** method with implementation

**concrete class** is a class that has an implementation for all of its methods

**abstarct class** is a class that has one abstract method

There are two ways to achieve abstraction in java

|  |  |
| --- | --- |
| **1.Abstract class (0 to 100%)** | **2.Interface (100%)** |

## 

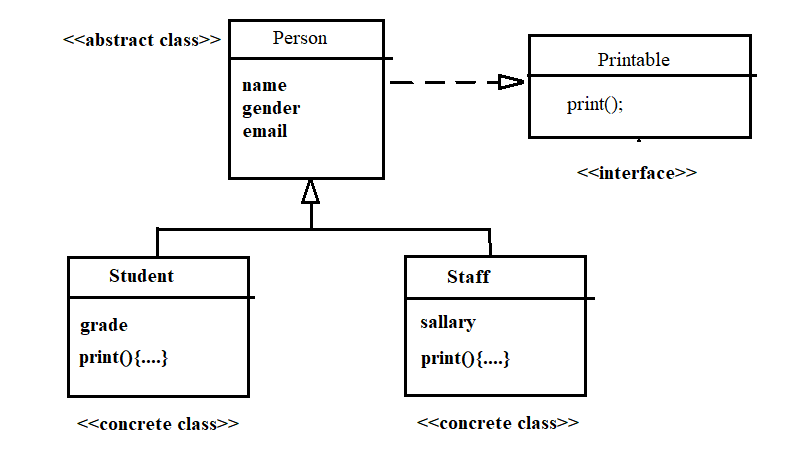
|  |  |
| --- | --- |
| It can have constructor and static methods also.  It can have final methods | The interface keyword is used to declare an interface.  Example  interface Person  {  //Data members will be made **public static final**  //Member functions will be made **public abstract**.  } |

## 

## 

## **PROBLEM STATEMENT:-**

**Build java program for the following class structure.**



When to use abstract:--

If you want to provide some **common or default functionality** then the abstract class would be a better choice.

When to use interface:--

If you want to **enforce some rule or want to make a contract** of implementation the interface would be a better choice. **To support multiple inheritance**

**NOTE**:- We can use the abstract class and interface together. An interface can provide only a contract but by use of an abstract class, we can provide skeletal of that contract.

**INTERVIEW QUESTION**

**What is marker interface?**

A marker interface is an interface that doesn't have any methods or constants inside it. It provides run-time type information about objects, so the compiler and JVM have additional information about the object. A marker interface is also called a tagging interface.

**INTERVIEW QUESTION**

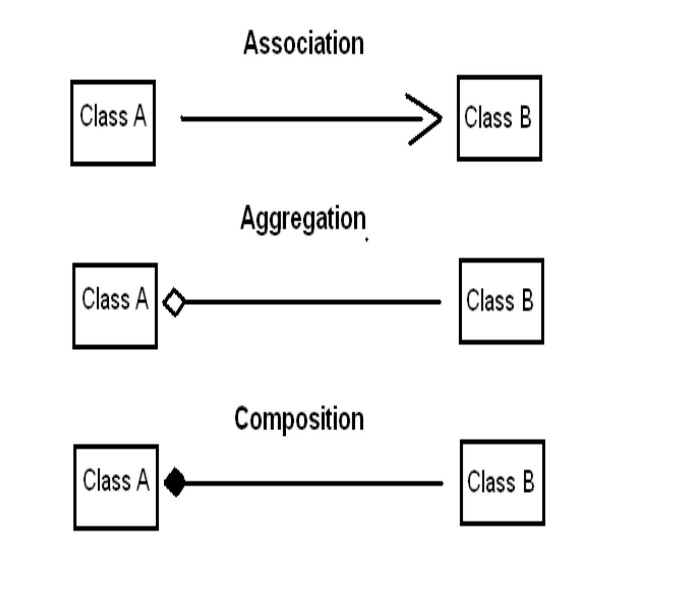
**INTERVIEW QUESTION**

**Super keyword vs super()?**

The super keyword in Java is a reference variable that is used to refer parent class objects. The super() in Java is a reference variable that is used to refer parent class constructors. super can be used to call parent class' variables and methods. super() can be used to call parent class' constructors only.

**Association**, is the relationship between two classes

|  |  |  |  |
| --- | --- | --- | --- |
| **one-to-one**  **A person can have only one passport.** | **one-to-many**  **A person can have many mobile phones.** | **many-to-one**  **Many Books in one Library** | **many-to-many**  **A student can have multiple teachers and a teacher can have multiple students.** |



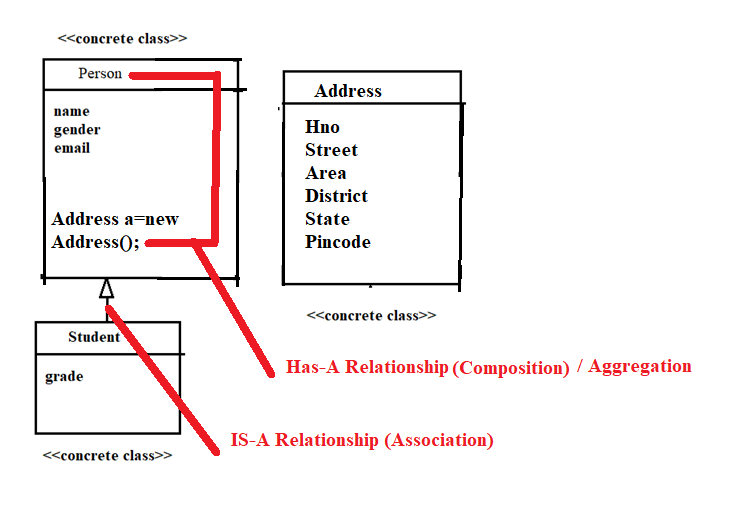
**NOTE:-**When there is an aggregation between two entities, the aggregate object can exist without the other entity, but in the case of Composition, the composed object can't exist.

**INTERVIEW QUESTION**

**Association(is-a) vs Aggregation(has-a) vs Composition(has-a)**

## **PROBLEM STATEMENT:-**

**Build java program for the following class structure.**

****

### **Why use Aggregation?**

* For Code Reusability.

### **Why use Association?**

* [runtime polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java) can be achieved).
* For Code Reusability.

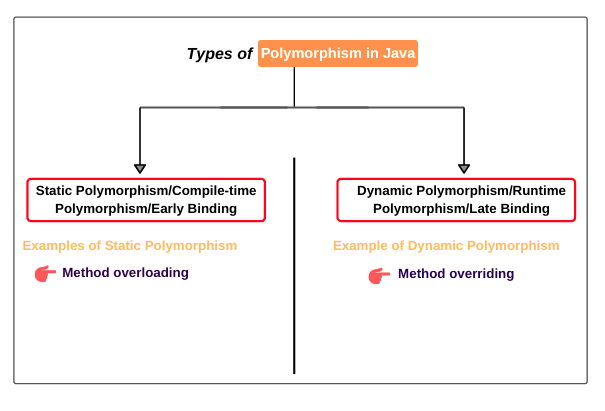
NOTE:-we cannot use **this()** and **super()** keywords in the same block.

**INTERVIEW QUESTION**

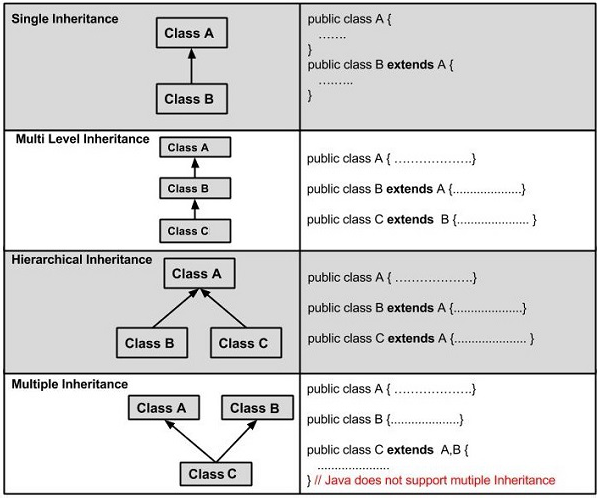
**super vs super()**

The super keyword in Java is a reference variable that is used to refer parent class objects. The super() in Java is a reference variable that is used to refer parent class constructors. super can be used to call parent class' variables and methods. super() can be used to call parent class' constructors only.

**Polymorphism**

****

**Inheritance**

****



**Aggregation vs composition**

The**super keyword in Java** refer parent class data member or constructor or method

## **Super()**

Whenever a child class constructor gets invoked it implicitly invokes the constructor of parent class.

**Non-access Specifiers**

Static

Final

The **static keyword** in [Java](https://www.javatpoint.com/java-tutorial) is used for memory management

The static can be:

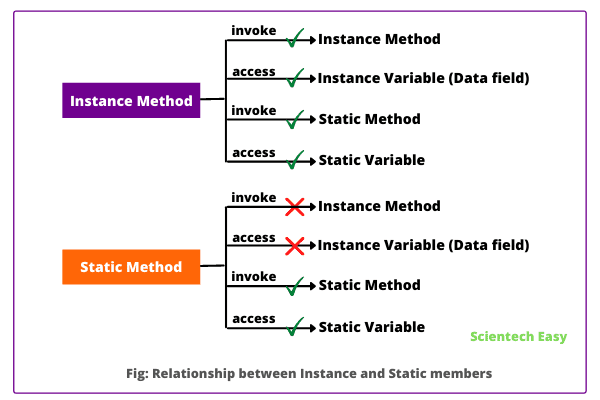
1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class

There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

|  |  |
| --- | --- |
| Static variable  Class variable, shared among all objects  Static member can be accessed without reference to any object.  Static variables cannot access non-static members of a class, | Staic block   * Is used to initialize the static data member. |
| Static method  Any static member can be accessed before any objects of its class are created, and without reference to any object  Static methods cannot access non-static members of a class.  Static methods can be overloaded, **but not overridden**   * They cannot refer to [this](https://www.geeksforgeeks.org/this-reference-in-java/) or [super](https://www.geeksforgeeks.org/super-keyword/) in any way. | Static class  A class can be made **static** only if it is a nested class.  Nested static class doesn’t need a reference of Outer class. In this case, a static class cannot access non-static members of the Outer class. |

* static block and static variables are executed in the order they are present in a program.



**Guess the O/P**

**class Test**

**{**

**// static variable**

**static int a = 10;**

**public static void main(String[] args)**

**{**

**int a = 20;**

**// Cannot use super in a static context**

**System.out.println(super.a); // compiler error**

**}**

**}**

**WAP to create student records getting unique rollNo through the use of static variable**

Can a non-satic method acces static data member?

NO

**Final**

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class 

A final variable that have no value it is called blank final variable or uninitialized final variable.

It can be initialized in the constructor only.

The blank final variable can be static also which will be initialized in the static block only.

## **Java final variable**

If you make any variable as final, you cannot change the value of final variable(It will be constant).

**INTERVIEW QUESTION**

### **Is final method inherited?**

Ans) Yes, final method is inherited but you cannot override it.

### **What is blank or uninitialized final variable?**

### **Can we declare a constructor final?**

No, because constructor is never inherited.

**Inner Classes**

**Java inner class** or nested class is a class that is declared inside the class or interface.

use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.

**NOTE:-**An instance of an inner class cannot be created without an instance of the outer class

**Note** that inner classes can access outer class private members and at the same time we can hide inner class from outer world.

Java inner class is defined inside the body of another class. Java inner class can be declared private, public, protected, or with default access whereas an outer class can have only public or default access.



Types of inner classes

* Non-static nested class (inner class)
  1. Member inner class
  2. Anonymous inner class
  3. Local inner class
* Static nested class

|  |  |
| --- | --- |
| Inner class example | Static nested class example |

## Realtime Example of Inner class

1. Suppose there is a university. University contains several departments such as electronics, computer science, electrical, mechanical, etc.

Assume that tomorrow, the government decided to close this university due to its illegal activity happening in the university. If the university is closed, all departments of this university will be closed automatically.

That’s why we can say that departments are tied to its university. i.e. their functionalities are tied with its university. So, without an existing university object, there is no chance of existing department objects.

Since the department is always a part of the university class. Hence, we must declare the department class inside the university class.

class University {  // University is outer class.

  . . . . . . .

 class Department { // Department is inner class.

    . . . . . . . .

  }

}

Scenario2:--

Let’s understand it with the help of realtime example program.

Assume that we create a class BankAcct. To get some bank account details, we define instance variables ‘balance’ and ‘rate’ of interest in the class.

calInterest() method calculates the interest amount and will display balance amount. Look at the following source code.

class BankAcct {

// Balance and Rate of interest.

private double bal;

private double rate;

// Calculating interest and updated balance.

void calInterest() {

double interest = bal \* rate/100;

bal += interest;

System.out.println("Balance amount = " +bal);

}

}

As you observe that there is no security for this code because of not using any access modifiers before BankAcct class.

Since it is under default access modifiers, any programmer can easily create an object of this class and access the members of the class from outside other classes. Hence, there is no security for BankAcct class.

For example, any programmer can create another class. He will create an instance of BankAcct class and can change rate of interest like this:

class Myclass {

public static void main(String[ ] args ){

BankAcct ba = new BankAcct();

ba.calInterest(11.5);

}

}

Thus, calInterest() method is very sensitive. It must be protected from outsiders because the balance accounts can be updated from outside by any other person. Only authorized person should be able to update the balance accounts.

To provide security to calInterest() method and rate of interest, if we put code of calInterest() method inside an inner class and make it private, nobody can access them from outside class. Look at the code.

class BankAcct {

. . . . . . . .

private class Interest {

private double rate;

void calInterest() {

double interest = bal \* rate/100;

bal += interest;

System.out.println("Balance = " +bal);

}

}}

Now the question is how to access the inner class?

To access the inner class, we will create a contact() method inside the outer class. Inside contact() method, we will create an object of inner class and can access members of it.

An authentication bank person can interact with inner class by calling contact() method. When contact() method will be called, it will verify with password and then he will be able to use the inner class and access it.

Thus, an inner class can provide a safety mechanism for important code in Java.

## Types of Inner class in Java

* Normal or Regular Inner Class, A class created inside another class and outside the method without static modifier
* Method Local Inner Class,  class declared within a method of the outer class
* Anonymous Inner Class,  class declared without any name
* Static Nested Inner Class, A class declared with static modifier within a class

**Note:** The regular, method-local, and anonymous inner classes are non-static nested classes.

**Generics**

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects.

Type safety

Compile time checking

Type casting not required

Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

With Generics, it is required to specify the type of object we need to store.

Before Generics

List list = new ArrayList();

list.add(10);

list.add("10");

String s = (String) list.get(1);//typecasting

With Generics, it is required to specify the type of object we need to store.

List<Integer> list = new ArrayList<Integer>();

list.add(10);

list.add("10");// compile-time error

List<String> list = new ArrayList<String>();

list.add("hello");

String s = list.get(0); //typecasting notrequired

## **Type Parameters**

The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:

1. T - Type
2. E - Element
3. K - Key
4. N - Number
5. V - Value

|  |  |
| --- | --- |
| **Generic class** A class that can refer to any type is known as a generic class. Here, we are using the T type parameter to create the generic class of specific type.  List<String> list = new ArrayList<String>();  list.add("hello");  String s = list.get(0); //typecasting notrequired | **Generic Method** A method that can accept any type of arguments is known as generic method. Here, we are using the E type parameter to denote the element of specific type as argument in generic method.  public class Test{    public static < E > void printArray(E[] elements) {  for ( E element : elements){  System.out.println(element );  }  System.out.println();  }  public static void main( String args[] ) {  Integer[] intArray = { 10, 20, 30, 40, 50 };  Character[] charArray = { 'J', 'A', 'V', 'A', 'T','P','O','I','N','T' };    System.out.println( "Printing Integer Array" );  printArray( intArray );    System.out.println( "Printing Character Array" );  printArray( charArray );  }  } |

## **Wildcard in Java Generics**

The ? (question mark) symbol represents the wildcard element. It means any type.

|  |  |
| --- | --- |
| UpperBoundWildcard  <? extends Number>   1. **import** java.util.ArrayList; 3. **public** **class** UpperBoundWildcard {  6. **private** **static** Double add(ArrayList<? **extends** Number> num) { 8. **double** sum=0.0; 10. **for**(Number n:num) 11. { 12. sum = sum+n.doubleValue(); 13. } 15. **return** sum; 16. } 18. **public** **static** **void** main(String[] args) { 20. ArrayList<Integer> l1=**new** ArrayList<Integer>(); 21. l1.add(10); 22. l1.add(20); 23. System.out.println("displaying the sum= "+add(l1)); 25. ArrayList<Double> l2=**new** ArrayList<Double>(); 26. l2.add(30.0); 27. l2.add(40.0); 28. System.out.println("displaying the sum= "+add(l2));  31. } 33. } | UnboundedWildcard  <?>   1. **import** java.util.Arrays; 2. **import** java.util.List; 4. **public** **class** UnboundedWildcard { 6. **public** **static** **void** display(List<?> list) 7. { 9. **for**(Object o:list) 10. { 11. System.out.println(o); 12. } 14. }  17. **public** **static** **void** main(String[] args) { 19. List<Integer> l1=Arrays.asList(1,2,3); 20. System.out.println("displaying the Integer values"); 21. display(l1); 22. List<String> l2=Arrays.asList("One","Two","Three"); 23. System.out.println("displaying the String values"); 24. display(l2); 25. } 27. } |

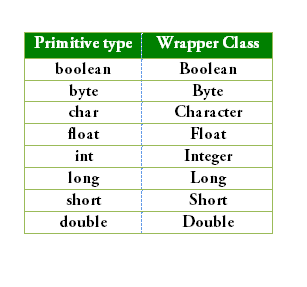
**Wrapper Classes:-**

The **wrapper class in Java** provides the mechanism to convert primitive into object and object into primitive.

Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

**Interview Question**

**Autoboxing**



since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

//Converting int into Integer

int a=20;

Integer i=Integer.valueOf(a);//converting int into Integer explicitly

Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally

//Converting Integer to int

Integer a=new Integer(3);

int i=a.intValue();//converting Integer to int explicitly

int j=a;//unboxing, now compiler will write a.intValue() internally