**Polymorphism in Java:**

* Polymorphism in Java is a concept by which we can perform a single action in different ways.The word "poly" means many and "morphs" means forms.
* That is, the same entity (method or operator or object) can perform different operations in different scenarios.

**Real-life Example:**

In a scenario like driving a vehicle. Different vehicles such as a car, a bike, and a bus, all have a common action that is they can "move". However, the way they move differs.

**In this example:**

* The action "move" is like a method in object-oriented programming.
* Each vehicle represents a different object or class.
* Despite calling the same method ("move"), each object executes it differently based on its type.

**Java Based Example:**

Suppose we have created a superclass: Polygon and two subclasses: Square and Circle. And we have render() method.

The main purpose of the render() method is to render the shape. However, the process of rendering a square is different than the process of rendering a circle.

Hence, the render() method behaves differently in different classes. Or, we can say render() is polymorphic.

**Types of Polymorphism:**

There are two types of polymorphism

1. **Compile-time Polymorphism**
2. **Runtime Polymorphism**
3. **Compile-time Polymorphism**: Also known as static polymorphism, it is achieved through method overloading. It allows multiple methods with the same name but different parameters in a class.The appropriate method is determined at compile-time based on the method arguments.

**2**. **Run-time Polymorphism:** Also known as dynamic polymorphism, it is achieved through method overriding. It allows a subclass to provide its own implementation of a method defined in its superclass.In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable at runtime.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Upcasting:**

Upcasting involves treating an object of a subclass as an object of its superclass. It's implicit and safe, as it's moving up the class hierarchy. For example, if you have a Dog class extending Animal, you can upcast a Dog object to an Animal reference: Animal animal = new Dog();

**Downcasting**:

Downcasting involves treating an object of a superclass as an object of its subclass. It's explicit and potentially unsafe because it's moving down the class hierarchy. You need to explicitly cast the superclass reference to the subclass type. For example, if you have an Animal reference and you know it refers to a Dog object, you can downcast it: Dog dog = (Dog) animal; However, if the actual object isn't of the specified subclass type, it will throw a ClassCastException at runtime.

**Note:** What it is able to access is defined by reference which one it will able to access is defined by the object.

Parent obj = new Child();

here parent is reference and child is object created of

This is how method overriding works

**Android Development Example: Music Player App**

Imagine a music player app with different types of playlists:

Parent Class - Playlist: Create an abstract class Playlist with a method play().

Child Class - FavoritePlaylist: Make a class FavoritePlaylist that extends Playlist. It overrides play() to play favorite songs.

Child Class - PartyPlaylist: Similarly, create PartyPlaylist that extends Playlist, playing energetic songs.

Both playlists use the play() method, but each subclass has its own version. This lets the app handle different playlists uniformly, showcasing polymorphism.