Amrita Vishwa Vidyapeetham
TIFAC-CORE in Cyber Security

20CYS312 - Principles of Programming Languages
Assignment-01: Exploring Programming Paradigms

Achyuth Anand

21st January, 2024

Paradigm 1: Object Oriented Programming

Object Oriented Programming(OOPS) involves the creation of objects that contain both data and methods. Data is contained in the object in the form of fields and the object is referred to using special keywords such as ***This*** or ***Self***. OOPS was developed from the 1950s and slowly, was incorporated in programming languages. Simula, in the mid 1970s introduced the concept of class and objects, which continue to be used in OOPS programming today. Lisp, a family of programming languages were influenced by ***smalltalk***, a purely OOPs based language developed in the 1970s to incorporate the use of OOPs concepts in their languages as well. These included Visual FoxPro 3.0, C++, and Delphi.

Subsequent development in the field of OOPS led to the incorporation of OOPS as the dominant programming paradigm in the 1990s when features encouraging the use of OOPS became popular. The following subsections discuss the various aspects of OOPS.

Overview

Languages that incorporate the use of OOPS, use the property of ***inheritance*** to reuse code. They also use inheritance for class extensions and prototype development. A ***class*** is the definition of a user-defined data format and specifies the blueprint for the creation of a type or class of objects. It may also specify data, methods and procedures(class methods). An ***object*** may be defined as an instance of a class. Class methods and data can be accessed through the creation of objects. The following are some of the essential components of a class :

- ***Class variables*** : A variable that belongs entirely to the class and is shared across all instances of the class.
- ***Instance variable*** : Variables that are unique to each object and often used as parameters for calling a specific class method for that particular object.
- ***Member variable***: A variable that belongs to an object and can be used to access all methods.
- ***Class methods*** : A function that belongs to the class as a whole and associated with an object. It is a behaviour of the object parameterized by the user.
- ***Instance method*** : A function that belongs to the object and have access to the instance variable for the respective object,inputs and class variables.

The following subsection discusses the features present in Object Oriented Programming.

Features

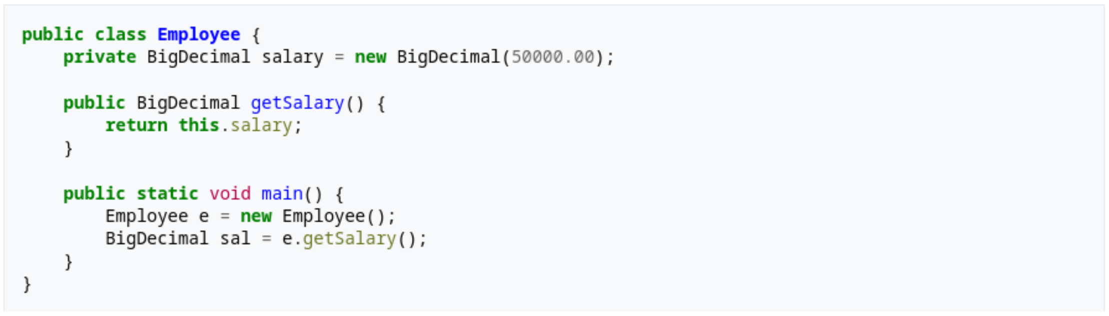
- ***Class-based and prototype-based***: In ***class-based programming***, classes are defined beforehand and the object is instantiated based on the class. The structure of the object, its behaviour are determined by the class. Objects are entities that encompass state, behaviour, and identity. State refers to the data associated with the object, behaviour refers to the procedures or methods that the object can perform, and identity refers to the object's unique existence among all other objects. An object is similar to a structure, with an additional method pointers, member access control, and an implicit data member which locates instances of the class.

  Prototype based programming gives the objects the monopoly of independent entities. No primary class exists. The prototype of an object is just another object on which the object is linked. For example, it is possible to create an object named fruit, and have two independent objects, mango and grapes as fruit as their prototypes. Each object has only one prototype link. This means that an object can contain the properties of only one other object(the prototype).

Objects are mutable, thus allowing for newer instances and giving newer methods and functions. A new class definition is not required for creating similar objects. Many experts argue that prototype based programming allows the user to focus on the behaviour of the set of objects and only later group these objects into a object replicating the functionality of the class. They argue that additional focus on the behaviour of the object allows the user to easily classify them into a prototype object and create the prototype link. The prototype property is called **prototype** in Self and JavaScript, or proto in Io.

Many of the programming languages offer the luxury of assigning the first newly created object called the ***root object*** as the default prototype to all subsequent objects created at run-time and carrying commonly used methods. **Cloning** is a process by which new objects inherit the properties of the old object(prototype) and the child objects maintain an explicit prototype link. Any changes done to the prototype object are reflected on the child object as well. However, some languages such as ***kevo*** do not reflect the changes done to the prototype in the child object.

- ***Encapsulation*** is the process of concealing some of the more sensitive/redundant(from the user point of view) aspects of the class from the user using access specifiers to maintain the structure of the class. The definitions of encapsulations focuses on grouping and packaging of related information, instead of security issues. However, security is a key aspect of encapsulation as it prevents the exposure of internal methods or variables from being directly accessed by clients or users. This maintains a consistency in the performance of the object defined as the instance of that class. Given below is an example in Java that shows how access to a data field can be restricted through the use of a private keyword:



```java
public class Employee {
    private BigDecimal salary = new BigDecimal(50000.00);

    public BigDecimal getSalary() {
        return this.salary;
    }

    public static void main() {
        Employee e = new Employee();
        BigDecimal sal = e.getSalary();
    }
}
```

Figure 1: Encapsulation

- ***Inheritance*** in class-based programming,is a process where new classes (child classes) are created as extensions of existing ones (parent classes). In typical class-based object-oriented languages such as C++, the child object acquires all the attributes of the parent class, except for constructors, destructors, overloaded operators, and friend functions of the base class. Single inheritance is when a child class (subclass) inherits the property from one parent class(super class). Multiple inheritance is when a child class inherits from multiple super classes. Multilevel inheritance (or Hierarchical inheritance) is when a subclass inherits the properties from another subclass as opposed to a superclass. Figure 2 shown above illustrates the different types of inheritance possible.

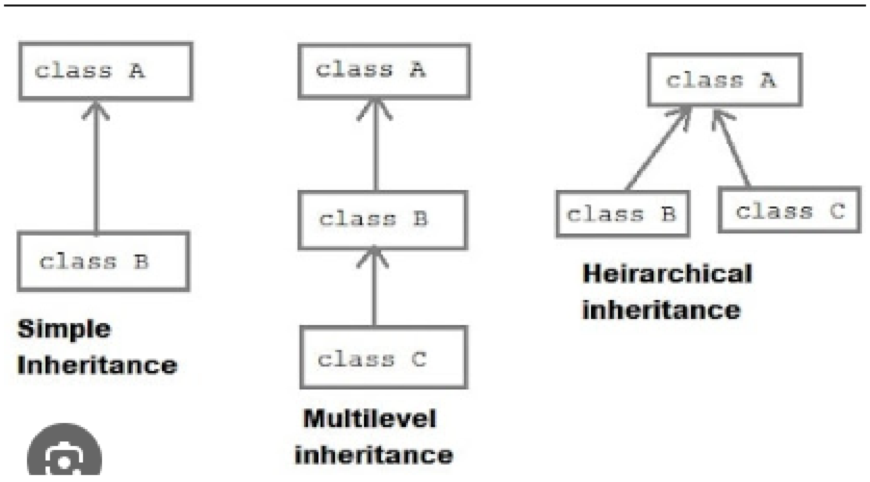


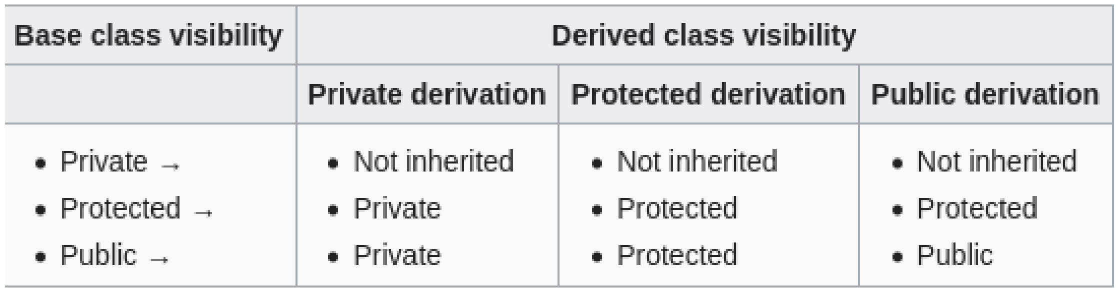
Figure 2: Types of Inheritance

Many OOPS programming languages contain the feature of ***Non-subclassable subclass***, which means that there exists no other class that inherits from this particular class. This functionality is useful for the system to determine at compile-time that pointers or references to objects instantiated to this class are pointing to this class exclusively and contain no inheritance.

Just like a subclass that cannot be inherited by another class, non- ***overrideable methods*** may have method modifiers that prohibit the method from being overridden. A method can be declared with the private access specifier, which prevents it from being overridden because it is not accessible outside the class.

If a superclass method is a ***virtual method***, it will be dynamically dispatched. Dynamic dispatching refers to the process of selecting the correct polymorphic function to be executed at runtime.

The child class can derive the properties from the superclass in different manners, namely, ***Private, Protected and Public***. Figure 3 illustrates the visibility of super class members in the child class.



Base class visibility	Derived class visibility		
	Private derivation	Protected derivation	Public derivation
• Private →	• Not inherited	• Not inherited	• Not inherited
• Protected →	• Private	• Protected	• Protected
• Public →	• Private	• Protected	• Public

Figure 3: Derived Class Visibility

- ***Dynamic dispatch/message passing***: Dynamic dispatch is the process of selecting which polymorphic implementation of a method to be selected at runtime. ***Polymorphism*** is the occurrence where objects, which can be used interchangeably, have an action with the same name but potentially different behaviour. If there exist two different objects and one of those objects calls a function, the system, at run-time, will be tasked with the challenge of sending the method to the right object. The dispatch mechanism normally functions by identifying the type and the number of arguments specified by the user. Languages with weak or no typing system, often carry a dispatch table as part of the object data for each object.

  C++ utilises dynamic dispatch when a user specifies a virtual function. They normally do so with the help of a data structure in C++ called a virtual function table that specifies the name to implementation mapping for a given class as a set of member function pointers. Type overloading does not produce dynamic dispatch in C++, because the type is seen as a part of the message name by the system. This means that the name given by the user is not the name used for binding. The below figure 4 displays the scenario in which dynamic dispatch can be used.

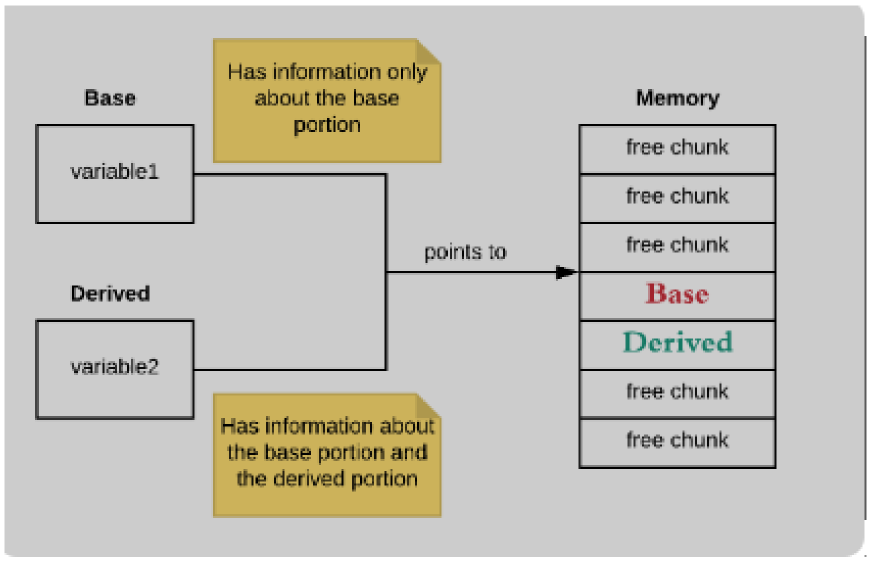
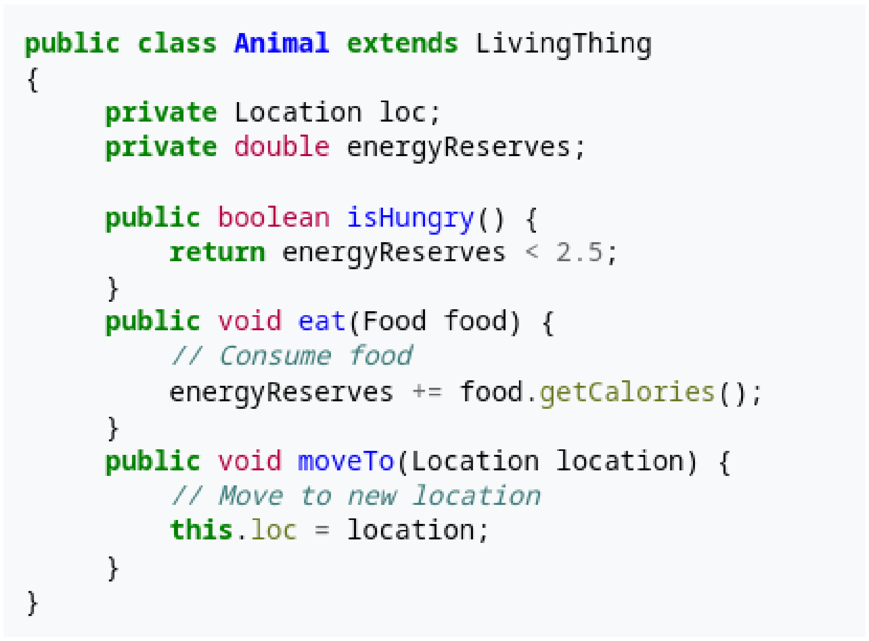

Base
variable1
Has information only about the base portion
Derived
variable2
Has information about the base portion and the derived portion
points to
Memory
free chunk
free chunk
free chunk
Base
Derived
free chunk
free chunk

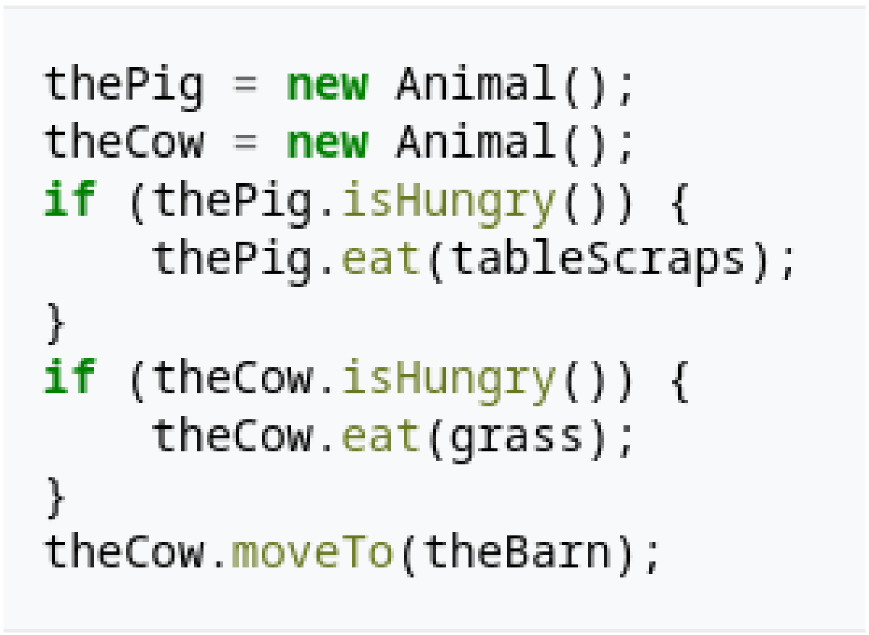
Figure 4: Dynamic Dispatch

- ***Data abstraction***: This feature allows for data to be viewed only by semantically similar functions so as to prevent misuse. This is often used for data hiding in OOPS and procedural programming as well. A class may not allow direct access to internal data and the data may be accessed only through methods. These methods also govern how this data is accessed or modified and are thus instrumental in preventing misuse of the data. Some languages like Java prevent the access of internal data through the use of ***private*** keyword and specify the methods meant for external use through the keyword ***Public***. Figures 5 and 6 illustrate a simple example to demonstrate data abstraction in Java.



```java
public class Animal extends LivingThing
{
     private Location loc;
     private double energyReserves;

     public boolean isHungry() {
         return energyReserves < 2.5;
     }
     public void eat(Food food) {
         // Consume food
         energyReserves += food.getCalories();
     }
     public void moveTo(Location location) {
         // Move to new location
         this.loc = location;
     }
}
```

Figure 5: Abstraction

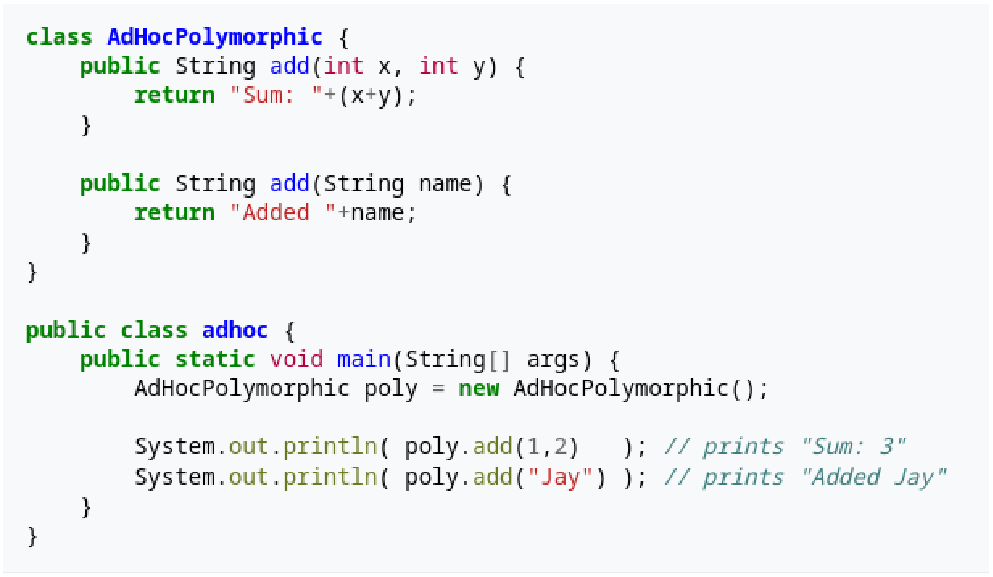


```java
thePig = new Animal();
theCow = new Animal();
if (thePig.isHungry()) {
    thePig.eat(tableScraps);
}
if (theCow.isHungry()) {
    theCow.eat(grass);
}
theCow.moveTo(theBarn);
```

Figure 6: Abstraction-2

- ***Polymorphism***: It is defined as the assignment of a single interface to different entities of different types. It can be classified into 3 types of polymorphism.

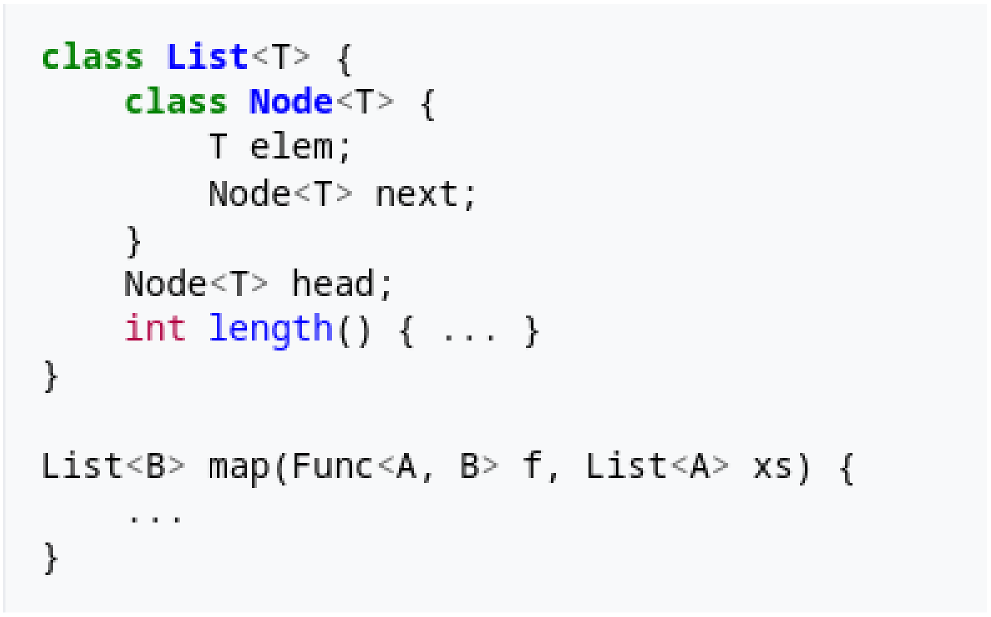
  Ad-hoc polymorphism refers to polymorphic functions which have different datatypes as arguments. Depending on the nature of the datatype, the function behaves in a certain manner or differently (function overloading or operation overloading). Figure 7 specifies an example to illustrate how the AD-HOC polymorphism occurs in Java. However, in dynamically typed languages, the situation becomes more complex as the function that needs to be invoked is determined only at run-time.



```java
class AdHocPolymorphic {
    public String add(int x, int y) {
        return "Sum: "+(x+y);
    }

    public String add(String name) {
        return "Added "+name;
    }
}

public class adhoc {
    public static void main(String[] args) {
        AdHocPolymorphic poly = new AdHocPolymorphic();

        System.out.println( poly.add(1,2)   ); // prints "Sum: 3"
        System.out.println( poly.add("Jay") ); // prints "Added Jay"
    }
}
```

Figure 7: AD-HOC Polymorphism

Parametric Polymorphism allows a function to be written generically so that it need not depend on the type of the argument provided to sort out the correct function. A polymorphic function is a function that can be evaluated or applied to values of different types. A data type that can take on the characteristics of a generalised type (e.g. a list containing entries of any type) is referred to as a polymorphic data type, which is derived from the generalised type it specialises. Figure 8 illustrates an example of a generic function in C++.



```
class List<T> {
    class Node<T> {
        T elem;
        Node<T> next;
    }
    Node<T> head;
    int length() { ... }
}

List<B> map(Func<A, B> f, List<A> xs) {
    ...
}
```

Figure 8: Parametric Polymorphism

Language for Object Oriented Programming: Java

Java is a programming language that is object-oriented, class-based, and high-level. It is specifically designed to minimise implementation dependencies. Java is a versatile programming language designed to enable programmers to build code that can be executed on any platform without the need for recompilation. Java typically runs by converting the code into ***bytecodes*** and can run on any Java Virtual Machine(JVM).

Principles

There were five primary goals in the creation of the Java language:

- It must be simple, object-oriented, and familiar.
- It must be robust and secure.
- It must be architecture-neutral and portable.
- It must execute with high performance.
- It must be interpreted, threaded, and dynamic.

General Features of Java

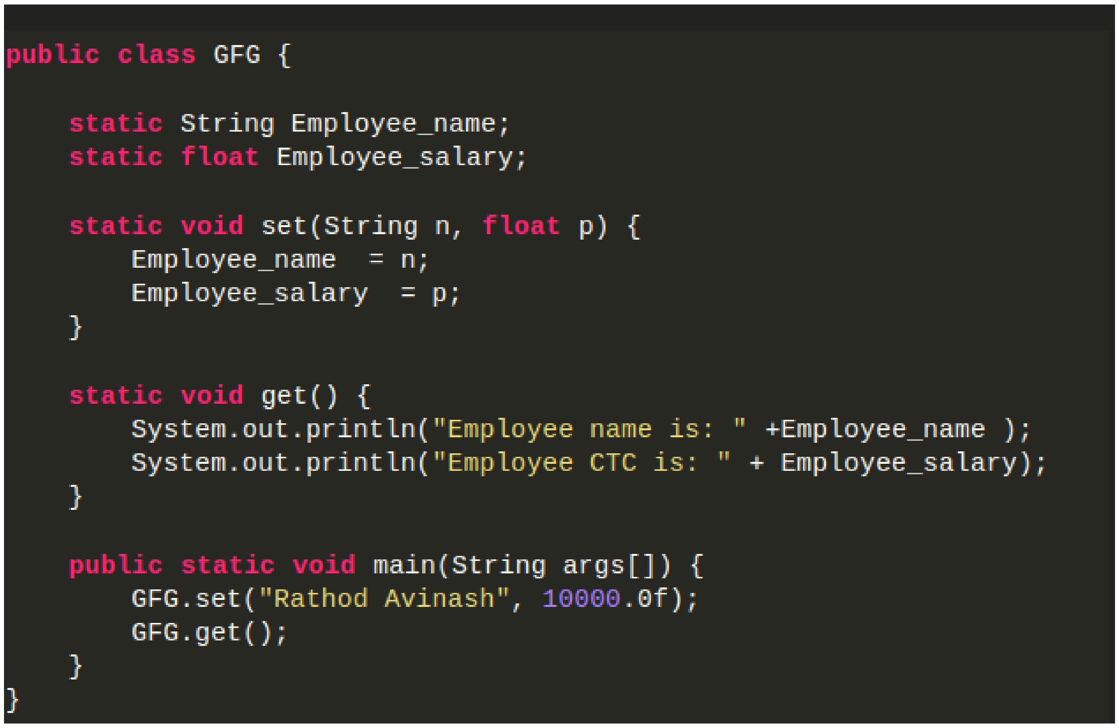
Java contains a provision for automatic garbage collection, ie, the Java runtime is responsible for automatically freeing up memory that is no longer needed. One there are no references to the memory address, the memory becomes eligible to be freed by the garbage collector. This doesn't however completely prevent ***memory leaks*** from happening. For example, if there is a memory address being addressed, but is not used, such a condition is called a ***logical memory leak*** and this cannot be automatically corrected by the Java environment. However the garbage collector in Java helps solve some of the common memory leaks in dynamic programming.

Java's syntax is heavily influenced by C++ and C. Java differs from C++ in that it primarily focuses on object-oriented programming, while C++ mixes the syntax for structured, generic, and object-oriented programming. The Code is exclusively written within classes, where each data item is treated as an object, except for primitive data types like as integers, floating-point numbers, boolean values, and characters. These primitive kinds are not treated as objects due to performance considerations. Java incorporates certain widely used features from C++ (such as the printf function).

Java lacks support for operator overloading and does not allow multiple inheritance for classes, although it does provide multiple inheritance for interfaces.

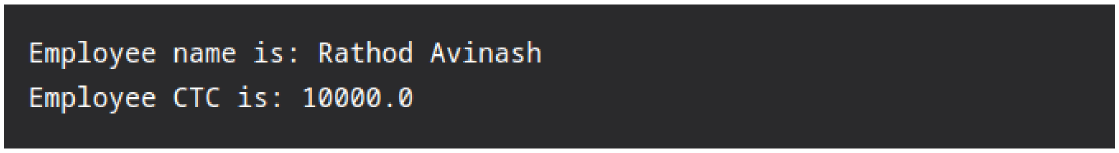
OOPs features in Java

Figure 9 shown below illustrates a simple Java code to create a class and object. This is the general syntax for the creation of a class in Java. Figure 10 shows the output obtained.



```
public class GFG {

    static String Employee_name;
    static float Employee_salary;

    static void set(String n, float p) {
        Employee_name  = n;
        Employee_salary  = p;
    }

    static void get() {
        System.out.println("Employee name is: " +Employee_name );
        System.out.println("Employee CTC is: " + Employee_salary);
    }

    public static void main(String args[]) {
        GFG.set("Rathod Avinash", 10000.0f);
        GFG.get();
    }
}
```

Figure 9: A simple class definition in Java

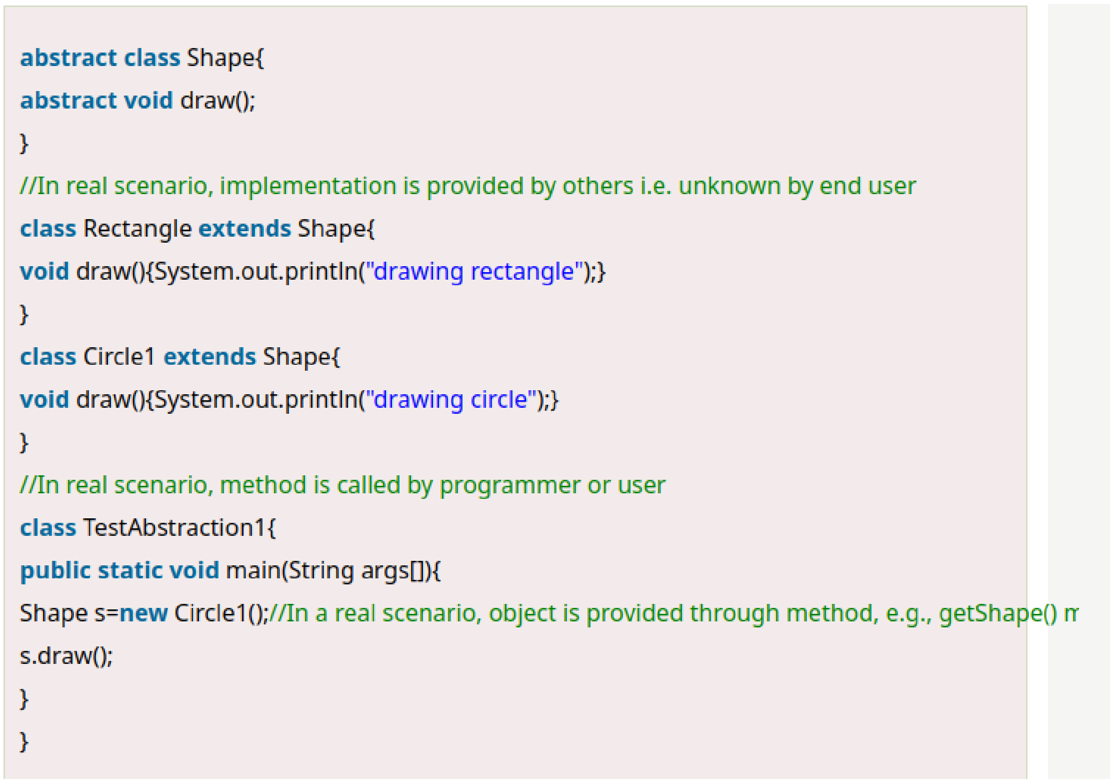


```
Employee name is: Rathod Avinash
Employee CTC is: 10000.0
```

Figure 10: Output for the class instance

- ***Abstractionin Java***: In the context of Java programming, abstraction refers to the concept where basic elements such as objects, classes, and variables are used to encapsulate and represent intricate underlying code and data. This is significant as it enables us to circumvent the repetition of identical tasks on several occasions.

  In Java, abstraction can be done with an entire class. This is done using the ***abstract*** keyword. It can have both abstract and non-abstract methods. It cannot be instantiated (an instance of that class cannot be created). It can contain ***final methods*** to prevent subclass from overwriting the method. ***Abstract methods*** in Java, is such a method that does not have an implementation. AN example of an abstract class containing an abstract method is illustrated in Figure 11.

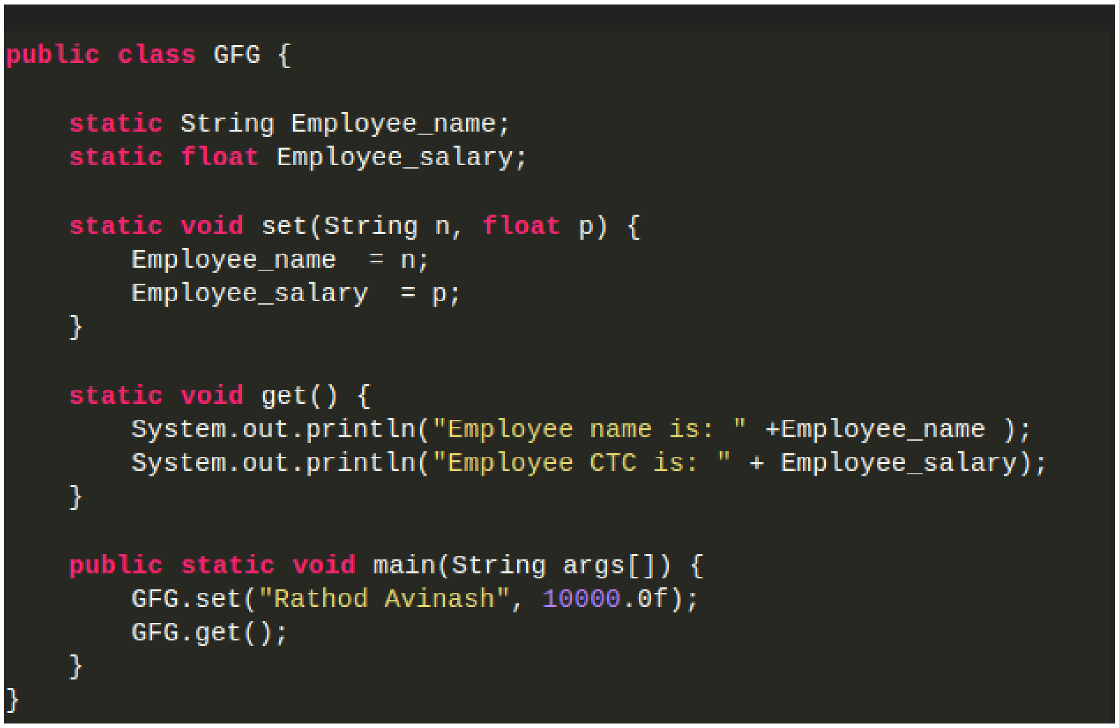


```
abstract class Shape{
abstract void draw();
}
//In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}
//In real scenario, method is called by programmer or user
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape() m
s.draw();
}
}
```

Figure 11: Example of an abstract class and abstract method

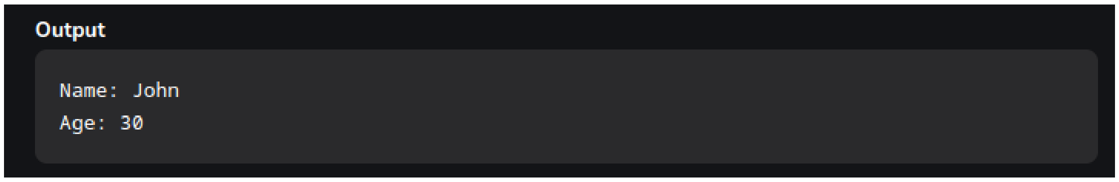
- ***Encapsulation in Java***: Encapsulation in Java is a core principle of object-oriented programming (OOP) that involves combining data and the corresponding methods into a single entity known as a class. Java Encapsulation is a mechanism that conceals the internal workings of a class from external access, allowing only a public interface to be exposed for interacting with the class.

  Encapsulation in Java is achieved by declaring the instance variables of a class as ***private***. This prevents access to these variables outside of the class, thereby concealing the internal functioning of the class. Java also has provision for outside access of these variables, with the help of public methods like ***getters*** and ***setters***, which are used for retrieving and modifying instance variables. The use of these methods allows the class to enforce its own validation rules and ensures the internal consistency of the class. Figure 12 and 13 illustrate a simple example of how encapsulation functions in Java using get and set method along with the output obtained.



```java
public class GFG {

    static String Employee_name;
    static float Employee_salary;

    static void set(String n, float p) {
        Employee_name  = n;
        Employee_salary  = p;
    }

    static void get() {
        System.out.println("Employee name is: " +Employee_name );
        System.out.println("Employee CTC is: " + Employee_salary);
    }

    public static void main(String args[]) {
        GFG.set("Rathod Avinash", 10000.0f);
        GFG.get();
    }
}
```

Figure 12: Encapsulation in Java

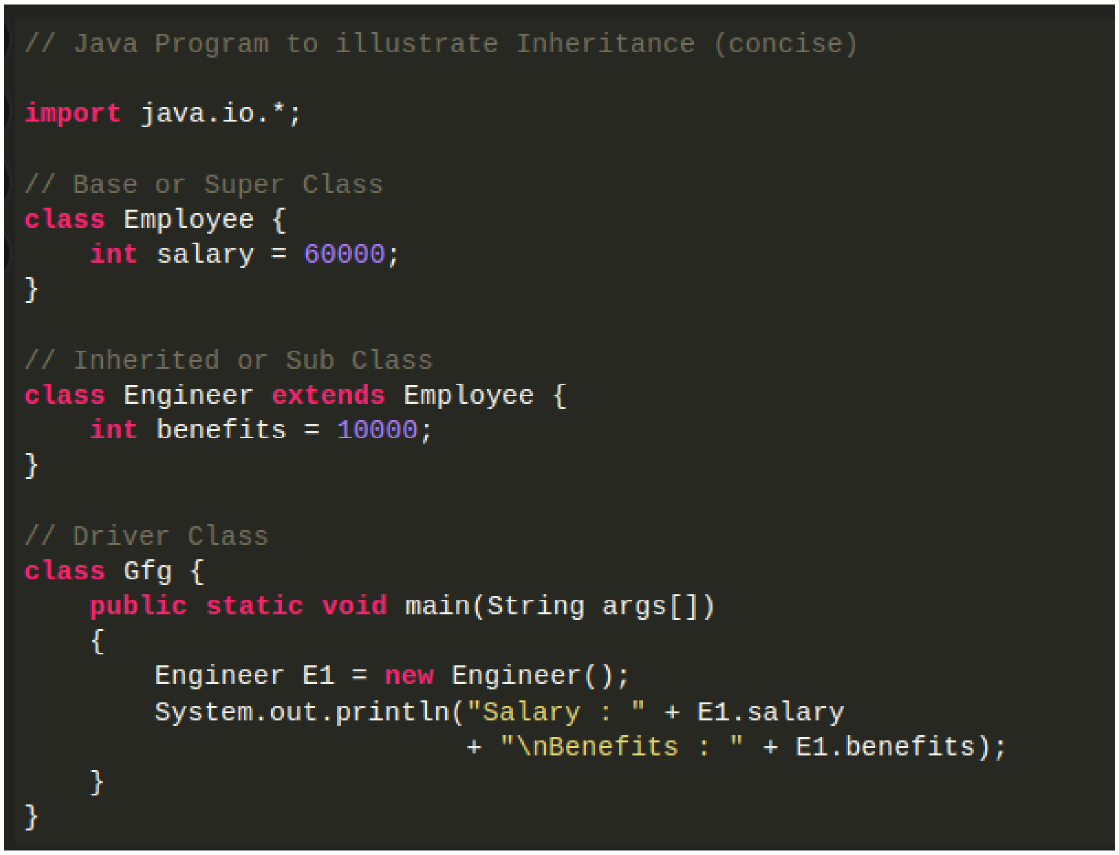


Output

```
Name: John
Age: 30
```

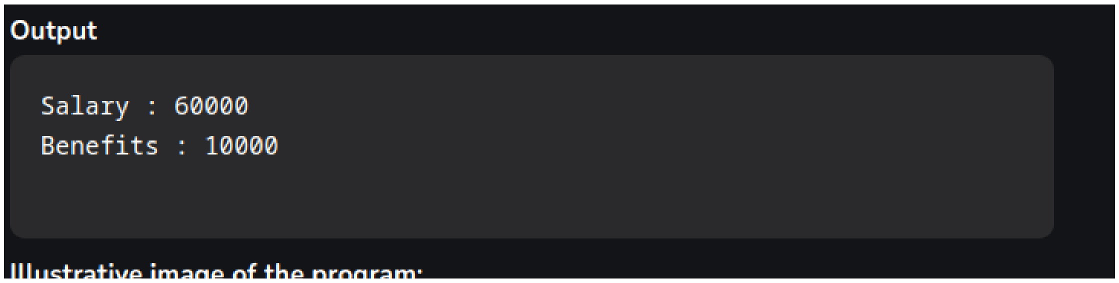
Figure 13: Output of encapsulation

- ***Inheritance in Java***: This is a distinctive characteristic of Object Oriented Programming in Java. programmers can utilise it to generate new classes that inherit certain attributes from preexisting classes. This enables us to leverage prior efforts without duplicating the class. Figure 14 illustrates an example to demonstrate the use of inheritance in Java. Figure 15 illustrates the output.



```java
// Java Program to illustrate Inheritance (concise)

import java.io.*;

// Base or Super Class
class Employee {
    int salary = 60000;
}

// Inherited or Sub Class
class Engineer extends Employee {
    int benefits = 10000;
}

// Driver Class
class Gfg {
    public static void main(String args[])
    {
        Engineer E1 = new Engineer();
        System.out.println("Salary : " + E1.salary
                           + "\nBenefits : " + E1.benefits);
    }
}
```

Figure 14: Inheritance in Java



Output

```
Salary : 60000
Benefits : 10000
```

Illustrative image of the program:

Figure 15: Output for inheritance in Java

There are multiple types of inheritance possible in Java. They are:

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance
- Hybrid Inheritance

Single inheritance: in Java, subclasses inherit the features of one superclass. Figure 16 below illustrates a single inheritance.

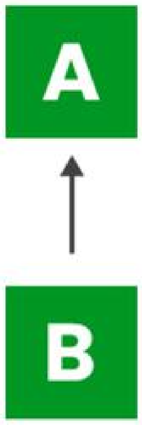


Figure 16: Single inheritance in Java

Multilevel Inheritance: In Multilevel Inheritance, a derived class inherits from a base class, and the derived class (intermediatory class) itself serves as the base class for further derived classes. Figure 17 illustrates the functioning of multilevel inheritance.

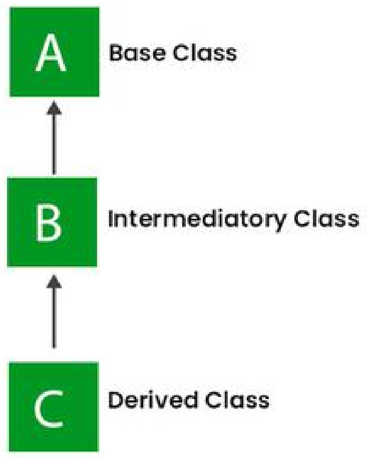


Figure 17: Multilevel inheritance in Java

Hierarchical Inheritance: Hierarchical Inheritance involves a superclass (base class) that is shared by multiple subclasses. Figure 18 shown below illustrates that class A acts as a superclass for the derived classes B, C, and D.

Multiple Inheritance (Through Interfaces): In multiple inheritance, a

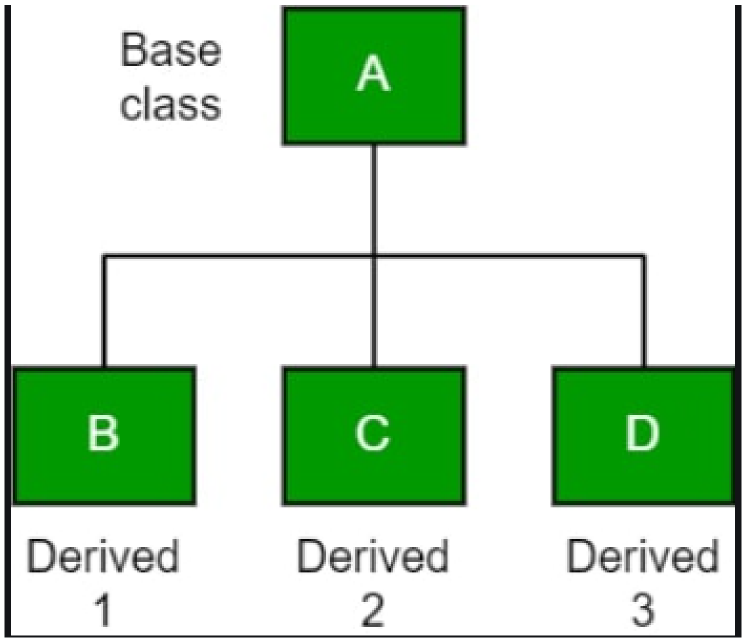


Figure 18: Hierarchical inheritance in Java

class can have more than one superclass, and it can get traits from all of its parents. Java does not allow classes to receive from more than one parent. The only way to have multiple inheritances in Java is to use Interfaces. Figure 19 shown below shows how connections A and B lead to Class C.

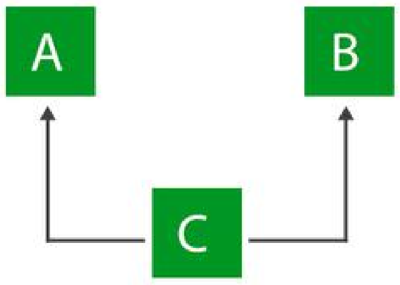


Figure 19: Multiple inheritance in Java

Hybrid Inheritance It's a mix of at least two of the other types here. Multiple inheritance is not possible with classes in Java, so mixed inheritance that uses multiple inheritance is also not possible with classes. If we want to use multiple inheritance to create hybrid inheritance in Java, we can only do it through interfaces. Figure 20 shows how hybrid Inheritance occurs.

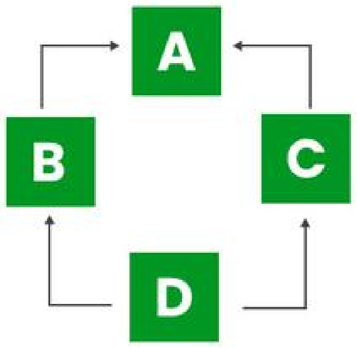


Figure 20: Hybrid Inheritance in Java

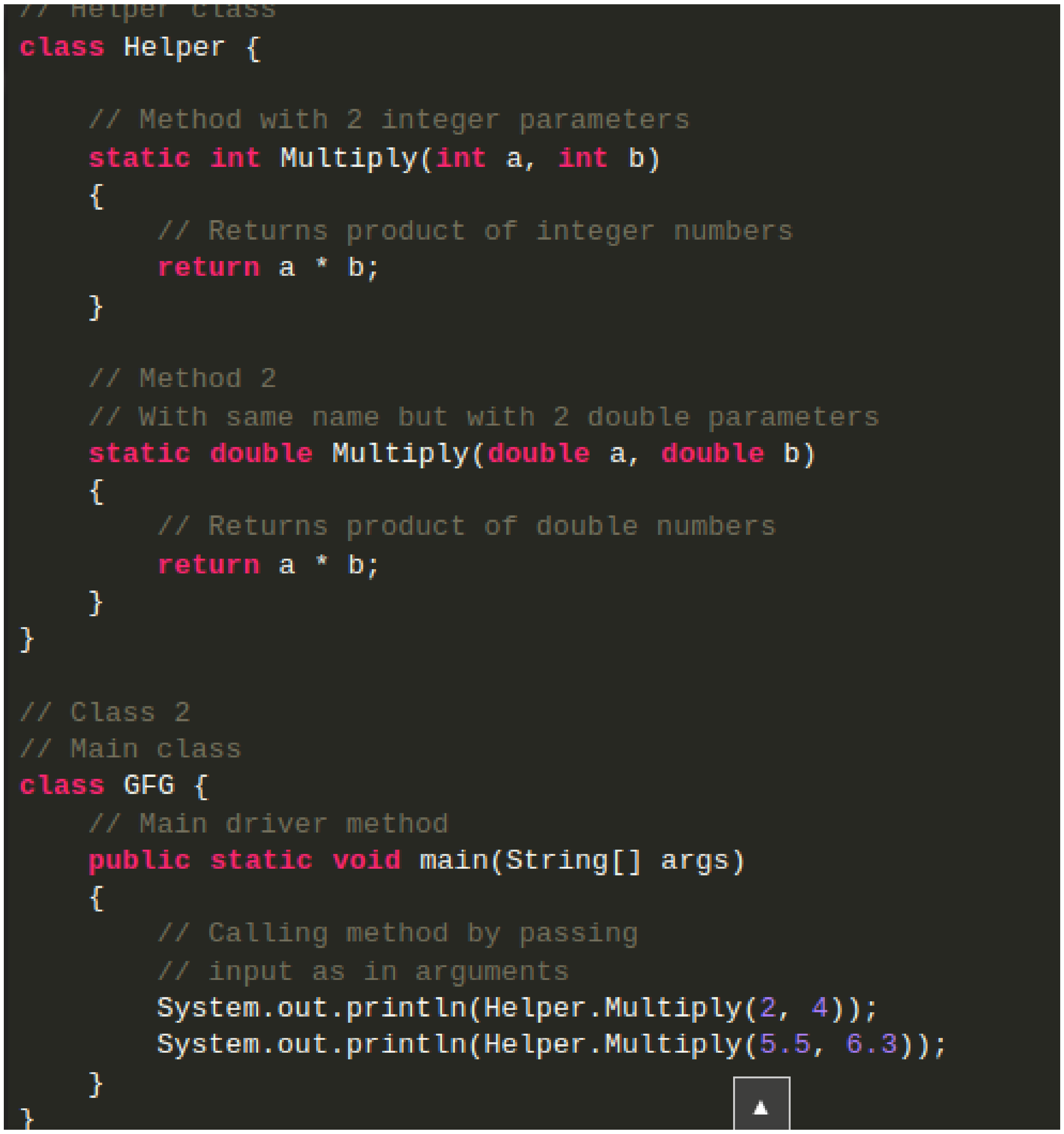
- ***Dynamic method dispatch in Java***: Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. When a superclass reference is used to call a modified method, Java figures out which version (superclass or subclasses) of that method to run based on the type of the object being asked about at the time of the call. In other words, this choice is made during run time. No matter what type of reference variable it is, the type of the object being referred to decides which version of an overridden method will be run at run time. A reference variable from a superclass can point to an object from a subclass. It's also called ***upcasting***. This is how Java handles calls to methods that have already been called at run time.

  The advantage of this is that Java allows a class to specify methods that will be common to all of its derivatives while allowing subclasses to define the specific implementation of some or all of those methods. It is, however to be noted that Only methods can be overridden in Java. variables (data members) cannot be overridden. This means that data members cannot be used for runtime flexibility.

- ***Polymorphism in Java***: Java Polymorphism can be defined as the capacity of a message to be exhibited in multiple forms. Typically, Polymorphism in Java is mainly divided into two types:

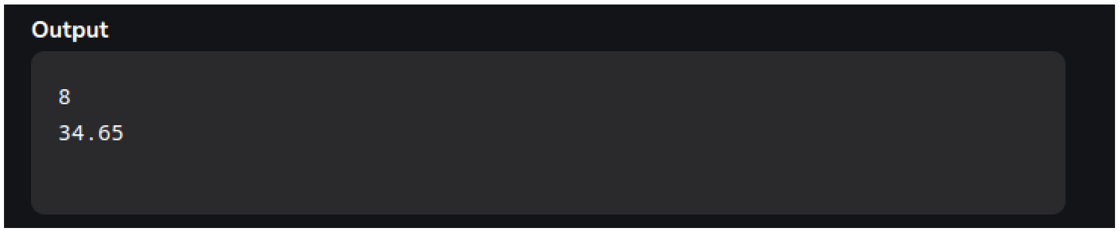
  – Compile-time polymorphism
  – run-time polymorphism

  Compile-time polymorphism typically achieved by function overloading or operator overloading. However, Java does not support Operator overloading. ***Method overloading*** occus when there are different functions with the same name but different parameters. Functions can typically be overloaded by modifying the number or type of arguments. Figure 21 illustrates a simple example to demonstrate method overloading. Figure 22 displays the output of the code snippet.



```
// Helper class
class Helper {

    // Method with 2 integer parameters
    static int Multiply(int a, int b)
    {
        // Returns product of integer numbers
        return a * b;
    }

    // Method 2
    // With same name but with 2 double parameters
    static double Multiply(double a, double b)
    {
        // Returns product of double numbers
        return a * b;
    }
}

// Class 2
// Main class
class GFG {
    // Main driver method
    public static void main(String[] args)
    {
        // Calling method by passing
        // input as in arguments
        System.out.println(Helper.Multiply(2, 4));
        System.out.println(Helper.Multiply(5.5, 6.3));
    }
}
```

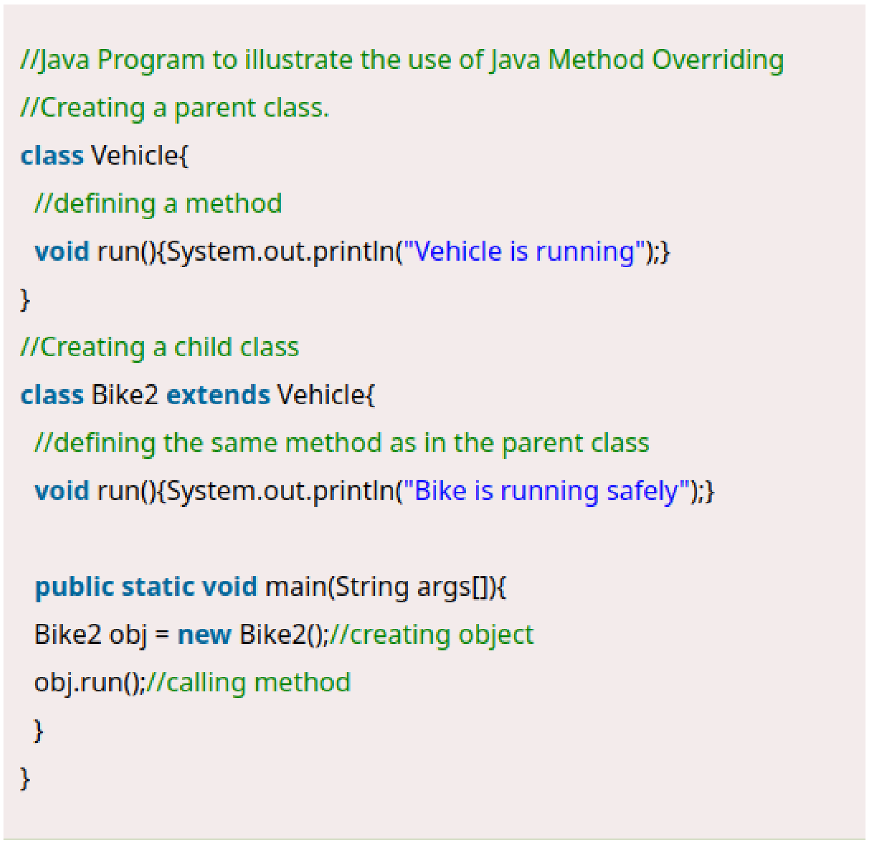
Figure 21: Function Overloading in Java



```
Output

8
34.65
```

Figure 22: Output for Function Overloading

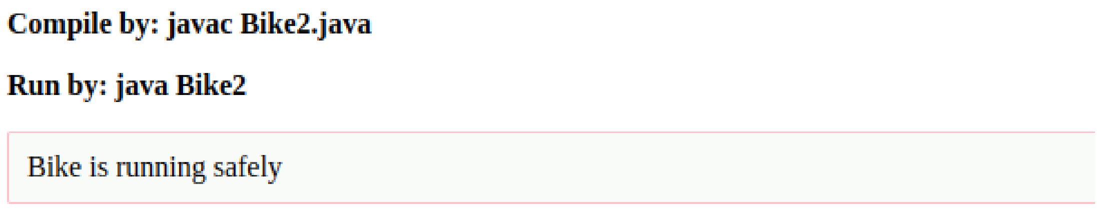
Run-time polymorphism: Runtime polymorphism is a procedure where a function call to an overridden method is resolved during program execution. Method Overriding is the means by which this form of polymorphism is accomplished. Method overriding refers to the situation where a derived class provides its own implementation for a member function that is already defined in the base class. The basic function is reported to be overridden. Figure 23 illustrates a simple code to demonstrate function overriding.



```java
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle{
 //defining a method
 void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
 //defining the same method as in the parent class
 void run(){System.out.println("Bike is running safely");}

 public static void main(String args[]){
 Bike2 obj = new Bike2();//creating object
 obj.run();//calling method
 }
}
```

Figure 23: Function Overriding in Java

Virtual functions in Java enables an instance of a subclass to function as if it were an instance of the superclass. The derived class has the ability to



Compile by: javac Bike2.java

Run by: java Bike2

```
Bike is running safely
```

Figure 24: Output for Function Overriding in Java

replace the virtual function of the base class with its own implementation. The function call is dynamically resolved at runtime, based on the specific type of the object.

Paradigm 2: Logic Programming

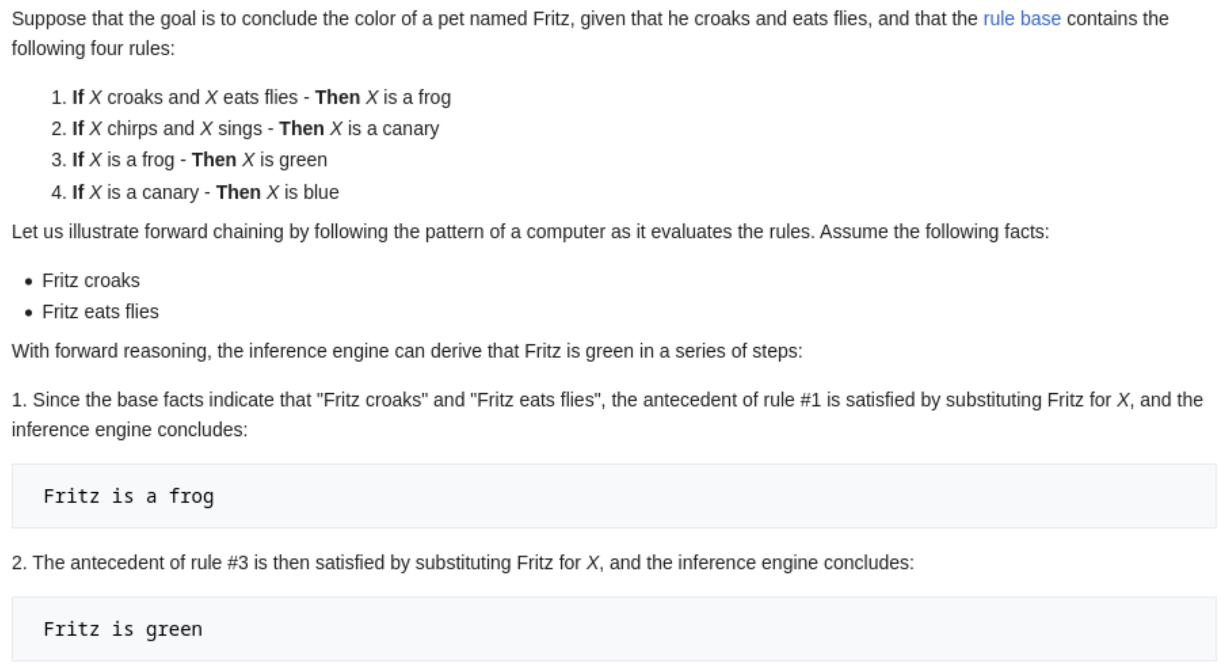
Logic programming is a computational paradigm that relies on formal logic to program, store and represent knowledge. A logic program consists of a collection of phrases expressed in logical form, which serve to represent knowledge pertaining to a specific problem domain. Computation involves the application of logical reasoning to utilise information and solve problems within a certain subject. Major logic programming language families include Prolog, Answer Set Programming (ASP) and Datalog.

In ASP and Datalog, logic programs are interpreted in a purely declarative manner. Their execution is carried out by a proof method or model generator, which is not intended to be controlled by the programmer. However, within the prolog family of languages, logic programs can also be interpreted procedurally as goal-reduction procedures: **to solve A, solve B1, and ... solve Bn.** A significant portion of the study in the domain of logic programming has focused on the attempt to provide a logical framework for negation as failure, as well as on the exploration of alternative semantics and implementations for negation. These advancements have played a crucial role in facilitating the progress of formal methods for verifying logic-based programs and transforming programs.

Concepts

The two main problem solving strategies are ***forward reasoning*** and ***backward reasoning***.

Forward reasoning is a problem solving strategy which employs the use of initial facts and data to arrive at the final solution. It includes the use of original facts to derive additional facts using ***inference rules*** until a goal is reached. An inference engine keeps searching the inference rules until one of the rules satisfy the ***if condition***(***antecedent***) is satisfied. An antecedent is the first half of a hypothetical proposition, whenever the if-clause precedes the then-clause. For example, if P, then Q, for this statement, P is considered the antecedant and Q is considered the ***Consequent***. When the search engine arrives at a inference rule in which the antecedant is satisfied, it is considered to have derived a new fact from that inference rule. Figure 25 shown below illustrates in detail, an example of how a conclusion is reached by using initial facts given.



Suppose that the goal is to conclude the color of a pet named Fritz, given that he croaks and eats flies, and that the rule base contains the following four rules:

1. **If** *X* croaks and *X* eats flies - **Then** *X* is a frog
2. **If** *X* chirps and *X* sings - **Then** *X* is a canary
3. **If** *X* is a frog - **Then** *X* is green
4. **If** *X* is a canary - **Then** *X* is blue

Let us illustrate forward chaining by following the pattern of a computer as it evaluates the rules. Assume the following facts:

- Fritz croaks
- Fritz eats flies

With forward reasoning, the inference engine can derive that Fritz is green in a series of steps:

1. Since the base facts indicate that "Fritz croaks" and "Fritz eats flies", the antecedent of rule #1 is satisfied by substituting Fritz for *X*, and the inference engine concludes:

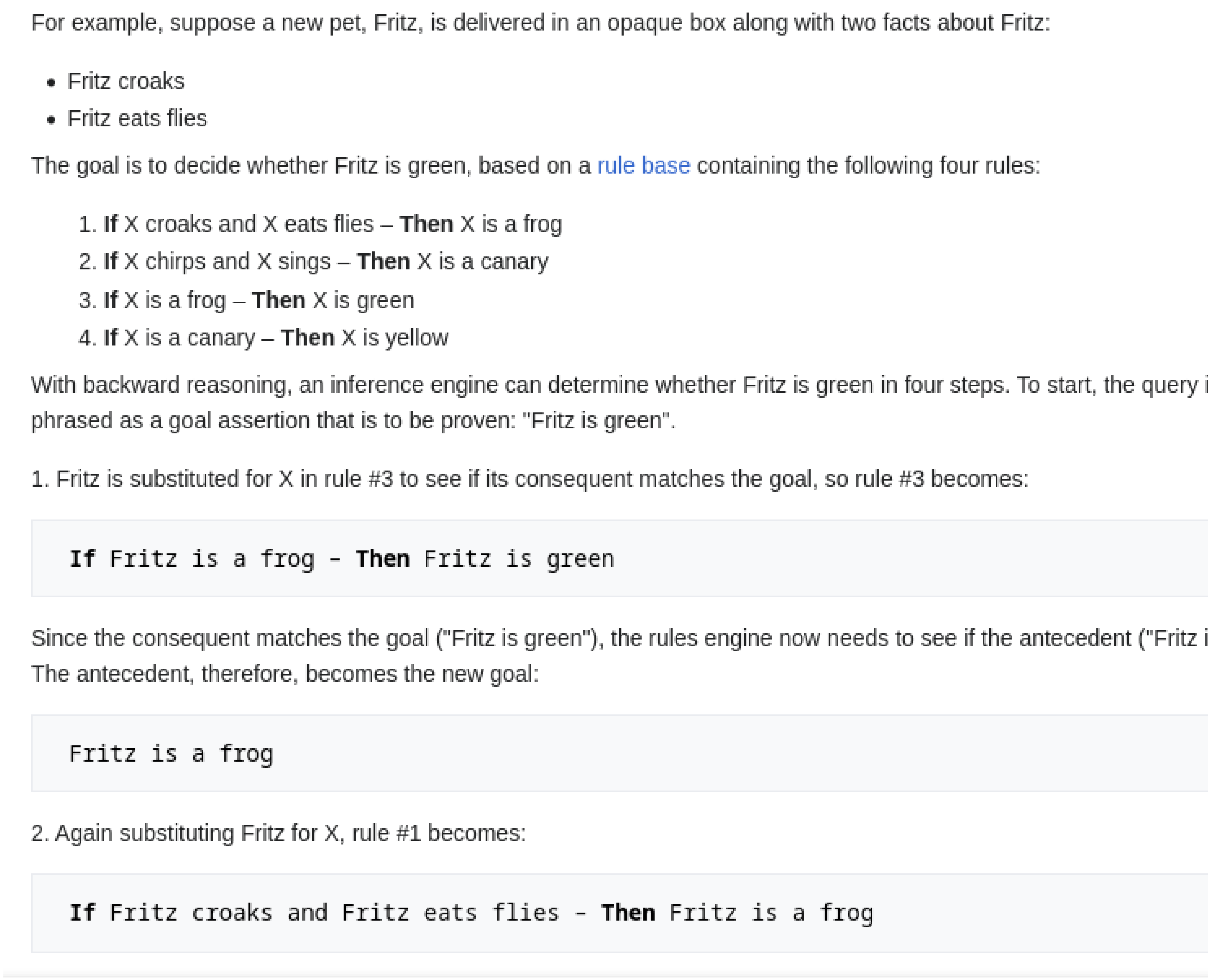
```
Fritz is a frog
```

2. The antecedent of rule #3 is then satisfied by substituting Fritz for *X*, and the inference engine concludes:

```
Fritz is green
```

Figure 25: Forward-Reasoning in Logical Programming

Backward reasoning is the opposite of forward reasoning, where the goal is studied to determine the rules, initial facts, and data. In this method of problem solving, The goal state and rules are chosen such that the goal state is located in the THEN portion, serving as the conclusion. From the IF part of the selected rule the sub-goals are made to be satisfied for the goal state to be true. It is essential to ensure that all the initial conditions satisfy the subgoals. The engine then checks whether the provided initial state matches with the established states. If it is satisfied, the goal is the solution required. Figure 26 and Figure 27 illustrate in detail, the functionality of backward reasoning.



For example, suppose a new pet, Fritz, is delivered in an opaque box along with two facts about Fritz:

- Fritz croaks
- Fritz eats flies

The goal is to decide whether Fritz is green, based on a rule base containing the following four rules:

1. **If** X croaks and X eats flies – **Then** X is a frog
2. **If** X chirps and X sings – **Then** X is a canary
3. **If** X is a frog – **Then** X is green
4. **If** X is a canary – **Then** X is yellow

With backward reasoning, an inference engine can determine whether Fritz is green in four steps. To start, the query i
phrased as a goal assertion that is to be proven: "Fritz is green".

1. Fritz is substituted for X in rule #3 to see if its consequent matches the goal, so rule #3 becomes:

```
If Fritz is a frog - Then Fritz is green
```

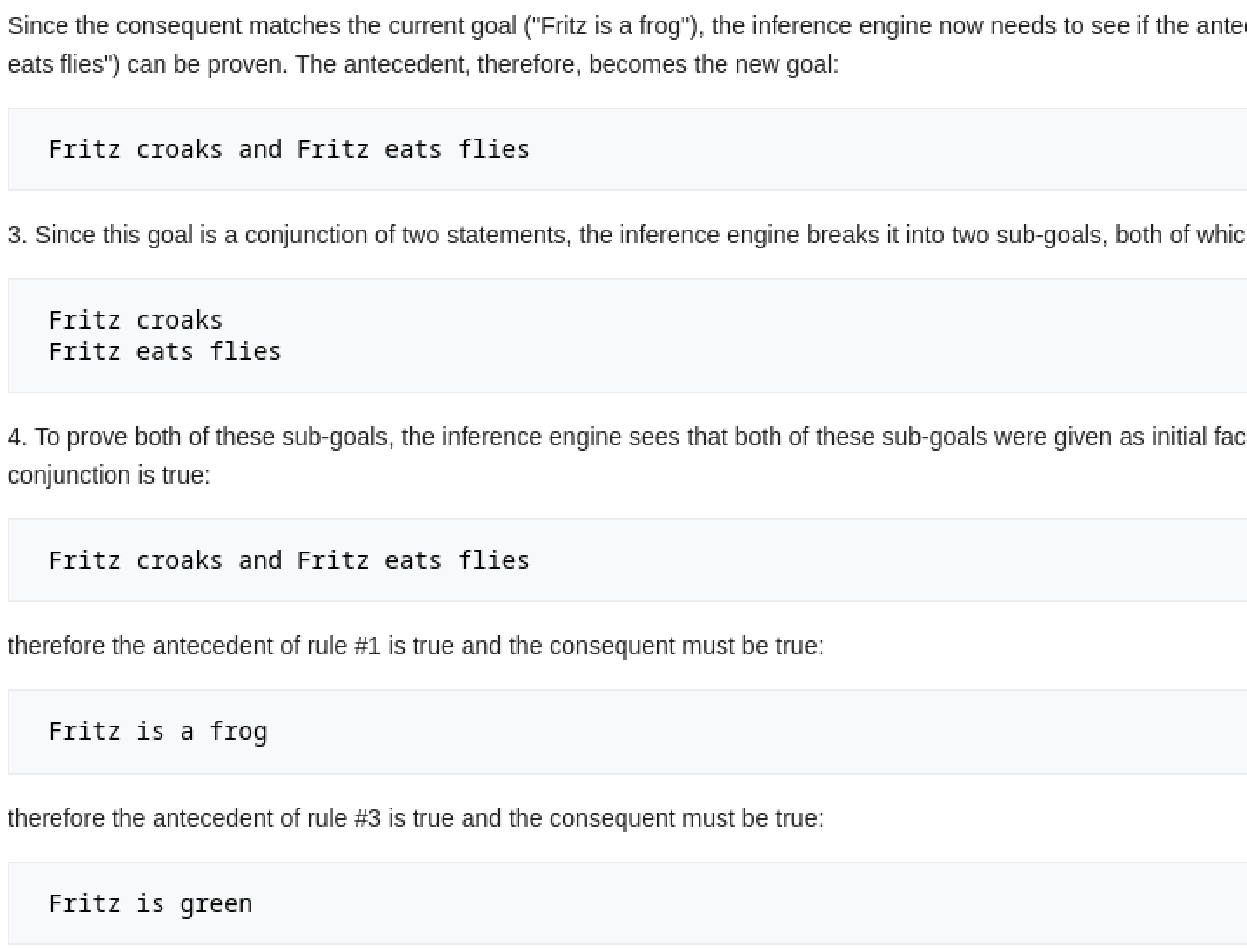
Since the consequent matches the goal ("Fritz is green"), the rules engine now needs to see if the antecedent ("Fritz i
The antecedent, therefore, becomes the new goal:

```
Fritz is a frog
```

2. Again substituting Fritz for X, rule #1 becomes:

```
If Fritz croaks and Fritz eats flies - Then Fritz is a frog
```

Figure 26: Backward reasoning in Logic Programming


Since the consequent matches the current goal ("Fritz is a frog"), the inference engine now needs to see if the ante
eats flies") can be proven. The antecedent, therefore, becomes the new goal:

```
Fritz croaks and Fritz eats flies
```

3. Since this goal is a conjunction of two statements, the inference engine breaks it into two sub-goals, both of whic

```
Fritz croaks
Fritz eats flies
```

4. To prove both of these sub-goals, the inference engine sees that both of these sub-goals were given as initial fac
conjunction is true:

```
Fritz croaks and Fritz eats flies
```

therefore the antecedent of rule #1 is true and the consequent must be true:

```
Fritz is a frog
```

therefore the antecedent of rule #3 is true and the consequent must be true:

```
Fritz is green
```

Figure 27: Backward reasoning in Logic Programming

In most cases, backward reasoning from a query or goal is more efficient than forward reasoning. But sometimes with Datalog and Answer Set Programming, there may be no query that is separate from the set of clauses as a whole, and then generating all the facts that can be derived from the clauses is a sensible problem-solving strategy.

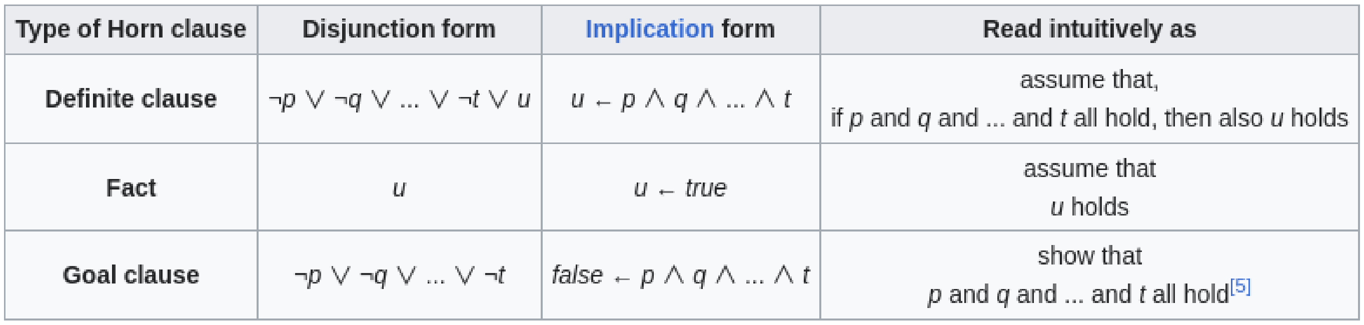
Features of Logic programming

Horn clauses: In mathematical logic and logic programming, a Horn clause is a logical formula of a particular rule-like form that gives it useful properties for use in logic programming, formal specification, universal algebra and model theory. Many of the features of Logical programming incorporate the use of

Horn clauses and thus, it is imperative to discuss the definition and types of horn clauses briefly. Figure 28 describes the commonly used horn clauses, what they denote, and how they are used.

From a purely logical point of view, there are two ways to look at Horn clause logic programs' formal meanings:

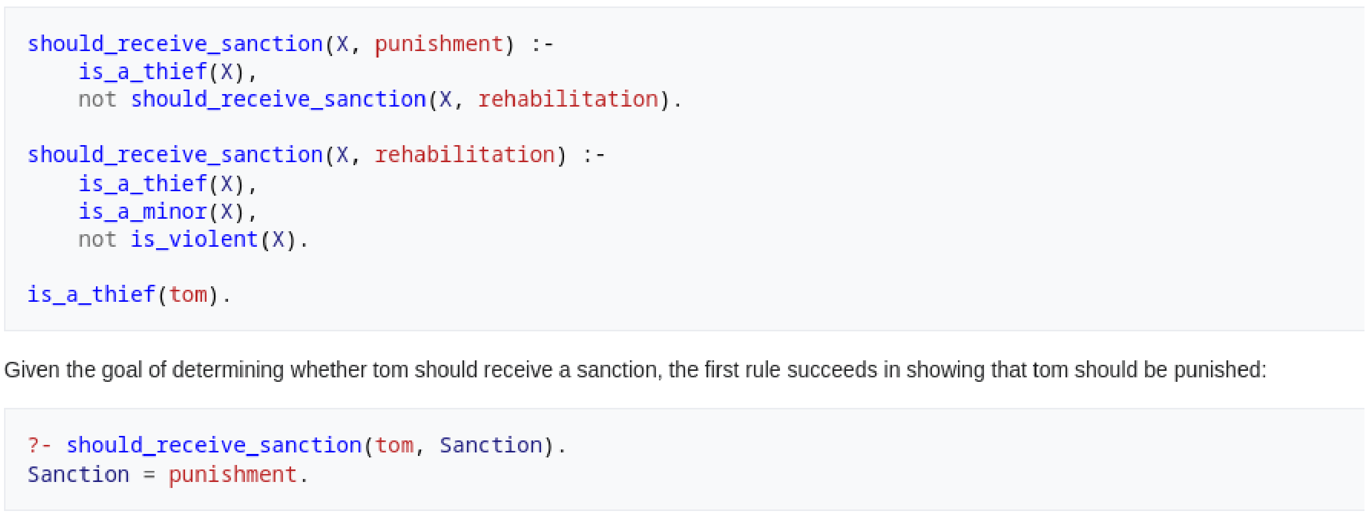
- ***The original logical consequence semantics*** is one way to look at it. It says that to solve a goal, you have to show that the goal is a theorem that is true in all models of the program.
- ***The satisfiability semantics*** is the other way to look at Horn clause program's declarative semantics. It says that to solve a goal, you have to show that the goal is true (or satisfied) in some designed (or standard) model of the program. There is always a standard model for Horn clause programs. It is the program's one and only basic model.



Type of Horn clause	Disjunction form	Implication form	Read intuitively as
Definite clause	$\neg p \vee \neg q \vee ... \vee \neg t \vee u$	$u \leftarrow p \wedge q \wedge ... \wedge t$	assume that, if *p* and *q* and ... and *t* all hold, then also *u* holds
Fact	u	$u \leftarrow true$	assume that *u* holds
Goal clause	$\neg p \vee \neg q \vee ... \vee \neg t$	$false \leftarrow p \wedge q \wedge ... \wedge t$	show that *p* and *q* and ... and *t* all hold[5]

Figure 28: Types of Horn clauses

Negation as Failure: Negation as failure, or NAF, is a logic programming rule that doesn't work for all cases. It attempts to derive $\neg p$, from failure to derive p. Negation as failure has been an important feature of logic programming since the earliest days of both Planner and Prolog. In Prolog, it is usually implemented using Prolog's extralogical constructs. The program in Figure 29 illustrates the functioning of Negation. It also illustrates the output that is received as a result of this problem solving method.



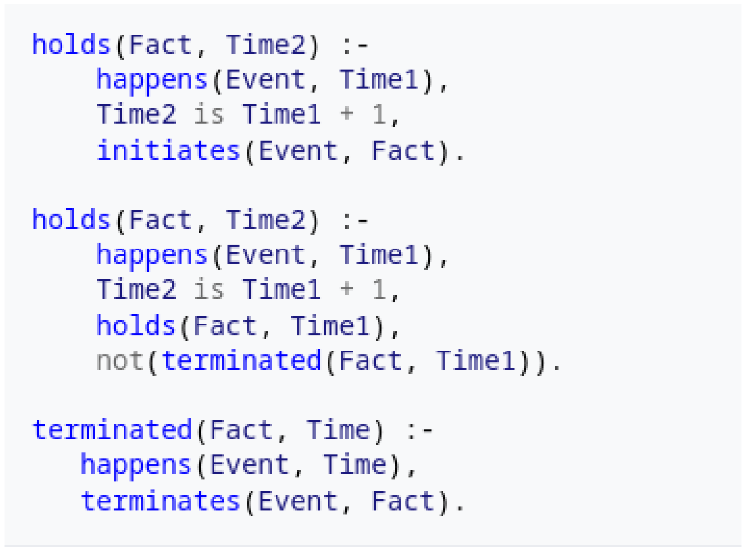
```
should_receive_sanction(X, punishment) :-
    is_a_thief(X),
    not should_receive_sanction(X, rehabilitation).

should_receive_sanction(X, rehabilitation) :-
    is_a_thief(X),
    is_a_minor(X),
    not is_violent(X).

is_a_thief(tom).
```

Given the goal of determining whether tom should receive a sanction, the first rule succeeds in showing that tom should be punished:

```
?- should_receive_sanction(tom, Sanction).
Sanction = punishment.
```

Figure 29: Negation as a failure in Logic Programming

Knowledge Representation: One of the main ideas behind the early stages of logic programming was to use logic to describe both procedural knowledge and strategic information. To this day, it is still an important part of the prolog family of logic computer languages. However, more and more logic programming applications, such as prolog applications, focus on using logic to represent information that is only declarative. People can use these applications to show both general commonsense knowledge and skill in a certain area. Figure 30 illustrates a program that illustrates the functionality of such formalisms.



```
holds(Fact, Time2) :-
    happens(Event, Time1),
    Time2 is Time1 + 1,
    initiates(Event, Fact).

holds(Fact, Time2) :-
    happens(Event, Time1),
    Time2 is Time1 + 1,
    holds(Fact, Time1),
    not(terminated(Fact, Time1)).

terminated(Fact, Time) :-
   happens(Event, Time),
   terminates(Event, Fact).
```

Figure 30: Knowledge Representation in Logic Programming

According to the code, holds is a meta-predicate, and the first argument of holds is a fact and the second argument is a time (or state). The atomic formula holds(Fact, Time) expresses that the Fact holds at the Time. Such time-varying facts are also called ***fluents***. In simple words, fluents is a condition that changes over time. The answers to the goal holds (Fact, Time) are the same when you use forward and backward thinking. On the other hand, forward reasoning creates fluents more quickly in terms of time, while backward reasoning creates fluents more slowly, like in the situation calculus, where ***regression*** is used in a very specific way. Regression, here is a mechanism for proving consequences in the situation calculus.

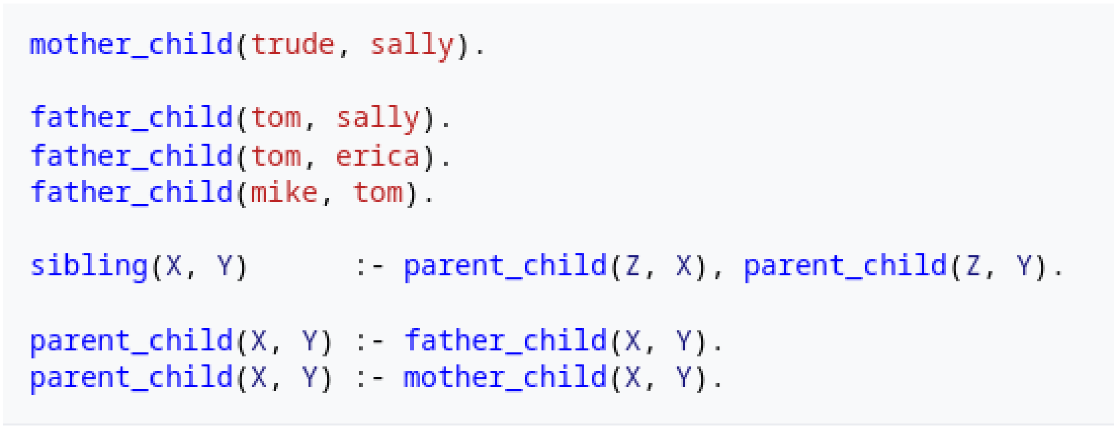
Language for Logic Programming: Prolog

Prolog is a programming language for reasoning that comes from the fields of artificial intelligence and computational linguistics. prolog comes from first-order logic, which is a formal logic. Unlike many other programming languages, prolog is mostly meant to be a descriptive programming language, which means that the program is made up of facts and rules that describe relationships. A computation starts when a question is run through the program. It consists of the following Data Types:

- An ***atom*** which is a symbol name starting with a lower case letter or guarded by quotes.
- Numbers can be ***floats*** or ***integers***.
- ***Variables*** are denoted by a string consisting of letters, numbers and underscore characters, and beginning with an upper-case letter or underscore.
- A ***compound term*** is composed of an atom called a "functor" and a number of "arguments", which are again terms. To further elaborate,
  - A List is an ordered collection of terms. It is denoted by square brackets with the terms separated by commas, or in the case of the empty list, by []. For example, [1,2,3,4] or [red,green,blue].
  - If you put quotes around a string of characters, that string can be either a list of numeric character codes, a list of characters (atoms of length 1), or an atom, based on the value of the prolog flag double quotes.

Clauses in prolog programs outline relations that the programs talk about. Pure prolog can only use ***Horn clauses***. There are two kinds of Horn clauses that are used to describe prolog programs: ***rules and facts***. A rule looks like this:

Body: Head. It means that Head is true if Body is true. Calls to predicates make up the body of a rule. These are called the rule's goals. Clauses with empty bodies are called facts. An example of a fact is: human(socrates). which is equivalent to the rule: human(socrates) :- true. ***Predicates in Prolog***: A process definition or predicate is a group of clauses whose heads all have the same name and level of detail. A logic program is a set of predicates. The prolog code shown in Figure 31 illustrates a prolog program comprising of four predicates.



```
mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

sibling(X, Y)      :- parent_child(Z, X), parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
```

Figure 31: Predicates in Prolog

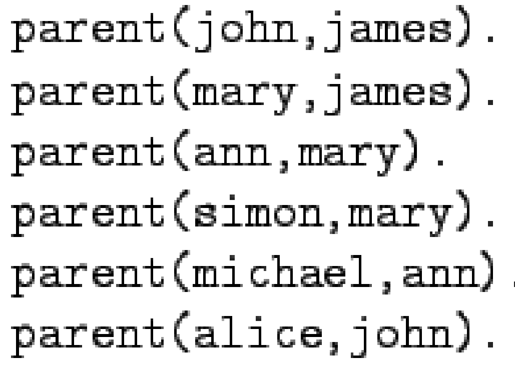
Negation as a failure : In pure prolog, NAF literals like $\neg P$ can be found in clause bodies and can be used to get other NAF literals. In Figure 32, the Four clauses can be used to derive the negation of them as well.

$$p \leftarrow q \wedge \text{not } r$$
$$q \leftarrow s$$
$$q \leftarrow t$$
$$t \leftarrow$$

Figure 32: Negation in Prolog

From Figure 32, prolog can derive $\neg S$, $\neg r$, $\neg p$, $\neg q$, $\neg t$ as well.

Knowledge Representation in Prolog: In prolog, predicates stand for knowledge. As in the Relational Model, a predicate is a structure that works like a connection. An example of family relationships is given in Figure 33. There are seven simple clauses of predicate parent/2. The first clause denotes that james is john's parent, the second one denotes that james is mary's parent, and so on. A complex clause talks about how pieces of knowledge are related,



```
parent(john,james).
parent(mary,james).
parent(ann,mary).
parent(simon,mary).
parent(michael,ann).
parent(alice,john).
```

Figure 33: Knowledge representation in Prolog

which is what it means to provide Intentional Knowledge. This kind of phrase is a structure or group of structures that make up a logical word. There is a head and a body to it. This is what the head says. The body has conditions already in place. The predicates in these structures are true, and they are set up by simple or complex clauses. Built-in predicates, which can do things like reasoning and math, could also be used with complex clauses.

Analysis

OOPs-Review

The following were identified as the benefits of using Object-Oriented Programming paradigm generally:

- By utilising common working modules that intercommunicate, we can construct programs without the need to write code from the beginning, resulting in time savings and increased productivity.
- OOP language allows to break the program into the bit-sized problems that can be solved easily (one object at a time).

- It is possible that multiple instances of objects co-exist without any interference,
- It is very easy to partition the work in a project based on objects.
- The principle of data hiding helps the programmer to build secure programs which cannot be invaded by the code in other parts of the program.
- By using inheritance, we can eliminate redundant code and extend the use of existing classes.

From various sources, These were found to be the drawbacks of using OOPs :

- Although OOP is a simple idea, it can take writers a while to get good at it. For newcomers, the ideas of inheritance, polymorphism, and packaging can be hard to grasp.
- It's possible for OOP code to be longer and more complicated than code written in other paradigms, which could make it run slower. More memory and processing power are also often needed for OOP than for other models.
- OOP can lead to complex code, especially when dealing with large systems that have many interdependent objects. This complexity can make it more difficult to debug and maintain code.
- Although OOP allows for code reusability, it can also lead to tightly coupled code that is difficult to reuse in other contexts. This can make it challenging to maintain the codebase in the long run.

Oops in Java-A review

The following were found to be the advantages of using the various aspects of OOPs in Java:

- ***Encapsulation***
  - Programming is made flexible with encapsulation.
  - With encapsulation, it's simple and easy to debug an application.
  - It is also possible to change and improve your script without stopping your program from running normally.
  - It enables the programmer to monitor the data accessibility of a class.

- ***Inheritance***
  - Inheritance allows for code reuse and reduces the amount of code that needs to be written. The subclass can reuse the properties and methods of the superclass, reducing duplication of code.
  - Inheritance allows for the creation of abstract classes that define a common interface for a group of related classes. This promotes abstraction and encapsulation, making the code easier to maintain and extend.
  - Inheritance allows for polymorphism, which is the ability of an object to take on multiple forms. Subclasses can override the methods of the superclass, which allows them to change their behavior in different ways.
- ***Abstraction***
  - It reduces the complexity of viewing things.
  - Avoids code duplication and increases reusability.
  - It improves the maintainability of the application.
  - Hides implementation details and exposes only relevant information.
  - Supports modularity, as complex systems can be divided into smaller and more manageable parts.
  - Provides a clear and simple interface to the user.
- ***Dynamic Dispatch***
  - Dynamic method dispatch allows Java to support overriding of methods which is central for run-time polymorphism.
  - It also lets subclasses add their own methods, which lets subclasses decide how some things should be implemented.
  - It allows a class to specify methods that will be common to all of its derivatives while allowing subclasses to define the specific implementation of some or all of those methods.
- ***Polymorphism***
  - Increases code reusability by allowing objects of different classes to be treated as objects of a common class.
  - Improves readability and maintainability of code by reducing the amount of code that needs to be written and maintained.

  - Supports dynamic binding, enabling the correct method to be called at runtime, based on the actual class of the object.

The drawback of using various OOPS concepts in Java were found as follows:

- ***Encapsulation***
  - Can lead to increased complexity, especially if not used properly.
  - Can complicate comprehension of the system's functioning.
  - May limit the flexibility of the implementation.
- ***Inheritance***:
  - Multiple inheritance is not supported in classes, and thus needs to be achieved through the use of interfaces. The same holds true for hybrid inheritance as well.
  - Complexity: Inheritance can make the code more complex and harder to understand. This is especially true if the inheritance hierarchy is deep or if multiple inheritances are used.
  - Tight Coupling: Inheritance creates a tight coupling between the superclass and subclass, making it difficult to make changes to the superclass without affecting the subclass.
- ***Dynamic Dispatch***:
  - It adds some performance overhead to method calls. The JVM(Java Virtual Machine) has to determine the actual class of an object at runtime before it can call the appropriate method.
  - It can cause unintended consequences if a subclass overrides a method in a superclass that was not intended to be overridden.
  - It can also introduce security issues if malicious code can access objects of unexpected types or override methods in unexpected ways.
  - It can add complexity to code.
- ***Polymorphism***
  - Can make it more difficult to understand the behaviour of an object, especially if the code is complex.
  - This may lead to performance issues, as polymorphic behaviour may require additional computations at runtime.

Logic-review

The advantages of the Logic paradigm are discussed below:

- Logic programming demonstrates that verifying the correctness of a program is straightforward, as the system itself handles the problem with minimal programming steps.
- It is very useful for representing knowledge. Logical relationships can easily be transferred into facts and rules for use in a logic program.
- It can be used to represent very complicated ideas and rapidly refine an existing data model.
- Users do not have to be experts in traditional programming to use it. They only have to understand the logical domain and know how to add the predicates.
- It allows data to be presented in several different ways.
- It is efficient in terms of memory management and data storage.
- It is very good at pattern matching.

However, the Logic paradigm also has a few drawbacks which are discussed below:

- There is no suitable method of representing computational concepts originating in a built-in mechanism of state variables like it is found in conventional languages.
- Logic programming is constrained to specific categories of issues for resolution. The program's execution may exhibit sluggishness, and employing an unorthodox approach, such as utilising a binary decision, does not suffice to resolve all the issues.
- The poor facilities to support arithmetic negative effect on programmers. programmers always prefer the visible operational behaviour of machines and the dynamic control of moving parts.

Logic Programming in Prolog

The benefits of prolog are discussed below:

- Pattern matching is easy. Search is recursion-based.
- It has built-in list handling. Makes it easier to play with any algorithm involving lists.
- Prolog excels in symbolic reasoning, making it highly suitable for applications in artificial intelligence, natural language processing, expert systems, and knowledge representation.
- Prolog is commonly employed for interrogating and manipulating logical databases. The capability to articulate connections and execute intricate inquiries renders it appropriate for applications necessitating sophisticated database interactions.
- The rule-based design of Prolog enables developers to easily create rule-based systems. This is particularly advantageous in domains like expert systems, where decision-making is dependent on a predefined set of rules.
- Prolog's design, which is centred on rules, allows developers to effortlessly construct systems that rely on rules. This is especially beneficial in fields such as expert systems, where decision-making relies on a predetermined set of rules.
- The backtracking technique in Prolog allows for the systematic study of alternate solutions. This feature is advantageous in situations where a program needs the ability to explore multiple paths in order to discover a solution, therefore making it well-suited for problem-solving scenarios.

Prolog also has some drawbacks which have been discussed below:

- Prolog may not perform as well as other languages, particularly for tasks that require extensive number crunching or low-level system operations. It may not be the best choice for performance-critical applications.
- Prolog's unique declarative and logic-based paradigm can be challenging for developers accustomed to imperative or object-oriented programming.
- Compared to more popular languages, Prolog doesn't have as many libraries, frameworks, and other tools that are available.
- Multiple Prolog dialects exist, with each dialect possessing its own distinct syntax, characteristics, and oddities. The absence of standardisation

can lead to compatibility problems when transferring code between various prolog implementations.

- While Prolog is powerful for certain problem domains, it may not be the best choice for all types of applications.

Comparison

Comparision between the two paradigms

A study was conducted to highlight the key differences between the two paradigms in question and are illustrated in Table 1 shown below.

Disclaimer: some of the differences were mentioned after the analysis was done and a few other differences were obtained from Chatgpt

Table 1: Differences between OOPs and Logic programming

Object-Oriented Programming	**Logical Programming**
It follows Imperative paradigm	It follows declarative Paradigm.
It is represented using classes and objects	It is represented using Predicates and Rules
It follows a procedural execution	It follows a backtracking method of execution
Variables represent the attribute using an object	There is a unification of variables done
Inheritance is a prominent concept	Inheritance is not used
Encapsulation is a major feature	Very little Encapsulation is used.
Widely used for software development	AI,NLP, Expert systems

The study also highlighted some of the similarities shared by the two paradigms which have been highlighted below:

- prolog is declarative. In Prolog, you state your goal without stating how. Some OOP features are declarative. In object-oriented programming (OOP), you declare objects and their interactions, and the system handles the rest. Class and relationship definitions use a declarative approach.
- Logic Programming allows data abstraction through predicates and rules. As discussed before, OOPs promote data abstraction through encapsulation.
- Logic codes can be arranged into modules, providing a form of modularity. One of the important features of OOPs is its modularity property, which

is achieved through encapsulation.

- Logic Programming (Prolog): Prolog is useful for modelling relationships and solving problems, especially in artificial intelligence and expert systems. A powerful technique to model real-world items and interactions, object-oriented programming (OOP) is expressive for developing complicated systems.

Comparision between Java and Prolog

This study analysed the difference between Java and Prolog in terms of their specific use cases as well as the paradigm(which has been illustrated above ref table 1). Table 2 shows a few differences between the Languages in terms of their structure and use case. Please note that this does not highlight the difference between the languages in terms of their paradigm, since the differences between the two paradigms have already been illustrated.

Table 2: Differences between Java and Prolog

Java	**Prolog**
Java is an object-oriented language	Prolog is a Logic oriented Language
Java provides greater secure practices	Prolog gives less secure practices
It has advanced environment for web development	It excels at representing, solving problems in AI
Java focuses on software development	Prolog focuses on AI, NLP and expert systems

Challenges Faced

The challenges faced in this study are discussed in this section. One of the problems faced was the vague writeup pertaining to logical paradigm and Prolog. To elaborate, the relevance of many articles pertaining to logical paradigm were questioned. Many of them were far too analytical and seemed to be out of scope of this study. Filtering out the articles of importance proved to be one of the challenges in this report. This was addressed by basing most of the report and findings from ***Wikipedia***. The reputation regarding Wikipedia as an authentic source along with its more relevant content forced most of the references, especially regarding Logic Programming to be done from Wikipedia.

Another challenge was to find relevant similarities between Oops and Logic paradigm. Most of the writeups discussed the differences between OOPs and procedural language and functional program too, but there didn't seem to be much content on similarities between OOPs and Logic paradigms. This challenge was addressed through the use of ChatGPT.

ChatGPT was also used to obtain some additional information on the similarities between Java and Prolog. The two languages were cited to be very different and having different functionalities in many of the articles that were referred. This challenge was just addressed through the use of ChatGPT.

Much of the code was taken from Wikipedia or from ***GeekforGeeks***. Another challenge was the vast amount of information that needed to be conveyed in OOPs and Java which made the report on Logic paradigm and prolog less thorough than OOPs. However, sufficient effort was undertaken in going through articles, content, and programs to do a thorough report on the Logic paradigm as well.

Conclusion

A thorough study on the two paradigms and the features of the respective languages pertaining to the paradigm were discussed. OOPs paradigm was illustrated, its features and the advantages and disadvantages of those features. The report also illustrated how those features are implemented in Java and what the advantages and disadvantages of each feature implementation in Java. The real-time use case of Java was also discussed briefly.

Similarly, the paradigm of Logic was explored and has been illustrated in the report. The report discussed some of the core features of Logic programming and also illustrated them through examples. The advantages and disadvantages of the paradigm were discussed and the functioning of Prolog programming language was also discussed, along with their advantages and disadvantages. The real-time uses of prolog were also explored and discussed briefly.

A comparison of the two paradigms and the languages, stating their differences and similarities, their advantages and disadvantages, and their real-time use cases were discussed. A brief analysis regarding the challenges in obtaining relevant literature was also discussed, stating the open-source content in the internet pertaining to the paradigms and their languages.

References

https://en.wikipedia.org/wiki/Object-oriented_programming
https://en.wikipedia.org/wiki/Prototype-based_programming
https://en.wikipedia.org/wiki/Class-based_programming
https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)
https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)
https://en.wikipedia.org/wiki/Abstraction_(computer_science)
https://en.wikipedia.org/wiki/Polymorphism_(computer_science)
https://en.wikipedia.org/wiki/Java_(programming_language)
https://www.geeksforgeeks.org/object-oriented-programming-oops-concept-in-java/
https://tutos.fandom.com/es/wiki/The_Concepts_of_OOPS_in_java!
https://www.geeksforgeeks.org/encapsulation-in-java/?ref=lbp
https://www.geeksforgeeks.org/inheritance-in-java/
https://www.javatpoint.com/abstract-class-in-java#:~:text=in%20Java%20first.-
,Abstraction%20in%20Java,only%20functionality%20to%20the%20user.&text=Another%2
https://www.javatpoint.com/method-overriding-in-java
https://www.geeksforgeeks.org/polymorphism-in-java/
https://en.wikiversity.org/wiki/Object-Oriented_Programming/Polymorphism/Java
https://en.wikipedia.org/wiki/Logic_programming
https://en.wikipedia.org/wiki/Forward_chaining
https://techdifferences.com/difference-between-forward-and-backward-reasoning-in-ai.html
https://en.wikipedia.org/wiki/Antecedent_(logic)
https://en.wikipedia.org/wiki/Backward_chaining
https://en.wikipedia.org/wiki/Horn_clause
https://en.wikipedia.org/wiki/Proof-theoretic_semantics https://en.wikipedia.org/wiki/Sa
https://en.wikipedia.org/wiki/Negation_as_failure
https://en.wikipedia.org/wiki/Knowledge_representation_and_reasoning
https://en.wikipedia.org/wiki/Fluent_(artificial_intelligence)
https://en.wikipedia.org/wiki/Situation_calculus#Regression
https://en.wikipedia.org/wiki/Logic_programming
https://en.wikipedia.org/wiki/Prolog
https://geeksforgeeks.org/prolog-an-introduction/
https://en.wikipedia.org/wiki/Negation_as_failure#Prolog_semantics
https://www.geeksforgeeks.org/benefits-advantages-of-oop/

https://home.agh.edu.pl/w̃ojnicki/phd/node18.html#sec:relational_model
https://www.developer.com/java/oop-advantages-disadvantages/
https://www.krayonnz.com/user/doubts/detail/61cf0bf6d873e800403a059d/what-are-the-advantages-and-disadvantages-of-OOP
https://www.tutorialspoint.com/Advantages-of-encapsulation-in-Java
https://www.geeksforgeeks.org/inheritance-in-java/
https://www.geeksforgeeks.org/abstraction-in-java-2/
https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/
https://www.codingninjas.com/studio/library/dynamic-method-dispatch-in-java
OpenAI. (2024). ChatGPT (3.5) [Large language model]. https://chat.openai.com
https://en.wikipedia.org/wiki/Comparison_of_programming_paradigms//
https://sameedahmedkhan.medium.com/comparative-study-of-java-and-prolog-c65dce29c91d#:~:text=Functional%20Programming%3A,basic%20functionalities%20of%20d
https://emmachev.com/what-is-the-difference-between-prolog-and-java-programming-languages/
https://prob.hhu.de/w/index.php?title=Prolog_vs_Java_Comparison
https://sicstus.sics.se/sicstus/docs/3.7.1/html/sicstus_12.html
https://www.coursehero.com/file/p45d9uu/Advantages-and-Disadvantages-of-Logical-Programming-Paradigm-Advantages-Logic/
https://www.linode.com/docs/guides/logic-programming-languages/
https://www.allassignmenthelp.com/blog/logic-programming-what-are-its- https://www.al
programming-what-are-its-techniques/#:~:text=Logic%20programming%20is%20limited%2
https://www.geeksforgeeks.org/prolog-an-introduction/
https://www.freetimelearning.com/software-interview-questions-and-answers.php?What-are-the-advantages-and-disadvantages-of-using-Prolog-compared-to-other-programming-languages?&id=10133