Class # 10

1. **Polymorphism**

    - Polymorphism Concept
    - Method Overloading
    - Method Overriding

Polymorphism is a fundamental concept in object-oriented programming and is a key feature of the Java programming language. It allows objects of different classes to be treated as objects of a common superclass. In simpler terms, polymorphism enables you to work with objects of different types through a common interface.

There are two main types of polymorphism in Java:

Compile-Time Polymorphism (Static Binding):

Compile-time polymorphism, also known as static binding or method overloading, occurs when multiple methods in the same class have the same name but different parameters (different number or types of parameters). The compiler determines which method to call based on the method's signature at compile time. Here's an example:

java code

```
class Calculator {
int add(int num1, int num2) {
return num1 + num2; }
double add(double num1, double num2) {
 return num1 + num2; }
String add(String str1, String str2) {
 return str1 + str2; } }
public class CompileTimePolymorphism {
```

```
public static void main(String[] args) {

Calculator calculator = new Calculator();

int sum1 = calculator.add(5, 7); // Calls the int version of add

double sum2 = calculator.add(3.14, 2.71); // Calls the double version of add

String result = calculator.add("Hello, ", "world!"); // Calls the String version of add System.out.println("Sum1: " + sum1);

System.out.println("Sum2: " + sum2);

System.out.println("Concatenated String: " + result);

} }
```

Run-Time Polymorphism (Dynamic Binding):

Polymorphism is a fundamental concept in object-oriented programming, allowing objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent different types of objects, providing flexibility and extensibility to your code. In Java, polymorphism is typically achieved through method overriding and interfaces. Also known as method overriding.

Occurs when a subclass provides a specific implementation of a method that is already defined in its superclass.

The correct method to be executed is determined at runtime based on the actual type of the object.

This is achieved by using the @Override annotation and creating a method with the same signature in the subclass.

Here's a simple Java example that demonstrates polymorphism:

java code

```
class Animal {
```

```
void makeSound() {
System.out.println("Some Animal sound"); } }
class Dog extends Animal {
@Override void makeSound() {
System.out.println("Dog Sound Bark"); } }
class Cat extends Animal {
@Override void makeSound() {
 System.out.println("Cat Sound Meow"); } }
public class PolymorphismExample {
public static void main(String[] args) {
Animal commonAnimal = new Animal();
Animal myDog = new Dog();
Animal myCat = new Cat();
commonAnimal.makeSound(); // Calls Common Animals Sound
myDog.makeSound(); // Calls Dog's makeSound method
myCat.makeSound(); // Calls Cat's makeSound method
}
}
```

In this example:

1. We have a superclass **Animal** with a method **makeSound()**.
2. Two subclasses, **Dog** and **Cat**, extend the **Animal** class and override the **makeSound()** method to provide their own implementations.
3. In the **PolymorphismExample** class, we create objects of type **Animal** but instantiate them as **Dog** and **Cat**.

4. When we call the **makeSound()** method on these objects, it invokes the overridden method of the actual object type (polymorphism). This allows us to call **makeSound()** on an **Animal** reference, but the specific implementation is determined at runtime based on the object's actual type.