**Java OOPS Concept**

Q.Difference between Abstraction and Encapslation

Abstraction is a process of creating an instance(object) of a class and encapsulation is a process of providing a way for user to access and modify the instance value(s) while making sure unwanted access or modification can be avoided.

**What is encapsulation?**

Encapsulation is a principle based on hiding the state of objects and restricting their access to various parts of your program. Encapsulation is achieved with the use of private access modifiers and the getter and setter methods.

**How do you create a fully encapsulated class?**

1. Make all the instance variables of a class private.
2. Only use getter and setter methods to read or write values of the instance variables.

**Advantages of encapsulation:**

1. You can make your class read-only or write-only by declaring only getter or setter methods. This prevents other code or malicious code from accessing instance variables in your class that they should not read or modify.
2. You can add variable logic/constraints in the setter methods, so you have full control of the data that can be assigned to the instance variables.

Both of the above points contribute to the **objective of code-safety**, which we discussed at the start of this session.

To drive home the entire concept of encapsulation, let’s take the code of the Information Management System example we took in the last session and make the Student class a fully encapsulated class.

Here’s a quick summary of the topics you learnt about in this session:

1. **Private access modifier:** A data type that is declared while declaring any member of a class (instance variables, methods, etc.). Private access modifiers restrict the access of members within their own classes only.
2. **Declaring instance variables of a class as private:**All instance variables of a class should be declared as private in order to restrict the access to the variable’s data from outside the class, which makes the program safer.
3. **Getter methods to read the value of private variables:**A public getter method is declared in the class to give read-only access to the private variable from outside the class.
4. **Setter methods to write the value of private variables:** A public setter method is declared in the class to give write-only access to the private variable from outside the class. You can add constraints/logic of values to these variables.
5. **Declaring private methods and constructors as private:**Methods declared as private cannot be called from outside that class. Constructors declared as private don’t allow the creation of objects of that particular class outside that class.
6. **Making a fully encapsulated class:**We can boost the safety of a program by declaring all the data members/instance variables of that class as private. These classes can have read-only or write-only permissions by declaring getter or setter methods only.

Take a look at where you have reached in your **roadmap of learning object-oriented programming** and what you will learn in the upcoming modules.

The outer class in Java cannot be made private because you will not be able to use it or call its method anywhere in the code, even in the ‘main’ class; hence, it will be of no use.

The four main pillars of OOPS are abstraction, inheritance, encapsulation and polymorphism. While designing any problem using OOPS, implementing all these four features ensures that the OOPS convention is followed.

**Cards Game**

One of the main uses of Object-Oriented Programming is modelling requirements of a software design as classes and objects in the software program. Most of the times, you are given a scenario where you need to use OOP to create classes and objects in order to provide a design to the problem. This is entirely based on your judgement. In this segment, you will learn how to think of a solution to a vague problem of playing cards.

You are given a standard deck of 52 cards. You need to implement a general definition of cards and use it to build a classic game. This game will be played between two players, who will show their cards one by one, and the person who has the card of a greater value will get 10 points.

There is no particular answer for this and is highly based on an individual’s thinking but from the above problem statement you can assert to create mainly two classes:

1. Cards.
2. Classic game.

And you can use Inheritance or Composition to further improve the implementation of the software program and promote reusability of your code.  
Few fields that can be created for class Cards are ‘suit’, ‘value’, ‘color’, ‘points’, etc

## Inheritance & Polymorphism

**Why code reusability?**

The rising cost of building large software systems can be attributed to the fact that most of the development takes place from scratch. But by using the object-oriented programming methodology, large systems are built via modules that have been created earlier rather than by rewriting the code for the entire software from scratch.

The two main benefits of code reusability are:

1. It **saves on development effort** and, in turn, reduces the overall development cost of the software.
2. It **makes system maintenance easier**. This is because only the new module will have to be tested for errors, not the entire program.

Object-oriented programming provides these benefits through its last two principles:

1. Inheritance
2. Polymorphism

**What is inheritance?**

In object-oriented programming, inheritance provides a means by which an object of one class can use the properties of an object of another class.

**What does this imply?**

* A class can define certain behaviours that can be used later by new classes, instead of redefining the same functionalities again in the new classes.
* **A class that contains common functionalities for other classes to inherit is called a superclass**, base class or parent class.
* **A class that inherits from a superclass is called a subclass**, derived class or child class. A child class — apart from inheriting the properties of a superclass — can have its own additional properties in the form of its own methods and variables.

Hence, **a subclass can also be considered as a specialised version of a superclass.**

**Disadvantages of Inheritance:**

* Sometimes, inheritance is not useful because the data members and methods in the base class cannot be utilized to the fullest. At times, some of these members are left unused thereby resulting in wastage of memory.
* The inherited methods work slower than the normal methods due to the presence of indirection.
* Inheritance leads to the dependency of children classes on the base class thereby resulting in an increase in coupling.

**A subclass can inherit features from a superclass using the keyword ‘extends’**. So, if a class Square wants to inherit from a class Rectangle (Square is a type of Rectangle), **Rectangle is the superclass and Square is the subclass**, and Square has inherited all features of Rectangle class. The syntax of writing this will be:

**public** **class** **Square** **extends** Rectangle {

// class definition of Square

}

* **Every class in Java automatically extends the Object class.** Thus, by default, the Object class acts as the superclass for all the classes in Java.

Since a subclass will inherit all the features of its superclass, except its constructor, it must often invoke the superclass constructor for the initialisation of the superclass members. This is done using the keyword super in the constructor of the derived class.

* **If a subclass constructor does not explicitly call the superclass constructor, then the default constructor of the superclass is implicitly invoked by the compiler**. **If a superclass constructor is called using the keyword super, then this must be the first statement in the derived class constructor.**

Given below is an example of how you can call a superclass constructor from a base class:

**public** **class** **Square** **extends** Rectangle {

Square(argumentsList) { // constructor for the class Square

**super**(argumentsList); // calling the constructor of the superclass Rectangle

}

}

Observe the code snippet given below:

**public** **class** **Rectangle** {

**double** length, breadth;

Rectangle(**double** length, **double** breadth) {

**this**.length = length;

**this**.breadth = breadth;

}

}

**public** **class** **Square** **extends** Rectangle {

Square(**double** side) {

**super**(side, side); // invocation of the constructor of the superclass Rectangle

}

}

**Flow of Inheritance:**

In the above example, the *Square*class inherits the *Rectangle* class. When an object of the *Square*class is created, the constructor of the *Square*class is invoked. Inside the definition of the constructor of the *Square*class, the *super()* method is invoked. The super() method will call the superclass constructor, thus invoking the constructor of the *Rectangle* class. Thus, you can conclude that when an object of a subclass is created, the constructor of the subclass is invoked first and then the constructor of the superclass is invoked.

In the above case, since the superclass Rectangle has a parameterised constructor, the subclass constructor must explicitly invoke the constructor for the Rectangle class in its own constructor; failure to do so will result in a compilation error.

**When a subclass constructor calls the superclass constructor, this phenomenon is known as constructor chaining.**Apart from calling the constructor of the superclass, ‘super’ is also used to refer to the instance variables of a superclass.

**Example:** super.instanceVariable - Here, instanceVariable belongs to the superclass.

**public** **class** **Square** **extends** Rectangle {

**super**.length; // refers to the length attribute defined in the Rectangle class

}

1. When an object of the subclass is created it contains the superclass variables but it doesn’t create any superclass object in it.
2. When subclass object has created the constructor of the superclass is executed but this doesn’t mean that superclass object is created.
3. Go through the below program and its output.

**class** **Superclass**

{

**public** **Superclass**()

{

System.out.println("This is Super class constructor");

System.out.println(**this**.getClass().getName());

}

}

**class** **Subclass** **extends** Superclass

{

**public** **Subclass**()

{

System.out.println("This is Subclass constructor");

System.out.println(**this**.getClass().getName() + " " +

**super**.getClass().getName());

}

}

**public** **class** **Example**

{

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

{

Subclass Obje = **new** Subclass();

}

}

**Output:**

This is Super **class** **constructor**

Subclass

This is Subclass constructor

Subclass  Subclass

You can see from the output of the above program that, the object of the subclass is only created but both the constructors are executed. The superclass object and subclass object are both same.

Later in this module, we will look at another use of **super** to invoke the superclass method from the subclass method.

**Multiple Inheritance**

Java doesn’t support multiple inheritance because, consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

**class** **A** {

**void** **message**(){System.out.println("Class A");}

}

**class** **B** {

**void** **message**(){System.out.println("Class B");}

}

**class** **C** **extends** A,B {//suppose if it works

Public Static **void** **main**(String args[]){

C obje=**new** C();

obje.message();//which message() method would be invoked?

}

}

**Output:**

*Compile time error*

Note:

1.When an object of a subclass is created, the constructor of the class the object of which is created, which is the subclass, in this case, is invoked initially. Now, inside the constructor of a subclass, you have the super() method to invoke the constructor of the parent class, which is the superclass. Thus, first the constructor of the subclass is invoked, which invokes the constructor of the superclass. This option is the correct choice.

2. When the object of the subclass is declared then the constructor of the subclass is called first in which the first statement will be super(); which will, in turn, calls the constructor of the superclass, so the statements of superclass constructor are executed first and then the statements of the subclass. Even though subclass constructor doesn’t contain super(); it implicitly calls the superclass constructor. So constructor of class B is called first but statements of class A constructor will execute first.

3. When an object of the subclass is created, the constructor of the subclass will be invoked. The subclass constructor implicitly or explicitly invokes the constructor of the superclass. In case you want to explicitly call the superclass constructor, then the super() method must be invoked inside the subclass constructor definition. This method must be the first statement in the constructor definition before any other statement appears. Since the super() method invocation is not the first statement inside the constructor definition of Cube class, the code snippet will throw a compile-time error. This option is the correct choice.

**Inheriting from Subclasses**

**Protected Access Modifier**

You have already studied about two access modifiers - public and private in the last module. Let's now get introduced to the third type of access modifier - **protected** through an example.

Any method from within the Square class won’t be able to access Rectangle’s private attributes or methods directly. This may be fine sometimes; but at other times, it may be too restrictive. For example, suppose the Square class wants to access the length attribute declared in the Rectangle class for calculating the diagonal of a square.

**public** **class** **Square** **extends** Rectangle {

**public** **float** **diagonal**() {

**return** ((sqrt(**2**)) \* **super**.length); // Length is a private attribute defined in the Rectangle class

}

}

If the code given above is added to the Square class, it will lead to a compilation error. This is because ‘length’ is a private variable in the Rectangle class, and the private variables or methods of a class are not accessible outside the class. But what if the designer of the Rectangle class wants the subclasses to have direct access to the length attribute?

Let’s listen to our expert to provide a solution to this problem. Here, the member is neither being declared as private nor as public but will still be accessible to the subclasses.

Note :[Err 4:56 A protected variable or method can be accessed within the class, subclasses (in any package) as well as within all other classes inside the same package.]

Class members declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

In order to extend the visibility of the length attribute to the subclasses for direct access, you can use the **protected** **access specifier** in the Rectangle class, such as:

**protected** **float** length;

This would make the variable-length visible and accessible to all of the subclasses of the Rectangle class, such as the Square class, but not to any other classes that don't inherit from the Rectangle class and it is in a different package.

**Inheritance vs. Composition**

In the previous segment, you learnt about how one class can inherit from another. You saw how a Research Student class can inherit from a Student class and how a PhD Student class can inherit from a Research Student class.

But, how do you formally describe these relationships between different classes?

In this segment,

* We will discuss how you can formally describe relationships between classes built through inheritance.
* In addition to inheritance, we’ll also discuss a new concept called composition, which is another possible type of relationship between classes. We’ll tell you why composition is useful and when you should use it over inheritance, and vice versa.

Note:

Use inheritance only when the subclass is a proper subtype of the superclass. For example, ResearchStudent is a type of Student; hence, ResearchStudent will have an is-a relationship with the Student class. If some methods in the superclass are necessary, mandatory, or can be reused for its subclasses, use inheritance. On the other hand, use composition when you wish to use only some of the functionalities or characteristics of the class that you are creating a has-a relationship for. In the case of a Student class and a BankAccount class, you can say that a Student has-a Bank Account; hence, the has-a relationship holds true here.

Some reasons to use composition over inheritance are:

1. One reason for favouring Composition over Inheritance in Java is fact that Java doesn't support multiple inheritance.
2. Though both Composition and Inheritance allows you to reuse code, one of the disadvantages of Inheritance is that it breaks encapsulation. If the subclass is depending on superclass behaviour for its operation, it suddenly becomes fragile. When the behaviour of superclass changes, functionality in the subclass may get broken, without any change on its part.
3. Composition offers better test-ability of a class than Inheritance. If one class is composed of another class, you can easily create a Mock Object representing the composed class for sake of testing.
4. Like, There are many more reasons to favour Composition over inheritance

Let’s illustrate this relationship between a Shape class and a Rectangle class:

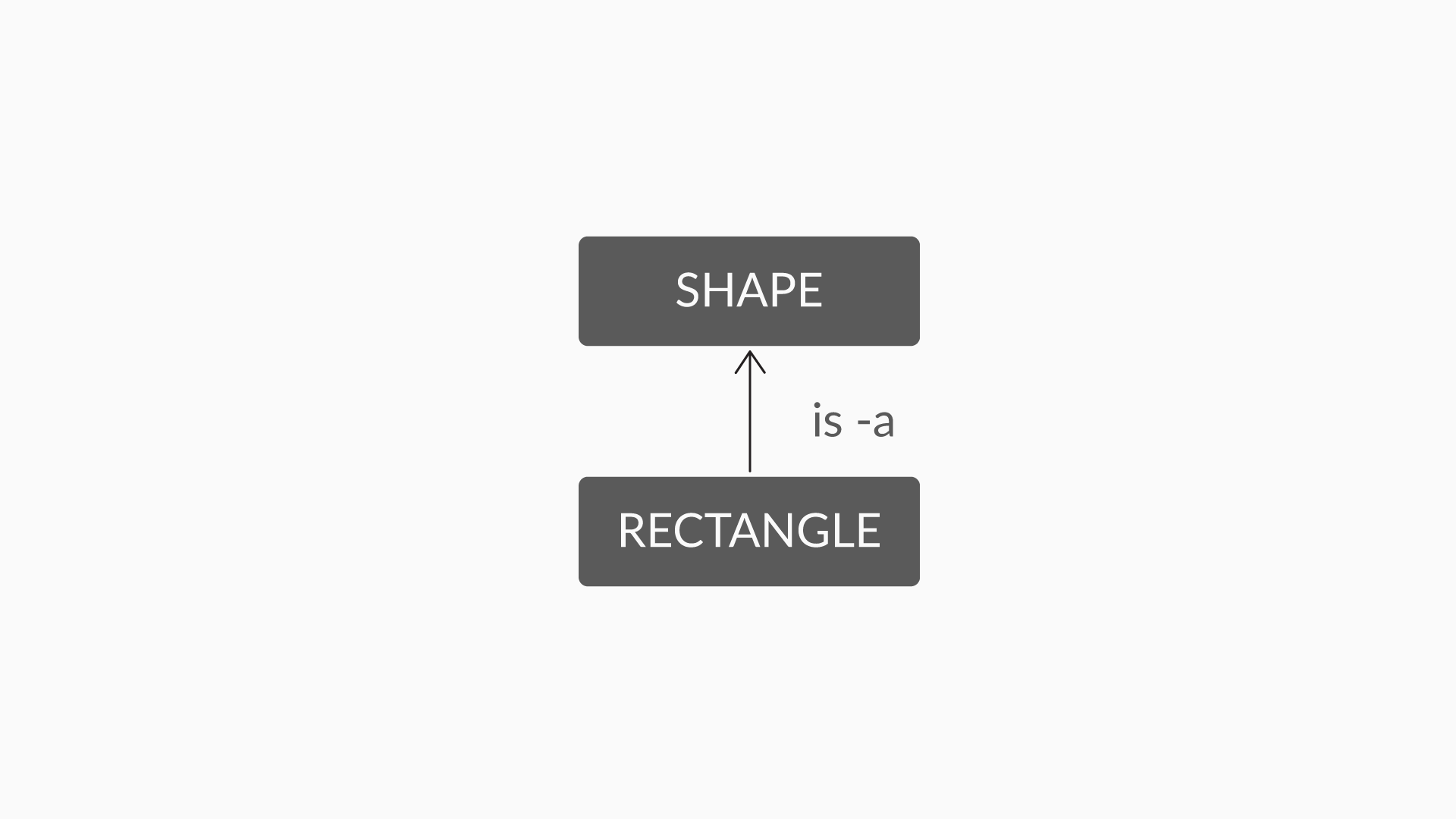


Figure 1 - is-a relationship

Now you know that shapes have some dimension. In this case, you can represent the dimensions of a shape with the help of a Dimension class. So now, the relationship between the Shape class and the Dimension class can be represented as follows:

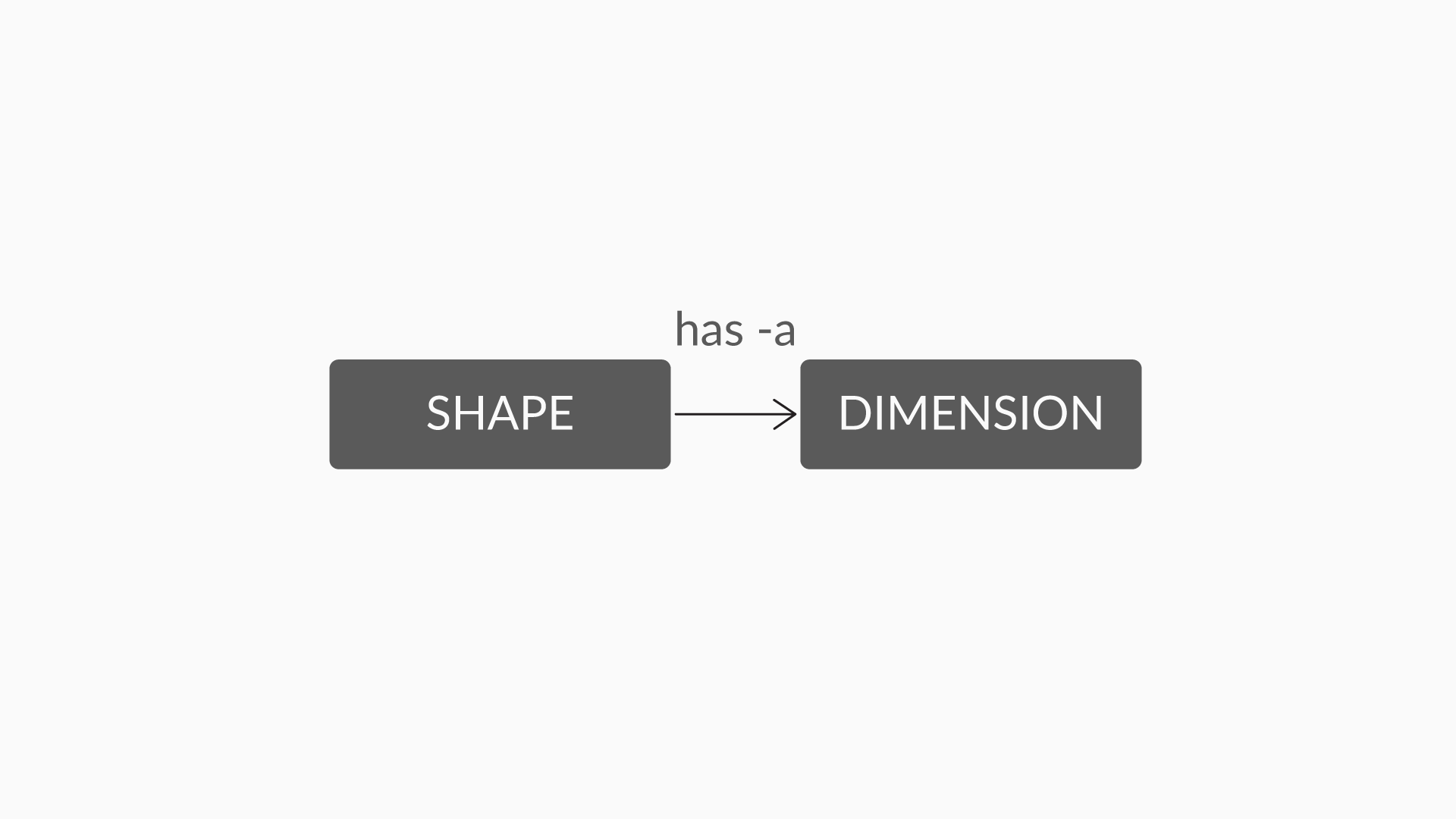


Figure 2 - has-a relationship

So, in the case of the Shape and Dimension classes, the composition relationship in a Java code will look like this:

public class Shape {

Dimension dimension;

}

The statement ‘Dimension dimension;’ creates an instance/object of the Dimension class inside the Shape class, which can be used by the Square class and the Rectangle class, which inherits from the Shape class.

#### Question: Modify Code

Given below is a Rectangle class consisting of an upper left coordinate and a lower right coordinate.

class Rectangle {

int upperLeftX;

int upperLeftY;

int lowerRightX;

int lowerRightY;

}

Since the upper left coordinate and lower right coordinate comprise of points, how would you modify the definition of the Rectangle class to include these points in your class?

Hint: You can create a Point class to store the x and y coordinates of the upper left and lower right points and use composition to include the upper left and lower right points inside the Rectangle class, as objects of the Point class.

Anse: **class** **Rectangle** {

Point ul;

Point lr;

}

**class** **Point** {

**int** x;

**int** y;

Point(**int** i, **int** j) {

**this**.x=i; **this**.y=j;

}

}

**Summary**

Let’s summarise what you learnt in this session:

1. **Inheritance:** The ability to create classes by making use of existing class’s members. This, in turn, lets you define a specialised version of the existing classes.
2. **Superclass and subclass:** The class that is being inherited (the generalised class) is known as the superclass, while the class that inherits the features of a superclass is known as the subclass (or the specialised class).
3. **Extends keyword:** To inherit the features of a superclass and derive a new class from it, you can use the extends keyword.
4. **Super keyword:**To call the superclass constructor from the subclass, or to refer to the superclass members from the subclass, you can use the super keyword.
5. ​​​​​​**Relationships:** There exist two types of relationships between classes in Java:
   * **Inheritance:** is-a relationship
   * **Composition:** has-a relationship

Use inheritance when there is enough opportunity for code reuse; otherwise, use composition.

Let's see where we have reached in the roadmap of object-oriented programming after this session on inheritance.

**Polymorphism**

One of the most important features of Object-Oriented Programming is its feature of **Polymorphism**. Polymorphism is essentially a Greek term which is derived from two Greek words - “Poly” meaning “many” & “morph” meaning “change in form”. So the basic definition of polymorphism means something which changes its form of occurrence.

**Polymorphism**

Polymorphism essentially means for something to occur in "different forms".

In the programming paradigm, it means the ability of a variable, function or object to take on multiple forms.

An example would be that many programming languages consider "integer", "float" and "double" as polymorphic forms of each other. They are different data types but one can add/subtract and perform many other arithmetic operations on them.

**The need for polymorphism**

A language that incorporates polymorphism allows developers to program in a universal manner which can be transformed and used in many paradigms, rather than write a very specific program which can only run in one scenario. For example, polymorphism allows us to consider an object as a generic version of something, but when you access it, the code determines which exact type it is and calls the associated code.

**Method Overriding**

In the previous session, you learnt that inheritance allows you to inherit methods from a superclass to a subclass without you having to redefine the same method again. But, what if you need to use the same method with an additional or modified functionality for your subclass? Let’s listen to our expert as he walks you through a similar situation and a solution to the same.

Keep in mind the following considerations while overriding a method:

* **The method name, the return type, and the argument list**of the overriding method in the subclass should match the method of the superclass.
* **The access modifier of the method in the subclass cannot be more restrictive than the one in the superclass.** This implies that if the method being overridden is declared as public in the superclass, it cannot be declared as private or protected in the subclass.
* **Methods defined as private, static, or final cannot be overridden.**

**Whenever a call to the overridden method is made, it is the subclass version of the method that is invoked and not the superclass version.**To call the superclass version of the overridden method, use the ‘super’ keyword along with the method name super.methodName();.

Note:

* 1. The access modifier of a method declared public in a superclass cannot be private in a subclass as it will be more restrictive when compared to the superclass version. Remember, the private access specifier only restricts access to a member or a method within the class.
  2. Any static method cannot be overridden because method overriding is decided during runtime but static methods or variables are decided during compile time itself. Since the main method is static we can’t override the main method.
  3. You cannot override all the methods in Java. Only the methods that are declared as public and protected can be overridden.

**Method Overloading**

Method overloading allows you to define multiple methods with the same name but with different definitions. But, wouldn’t it be a problem to have two or more methods with the same name in the same class?

Let’s listen to our expert who has a solution to such a problem.

Method overloading is also known as early binding as the type of the object is determined at the time of compilation.

**Two important considerations while using method overloading**are:

* The return type of a method may or may not be different.
* The parameter list MUST be different (either lengthwise or datatype wise) for each version of a method in a class.

Having the same parameter list but a different return type for a method will result in a compilation error.

**Example of method overloading:**

**class** **Addition**

{

**int** **add**(**int** a, **int** b)

{

**return** a+b;

}

**int** **add**(**int** a, **int** b, **int** c)

{

**return** a+b+c;

}

}

**public** **class** **Demo**

{

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

{

Addition obj = **new** Addition();

System.out.println(obj.add(**5**, **10**));

System.out.println(obj.add(**5**, **10**, **15**));

}

}

In the above program we have two definitions for the function add, so which one would be called will be determined at compile-time, by checking the parameters.

Hence method overloading is a type of Compile time polymorphism or Static polymorphism.

Note:

Method overloading is having two methods with the same name but a different parameter list, in the same class or subclasses. Clearly, both the versions of the method addition() have different parameter lists, and hence, these methods are overloaded. This option is the correct choice.

**Dynamic Polymorphism- I**

Recall that in the previous sessions you learnt the basics of Polymorphism, Method Overloading and Method Overriding. Let us now take a look at the two kinds of Polymorphism and draw the analogy between **Overloading / Overriding** & **Static / Dynamic Polymorphism**.

But let us first take a look at a very important concept in programming, namely what do we mean by compile-time and run-time.

**Compile-time**

Whenever a programmer creates a program, he writes a “code” for it which is known as the source code. This “source code” is then converted into machine code by a program called the Compiler, after which the program becomes executable by the machine.**This process of conversion of source code into machine code is called Compilation, and anything which happens during this time is called as “compile time”.**

Compile time errors are those which are detected by the compiler during the time of compilation. **They generally include errors arising out of mistakes in the syntax or semantics of the language.**Missing semicolons is an example of an error that is caught during compile time.

**Run-time**

A compiled program can then be executed and run by the user, and this process is called “running” a program. Anything that happens during this time is called **“run-time”**.

**Run-time errors are not detected during the time of compilation.**These type of errors can happen because of calling invalid functions, referencing missing variables or some logical mistakes.

## Two types of Polymorphism

Let us now see the two different kinds of Polymorphism and understand the basic difference between the two.

There are two types of polymorphism in Java:

a. **Static Polymorphism** is also known as **Compile Time Polymorphism.** This kind of polymorphism implies that **which method is to be called is to be decided during compile-time only.**

Recall that Method Overloading is a concept where we use the same method name many times in the same class, but with different arguments. **Method Overloading** would be an example of Static Polymorphism because in case of overloading, at compile time the compiler knows which method it should link to the call.

b. **Dynamic Polymorphism** is also known as **Runtime Polymorphism**. It is a process in which **a call to an overridden method is resolved at runtime.**

Recall that Method Overriding means having two methods with the same arguments, but different implementations. One of them would exist in the parent class, while another method with the same name and arguments can be written in the derived, which is the child class. **Method Overriding** is an example of Dynamic Polymorphism. In this case, **an overridden method is called through the reference variable of a superclass but the determination of the method to be called is based on the object being referred to by the reference variable.**

Let’s listen to our expert and find out more about this powerful feature and how it can prove beneficial for your programs.

Note:

1. Method overloading means that there can exist two functions with the same name but which take in different parameters. This is an example of static polymorphism. Method overriding means that at run-time, certain methods of functions can override other functions and this is an example of dynamic polymorphism.
2. Method overloading means that there can exist two functions with the same name but which take in different parameters. This is an example of static polymorphism. Method overriding means that at run-time, certain methods of functions can override other functions and this is an example of dynamic polymorphism.

**Abstract Classes - I**

When classes have common properties and share some relationship, you can implement this common behaviour using inheritance.

In our example, rectangles have areas and perimeters, and so do circles. Both are types of shapes, so you can now declare a new class named Shape and define both the area() and the perimeter() in this class. The class can now act as a superclass for Circle and Rectangle. Let’s go back to our expert and see him address a similar issue.

**An abstract class may or may not have abstract methods.**

It is important to note that you do not need to have only abstract methods in an abstract class. You can have non-abstract methods in it as well.

So, in order to declare the Shape class as abstract, the syntax is as follows:

**public** **abstract** **class** **Shape** {

**public** **abstract** **float** **area** ( ); // abstract methods

**public** **abstract** **float** **perimeter**( );

**public** **void** **resize** ( **float** newWidth, **float** newHeight) // non-abstract method

{

System.out.println("New width and height are : " + newWidth + "," + newHeight);

}

}

It is important to note that you cannot create instances of an abstract class. You can only inherit from an abstract class. Therefore, in order to use the abstract class Shape, the Rectangle class can extend it as follows:

**public** **class** **Rectangle** **extends** Shape {

// some code here

**public** **float** **area**() {

// implementation of the area() method

}

**public** **float** **perimeter**() {

// implementation of the perimeter() method

}

}

Remember that an abstract method should not be final. The whole purpose of declaring an abstract method is that the method can be used later in a subclass. You know that a final method cannot be overridden. So, if you define an abstract method as final, you will not be able to override it in any subclass. This defeats the purpose of declaring a method an abstract one in the first place.

Note:

An abstract class cannot be instantiated.

**✓ Correct**

**Feedback:**

You cannot instantiate an abstract class meaning that you cannot create an object of an abstract class. Thus, this option is one of the correct choices.



A method that is declared as abstract and does not have implementation is known as abstract method.

**✓ Correct**You missed this!

**Feedback:**

A method with no definition is an abstract method, which can be implemented by the class inheriting the abstract class. Thus, this option is one of the correct choices.

Non-abstract methods can be implemented in an abstract class.

**✓ Correct**

**Feedback:**

You can have abstract as well as non-abstract methods in an abstract class. This option is one of the correct choices.

An instance of an abstract class cannot be created, we can have references of abstract class type though. Abstract data class is not used to create objects in Java and it is designed only to act as a base class to be inherited by other classes

Q. **abstract** **class** **demo**

{

**public** **int** a;

demo()

{

a = **10**;

}

**abstract** **public** **void** **set**(**int** a);

**abstract** **final** **public** **void** **get**();

}

**class** **Test** **extends** demo

{

**public** **void** **set**(**int** a)

{

**this**.a = a;

}

**final** **public** **void** **get**()

{

System.out.println("a = " + a);

}

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

{

Test obj = **new** Test();

obj.set(**2**);

obj.get();

}

}

Ans: Compilation error

**✓ Correct**

**Feedback:**

Overriding final method is not possible, this implies that abstract functions shouldn’t be final. This option is the correct choice.

**Interfaces**

Assume that the shapes being defined in your program cannot be resized. Going by this constraint, the resize() method in the Shape abstract class declared earlier will no longer be a part of the class. What does this mean?

This means that the abstract class will now consist of only abstract methods. Such classes are represented in the form of interfaces. **An interface can thus be considered as a pure abstract class as it consists of only abstract methods.**

Let us now listen to our expert as he talks about the importance of interfaces and explains the situation where you use an interface over an abstract class.

Note: **From Java 8 onwards, interfaces too can consist of non-abstract methods (concrete methods) that are of the static and default types.**

**An interface is a collection of abstract methods and constant class variables, i.e. final static fields**. Since Shape does not have any concrete parts to it now, it can be represented in the form of an interface as shown below:

**public** **interface** **Shape** {

// abstract methods

**abstract** **float** **area**();

**abstract** **float** **perimeter**();

}

The syntax to use the Shape interface in a program is as follows:

**public** **class** **Rectangle** **implements** Shape {

// class definition for Rectangle

}

Just like an abstract class, an interface cannot be instantiated. Java allows a class to implement more than one interface. The syntax for implementing more than one interface is

public class **MyClass** implements **Interface1**, **Interface2**, **Interface3**…

Also, as you know, Java does not allow a class to extend multiple classes, but an interface can extend multiple interfaces. The syntax for extending multiple interfaces is —

public interface **MyInterface** extends **Interface1**, **Interface2**, **Interface3**…

You need to use the extends keyword once, followed by the comma-separated list of interfaces that you wish to extend.

Note:

1. Java allows a class to extend multiple interfaces only, not multiple classes. Hence, extending more than one class or inheriting from more than one class is not allowed in Java.
2. To invoke a superclass method from a subclass, the method name has to be prefixed with the keyword super as super.methodname().

**Summary**

Let’s go through a quick summary of what you learnt in this session:

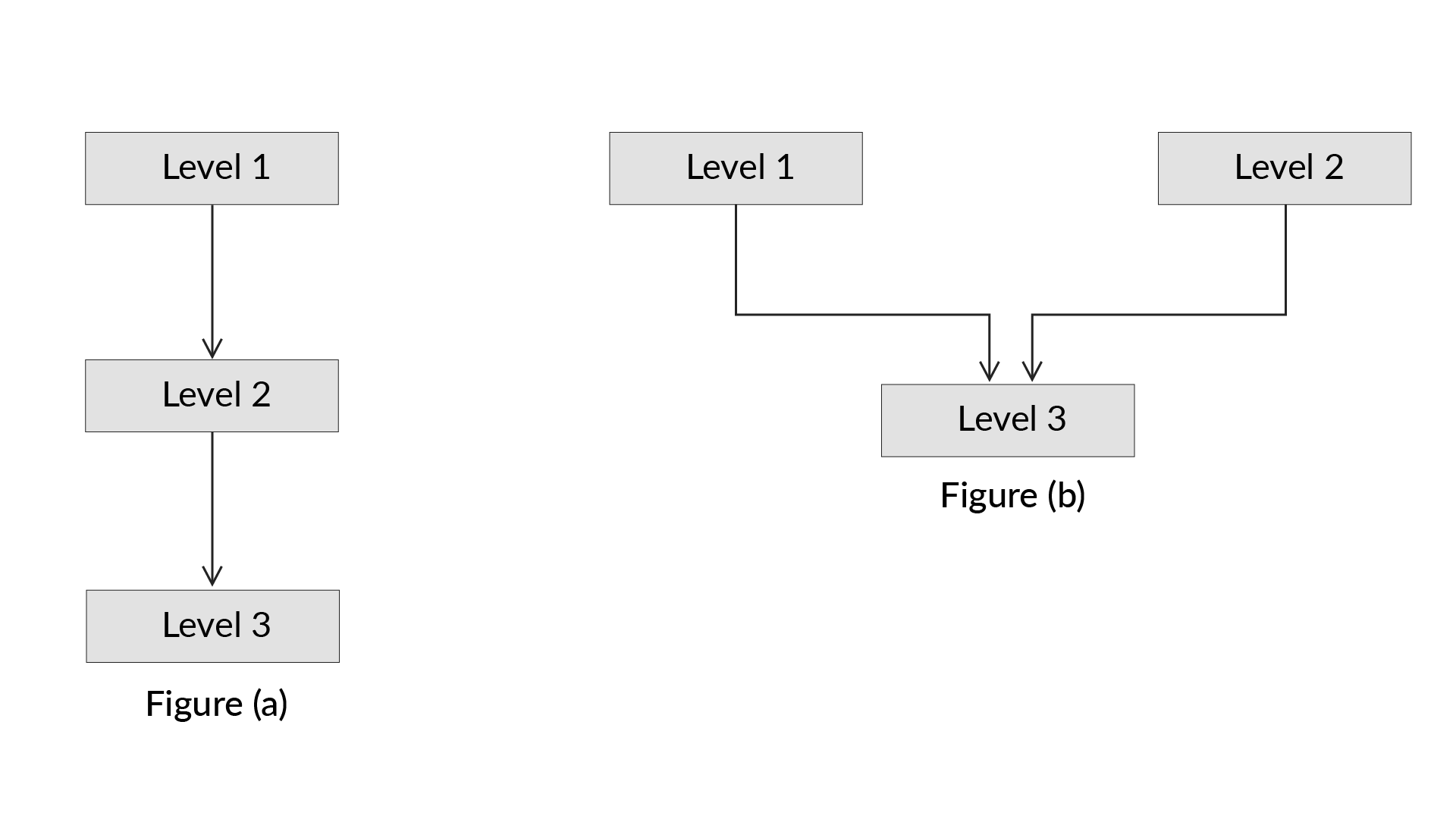
1. **Polymorphism:** It refers to the ability that allows methods with similar names to behave differently. Polymorphism can be either static or dynamic. It allows you to create extensible and flexible programs.
2. **Method overriding:**When a subclass decides to provide its own implementation of a method defined in its superclass, it is said to override the base version of the method. Method overriding is used to implement a specific type of polymorphism called dynamic polymorphism. In order to invoke the superclass version of an overridden method, you can use the super keyword.
3. **Method overloading:**When a class defines several methods with the *same*name, which perform the similar or the same tasks, but on different types or different numbers of parameters, the methods are said to be overloaded. Method overloading is commonly used to implement a specific type of polymorphism, known as compile time polymorphism.
4. **Compile-time polymorphism:** Whenever there exist two or more methods with identical names but with different parameter lists in the same class, which version of the method is to be invoked is decided at compile time. The decision is based on the parameters that are passed to the method.
5. **Run-time polymorphism:**Whenever a call to an overridden method is made, the decision to invoke the superclass version or the subclass version of the method is taken at run-time. This is based on the type of object, whether superclass or subclass, that is being used to invoke the method.
6. **Abstract classes:**The purpose of an abstract class is to share a common design or template, which other classes can inherit from or use to provide their own implementations. An abstract class consists of abstract methods (methods with no definition) and/or non-abstract methods and cannot be instantiated. They can only act as superclasses for other classes in the program. The definition of all the abstract methods of an abstract class must be provided by its subclasses.
7. **Interfaces:**An interface is considered to be a pure or a completely abstract class. They define a programming contract that must be fulfilled by the classes that implement the interface. In other words, all the abstract methods of an interface have to be defined in the classes that implement the interfaces. The **interface**keyword is used to define a new interface, while the **implements**keyword is used for implementing the interface in a class.

**Number System & Inheritance**

Inheritance is a feature of Object Oriented Programming that allows you to reuse the features of one class in other classes. The class from which the features are derived is called the **parent class** or the **superclass**, and the class that reuses the features of the parent class is called the **child class** or the **subclass**.

Being one of the main features of Object-Oriented Programming, **inheritance** needs to be understood really well. Attempt the questions given below to revise the concepts of Inheritance.

#### Q. Inheritance



The diagram above represents two different types of inheritance. Analyse-it and identify the type of inheritance each is representing.

Note :

Level 1 - Class A

Level 2 - Class B

Level 3 - Class C

Ans: Figure(a) is multilevel inheritance and Figure(b) is multiple inheritance .

**✓ Correct**

**Feedback:**

In multilevel inheritance, a class is derived from one class, and the derived class later becomes the base/parent class for another class. For example, class B is derived from class A, which makes class A the parent class of class B, which is the sub/derived class. Now, if you derive a class C from class B, this type of inheritance is known as multilevel inheritance. In Figure(a), class B is derived from class A and class C is derived from class B, which satisfies the definition of multilevel Inheritance. Therefore, it is a case of multilevel inheritance.  
In multiple inheritance, two or more classes act as a parent class for one child/subclass. For example, if class C is derived from both class A and class B, it is a case of multiple inheritance. But, Java doesn’t support multiple inheritance because, consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.