# Polymorphism in Java

Polymorphism is one of the four fundamental principles of object-oriented programming (OOP), along with encapsulation, inheritance, and abstraction. In Java, polymorphism allows you to write code that can work with objects of different classes in a uniform way. There are two types of polymorphism in Java: compile-time (static) and runtime (dynamic) polymorphism.

## 1. Compile-time (Static) Polymorphism:

- Compile-time polymorphism is also known as method overloading. It occurs when two or more methods in the same class have the same name but different parameters.
- The appropriate method to call is determined by the number and types of arguments passed at compile time.

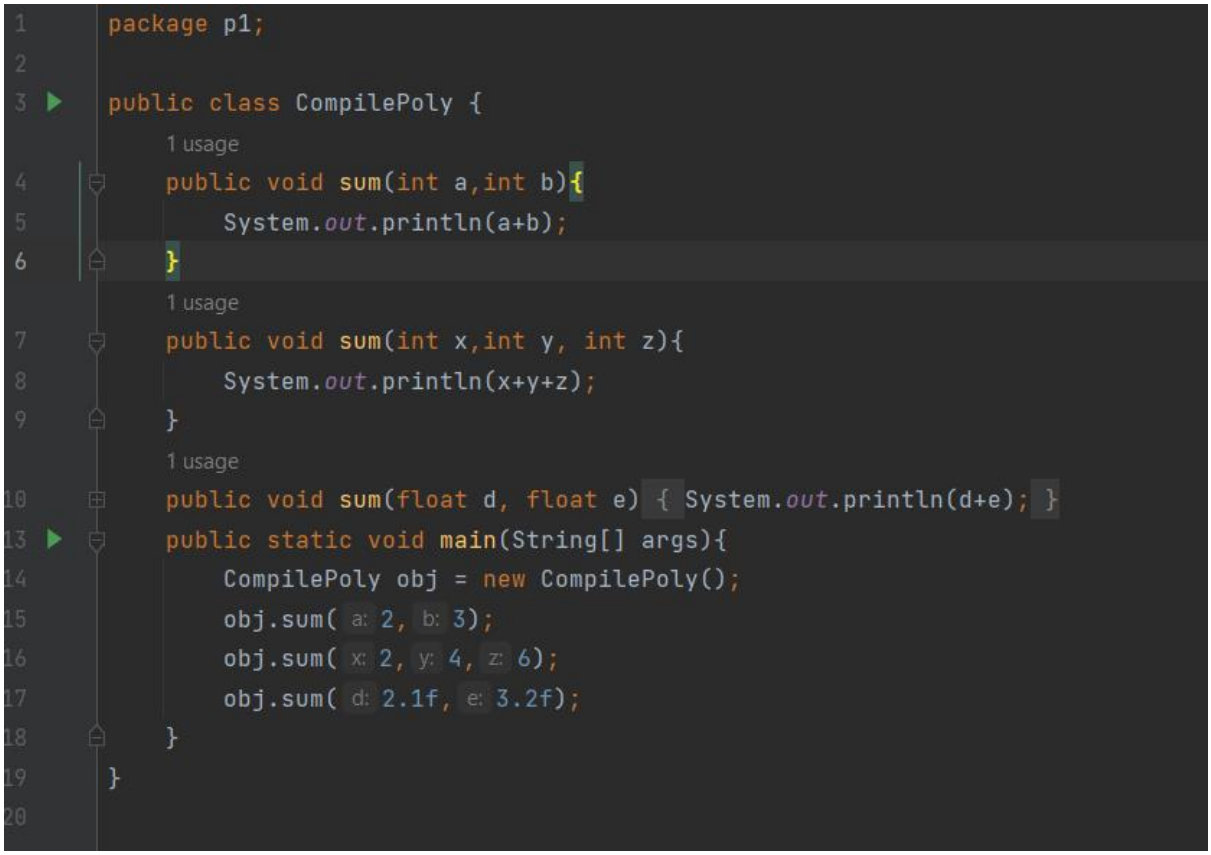
## Example –

```java
package p1;
class india{
    2 usages  2 overrides
    public void Language() {
        System.out.println("I can speak multiple languages");
    }
}
1 usage
class kerala extends india{
    2 usages
    @Override
    public void Language() {
        System.out.println("I can speak malayalam");
    }
}
class tamilNadu extends india{
    2 usages
    public void Language() { System.out.println("I can speak tamil"); }
}
public class RunPoly {
    public static void main(String[] args){
        india obj = new india();
        india obj1 = new kerala();
        obj.Language();
        obj1.Language();
    }
}
```

## 2. Runtime (Dynamic) Polymorphism:

- Runtime polymorphism is achieved through method overriding and is sometimes called "polymorphism" in the context of OOP.
- Method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass.
- The decision about which method to call is made at runtime based on the actual object type.

Example –



```java
package p1;

public class CompilePoly {
    1 usage
    public void sum(int a,int b){
        System.out.println(a+b);
    }
    1 usage
    public void sum(int x,int y, int z){
        System.out.println(x+y+z);
    }
    1 usage
    public void sum(float d, float e) { System.out.println(d+e); }
    public static void main(String[] args){
        CompilePoly obj = new CompilePoly();
        obj.sum( a: 2, b: 3);
        obj.sum( x: 2, y: 4, z: 6);
        obj.sum( d: 2.1f, e: 3.2f);
    }
}
```

# Life cycle of an object in java

he lifecycle of an object in Java refers to the various stages an object goes through from its creation to its eventual destruction by the garbage collector. Understanding the object lifecycle is crucial for efficient memory management and resource utilization. Here are the key stages in the lifecycle of a Java object:

- **Object Creation**: Objects are created using the new keyword, constructor methods, or other object creation mechanisms. During this stage, memory is allocated for the object, and the constructor is called to initialize its fields.

- **Initialization**: The constructor(s) of the object are executed to set its initial state. This stage is essential to ensure that the object is in a consistent state when it is first created.
- **Object Reference**: Objects can be referenced by variables, data structures, or other objects. When an object is referenced, it can be used and manipulated by the program.
- **Object Use**: During this stage, the object is actively used in the program. Methods are called on the object, and its data is accessed and modified as needed.
- **Scope and Lifespan**: The object's scope and lifespan are determined by where it is referenced. As long as there are references to the object in the program, it remains in scope and accessible. When there are no more references to the object, it becomes eligible for garbage collection.
- **Garbage Collection Eligibility**: An object becomes eligible for garbage collection when there are no strong references to it. Weak references, soft references, or phantom references may still exist, but they don't prevent the object from being garbage collected.
- **Garbage Collection**: When the Java Virtual Machine (JVM) determines that an object is no longer reachable and eligible for garbage collection, it reclaims the memory occupied by the object. The process of identifying and reclaiming such objects is performed by the garbage collector.

- **Finalization**: If an object has a finalize method (deprecated as of Java 9), it may be called by the garbage collector just before the object is reclaimed. This method can be used for cleanup operations or resource release.
- **Deletion**: After garbage collection, the object is removed from memory, and its resources are deallocated. The object no longer exists and cannot be accessed.

Example –

```
package p1;
class Math{
    1 usage
    public void Mul(int a,int b){
        System.out.println(a*b);
    }

}
public class LifeOfObject {
    public static void main(String[] args){
        Math obj = new Math();
        obj.Mul( a: 2, b: 3);
        obj = null;
        System.gc();


    }
}
```

# Fundamentals of Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm that is based on the concept of "objects." It is a fundamental approach to software development that helps in organizing and structuring code in a more modular and reusable manner. OOP is widely used in many programming languages, including Java, C++, Python, and more. Here are the fundamental concepts of object-oriented programming:

- **Class**: A class is a blueprint or a template for creating objects. It defines the properties (attributes) and behaviours (methods) that objects created from the class will have. For example, if you're creating a class for a "Car," it would define attributes like make, model, and colour, and methods like "start" and "stop."
- **Object**: An object is an instance of a class. It is a self-contained unit that contains both data (attributes) and the methods (functions) that operate on the data. Each object created from a class can have its own values for the attributes while sharing the same methods defined in the class.
- **Encapsulation**: Encapsulation is the principle of bundling data (attributes) and methods (functions) that operate on the data into a single unit (an object). It also involves hiding the internal details of how an

object works from the outside and exposing a well-defined interface. This helps in reducing complexity and enhances data security.

- **Inheritance**: Inheritance is a mechanism that allows you to create a new class based on an existing class. The new class inherits the properties and behaviors (attributes and methods) of the existing class, and you can also add new attributes and methods or override existing ones as needed. Inheritance promotes code reuse and the creation of hierarchical relationships between classes.
- **Polymorphism**: Polymorphism means the ability to take on multiple forms. In OOP, polymorphism allows objects of different classes to be treated as objects of a common base class. This enables you to write code that can work with objects of different types in a uniform way, leading to more flexible and extensible code.
- Abstraction: Abstraction is the process of simplifying complex reality by modelling classes based on the essential attributes and behaviours. It involves defining a clear and simplified interface for an object while hiding the underlying complexity. Abstraction helps in managing complexity and focusing on what an object does rather than how it does it.
- **Method**: A method is a function defined within a class that can perform specific actions or operations on the attributes of an object. Methods are called on objects to manipulate data or perform tasks associated with those objects.

- **Attribute**: An attribute, also known as a member variable or property, is a data element that belongs to an object. It represents the state of the object and defines its characteristics. For example, the attributes of a "Car" class might include "make," "model," and "color."