PART1:

-In this module we are organizing our structures in a hierarchy. What would be the advantages of doing it this way over a linked list?

Using a hierarchy to organize structures has several benefits over using a linked list:

Better Access performance: In a linked list, we must explore the complete list starting at the beginning in order to reach a particular entry. By tracing a route from the root node to the target node in a hierarchy, we can reach any element. For bigger data collections, this method can be considerably quicker.

Better Memory Management: A linked list has a lot of memory overhead because each entry refers to the one after it. Each node in a hierarchy only needs to keep a reference to its offspring nodes and parent node. Performance and memory handling may improve as a consequence.

Better Search: Algorithms like Depth-First Search (DFS) or Breadth-First Search (BFS) can effectively scan hierarchies.

-What is the purpose of this file (Organization.java)?

The interface contains a number of functions that stand in for features that a company ought to have.

Regular and contract workers can be added to the company using the first two methods, addEmployee and addContractEmployee. These procedures take into account a number of factors, including the employee's name, salary, gender, and boss's name, and they are made to prevent the employee from being added to the organization if the supervisor cannot be located.

The entire number of workers in the company is returned by the getSize method. It requires no inputs.

The number of workers in the company who belong to that gender is returned by the method getSizeByGender, which accepts a gender parameter.

The method allEmployees returns a list of all employees' names in the organization as a List of Strings.

The method countPayAbove takes in a parameter representing a threshold for the annual pay and returns the number of employees in the organization whose annual pay is above that threshold.

Finally, the method terminatedBefore takes in three parameters representing a termination date and returns the number of employees in the organization who are scheduled to be terminated before that date.

By defining this interface, any class that implements it must provide an implementation for all the methods defined within it. This ensures consistency across different implementations of an organization and makes it easier to use and maintain these classes in the future.

-Why start with this file for design (Organization.java)?

One of the basic rules of object-oriented programming (OOP) architecture is to start with an interface. An interface establishes a contract outlining the capabilities of a class that implements it. Without divulging how the utility is actually provided, it gives a high-level overview of what the class should be able to do.

An important aspect of software design is the division of responsibilities, which is promoted by using an interface. A worry is a group of connected tasks that a class ought to carry out. As a result of the division of concerns, each class is in charge of a distinct worry; they are not combined in one class. The interface specifies the duties that the class should perform, and the class itself implements those functions. This allows for greater flexibility and modularity in the design, as changes to one part of the codebase will not necessarily affect other parts of the codebase.

Additionally, interfaces offer polymorphism, which allows objects of various classes to be regarded as belonging to the same type so long as they adhere to the same interface. Since the same code can be used with various items as long as they follow the same protocol, it makes code more flexible and reusable.

-What is the purpose of this file (Employee.java)?

An employee in a company is represented by this Java interface. Any application of this interface must offer the collection of methods that are specified in this document. This interface's goal is to specify the usual actions that an employee takes while working for the company, such as obtaining their name, gender, and yearly salary, adding supervisees to the hierarchy, numbering employees who meet a certain requirement, and turning the hierarchy into a list.

In addition, a way is defined in the interface for retrieving an employee's job termination date, which may be necessary for procedural or human resources (HR) reasons.

-Why is it an interface (Employee.java)?

Due to its role as a contract or definition for other classes that implement the interface, the Employee class is specified as an interface rather than a class. To put it another way, the interface's main function is to specify the collection of methods and attributes that other classes must adhere to in order to be regarded as Employees.

The code that makes use of the Employee object can be separated from the particular version of the Employee by using an interface. As a result, it is possible to write the code in a manner that is not reliant on the specific Employee class being used. Since it is now simpler to switch out various Employee interface implementations without impacting the remainder of the code, versatility and maintainability are increased.

A class can implement a number of interfaces because interfaces support numerous inheritances. When a class needs to have behavior that is governed by several protocols, this can be helpful.

The ability to specify constants and basic behavior for a class is also provided by interfaces. There are no default methods in the Employee interface, but the use of an enum for gender could be interpreted as specifying a collection of constants for the class.

Overall, using an interface in this situation is suitable because it offers a precise definition of what it means to be an employee within the company and permits freedom and maintainability in the code that uses the Employee object.

-Could it have been an abstract class? If so, does it make sense to do so? If not, why (Employee.java)?

Indeed, an abstract class could have been used to represent the Employee type. In that it gives a specification for other classes to follow, an abstract class is comparable to an interface. However, it can also provide default implementations for some of its functions.

An abstract class may have been preferable in the instance of the Employee class if there were methods that could have been applied by default for all workers. Suppose we want to add a new method to the Employee class that calculates the bonus for an employee based on their annual pay. However, the calculation for the bonus is the same for all employees, regardless of their position or gender. In this case, an abstract class might be a more appropriate choice than an interface, as we can provide a default implementation of the bonus calculation method in the abstract class.

Nevertheless, it doesn't appear that there are any preset versions that would be suitable for all workers based on the methods specified in the Employee interface. The class that implements the interface must implement each function specifically because each one seems to be special to the Employee. Because it enables a clear division of responsibilities in this situation and clearly

specifies which functions the class that represents an Employee must implement, using an interface is an appropriate solution.

-Why is this in a separate file (Gender.java)?

Every Java class or list type should be contained in a distinct file; this is standard procedure. This is due to Java's requirement that the file's name correspond to the name of any public classes or enumeration types that are defined there. The completely qualified name of the Gender enumeration type.

It is simpler to handle and arrange the codebase when the Gender enumeration type is separated into its own file. It also provides for improved readability and maintainability of the code and adheres to the accepted Java coding style. It also makes the Gender enumeration class easier to utilize in other areas of the codebase or even in different applications.

-Does it have to be(Gender.java)?

Potentially, the Gender enumeration type could be defined in a different Java file that also includes other classes or interfaces. The clarity, maintainability, and reusability of the code can all be improved by dividing each class or enumeration type into its own file, though this is typically regarded as a good practice.

Furthermore, it may be more challenging to browse and find particular sections of code if there are several classes or enumeration types present in the same file. Additionally, you would have to import the complete file containing the Gender enumeration type if you wanted to use it in another file, which might introduce unnecessary dependencies and complicate your code.

-Why is this an abstract class instead of an interface (GenericEmployee.java)?

mainly because it offers some preset implementation of the Employee interface's specified methods. An interface doesn't offer any preset implementations; it merely specifies a contract that implementing classes must follow.

In this instance, the getEmploymentEndDate() and toList() methods are provided with default implementations by the GenericEmployee class, which can be overridden by specific versions as needed. Furthermore, it makes more sense for the count() method to be specified in a generic class

that has access to its own fields and methods because its implementation relies on the present object.

-If only contractEmployees have end dates, why do we need to include a getEndDate (GenericEmployee.java)?

To guarantee that all versions of Employee have a standardized method signature, the getEmploymentEndDate() method is included in the Employee interface and implemented in the GenericEmployee abstract class.

Because even though GenericEmployee's default version gives "XXXXXXX" for all workers, concrete implementations of Employee, other than ContractEmployee, can override this function to offer their own code that returns a real end date. This makes the job with different employee categories, regardless of whether they have an expiration date or not, simpler for users of the Employee interface.

-If interfaces enforce specific methods, why are we able to compile this class by itself without all those methods created yet (GenericEmployee.java)?

The GenericEmployee class, in this case, states that it uses the Employee interface and offers a legitimate implementation for the interface-overriding methods. The abstract class provides a fallback implementation for the methods that are not changed, such as getEmploymentEndDate(), so the actual subclass is not required to implement them.

-Why isn’t add supervisee here (GenericEmployee.java)?

It would be implied that all versions of Employee must have this function by including addSupervisee() in the Employee interface or the GenericEmployee abstract class, even if it is not applicable to them. This would go against the interface isolation concept.

-What is the super constructor doing here (Supervisor.java)?

The GenericEmployee constructor initializes the appropriate instance variables in the GenericEmployee class by taking name, salary, and gender as inputs. The name, pay, and gender values are given up to the parent class to be used in initializing the respective instance variables in the GenericEmployee class by calling super(name, pay, gender) in the Supervisor constructor.

-How is the internal data stored (Supervisor.java)?

The details of the employees it supervises is kept in a List of Employee instances by the Supervisor class.

-There are two variables here with the same name, but they are serving different purposes and they are even different types. Fix this by naming one better (Supervisor.java).

For instance, we could change the supervisee variable's name to assistants to denote that it contains a collection of the workers who are under this supervisor's authority.

-The if statement and the for loop are particularly interesting. Explain to each other what is going on here (Supervisor.java).

These are the 5 steps that can describe the if/else and for loop together:

a) The name of the supervisor to whom the new supervisee will be assigned and the Employee object that represents the new supervisee are passed as the first two arguments when calling the addSupervisee function.

b) The new supervisee is added to the supervisee list of the current Supervisor object using the add function if the name of the current Supervisor object fits the supervisorName parameter.

c) The for loop is used to run through each Employee object in the supervisee list of the current Supervisor object if the name of the current Supervisor object does not match the supervisorName argument.

d) The addSupervisee function is invoked repeatedly for each Employee object in the supervisee list, passing the same supervisorName parameter and the new supervisee Employee object.

e) Until it locates the Supervisor object with the correct supervisorName parameter, the recursive action will iterate through the chain of oversight. The new supervisee is added to that Supervisor object's collection of supervisees using the add function once it has been located.

-Why use the linked list set function instead of just get?

Instead, if the get function of the LinkedList class were used, it would just return the supervisee at the given number without changing the list. The set method is used to replace the outdated supervisee at the same index with the new supervisee, successfully adding the new supervisee to the list and adding the new supervisee to the right spot in the list.

-What is the purpose of this file (NonManagerEmployee)?

NonManagerEmployee, a specific version of the Employee interface, is defined in this file. This class implements the addSupervisee function inherited from the Employee interface and depicts an employee who has no managerial duties.

-What's the purpose of this file (ContractEmployees.java)?

This file defines the ContractEmployee class, which depicts an employee on a period contract with a company. The class includes a contractEndDate field, a LocalDate object reflecting the employee's contract end date, and extends the NonManagerEmployee class. The class also has a getEmploymentEndDate method that returns a formatted string representation of the contract end date, as well as a constructor that accepts the employee's name, pay, gender, and contract end date.

-We are making our way to OrganizationImpl. Why do we have to examine and compile each employee type first (ContractEmployees.java)?

We can make sure they are all accurately described and can communicate with one another as anticipated by looking at and assembling each employee category separately.

-Compile this file (ContractEmployees.java)?

```
3/25/2023 9:24 AM - Build completed successfully in 1 sec, 748 ms
```

-What is the purpose of this file (OrganizationImpl.java)?

The Organization interface, which simulates a company with workers, is implemented in this file. It contains techniques for hiring new employees and contract workers, determining the size of the company and the number of employees of a particular gender, obtaining a list of every employee, counting the number of workers earning salaries above a certain threshold, and calculating the number of people fired before a certain date. In order to keep and handle the workers, the implementation makes use of a variety of employee types, including NonManagerEmployee and ContractEmployee.

-Why does the only constructor take in an employee? Why does it have an employee root (OrganizationImpl.java)?

The name, salary, and gender of the organization's Leader are inputted into the constructor of the OrganizationImpl class, which depicts a company with workers. Since it utilizes this object to establish the base of the organization's hierarchy, it accepts an Employee object as an argument.

In other words, the highest point in the organizational structure is represented by the Employee object given as an input, which is the Leader of the company.

-How does the allEmployees function work? Check out the stream breakout code if needed (OrganizationImpl.java)?

a) The toList method on the main employee object is invoked by the function, and it gives a collection of every employee in the company.

b) The list of workers is given a call to the stream function, which turns it into a stream of elements.

c) When the stream's map method is invoked, each element of the stream is subjected to the given function (in this instance, e->e.getName()), and a new stream holding the results is returned.

d) When the collect function is invoked on a stream, Collectors.toList() is used to gather the stream's elements into a new list.

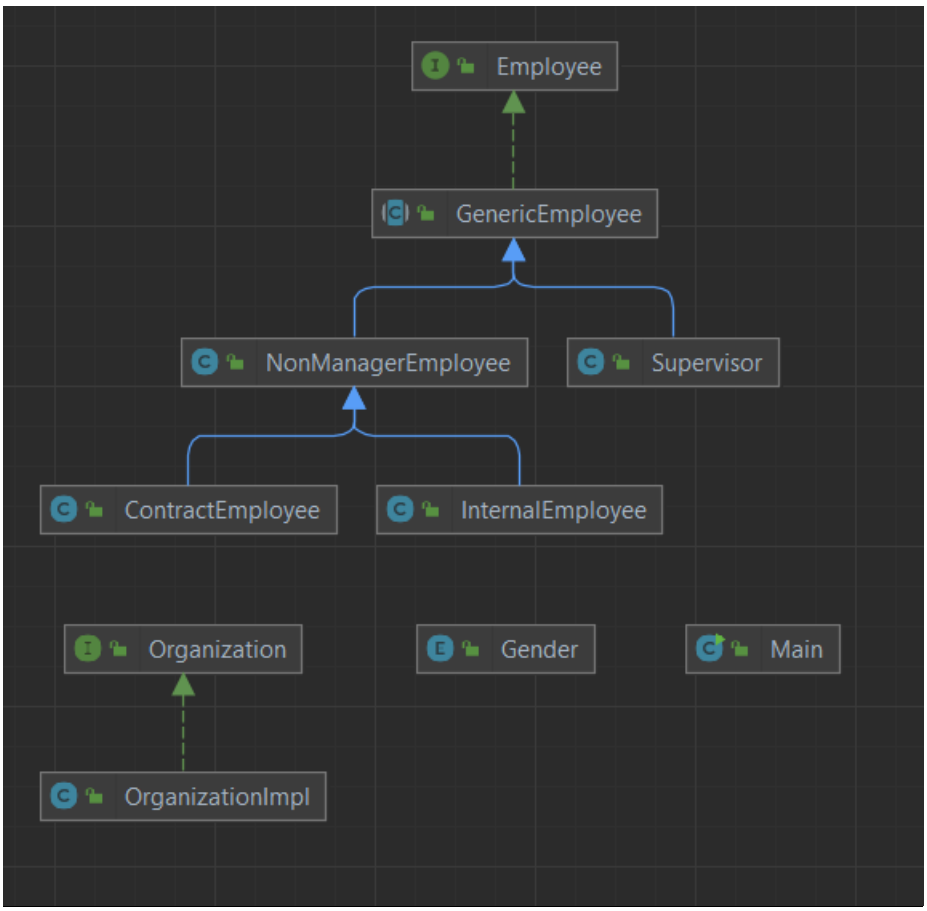
PART2

(Features have been added to source code)

- When printing an entire organization does the print order make sense?

Yes, they are getting print according to their breath generation, with CEO on the top and other supervisors leading. Going all the way down to non-manager staff.

- Draw out the inheritance of this code.



- How is this a “hierarchical data structure?

The fact that each node in this data structure has zero or more descendant nodes indicates that it is a hierarchy data structure. Each Employee object in this scenario has zero or more Supervisee objects, who are also Employee objects.

The Leader of the company is represented by the root object of the OrganizationImpl class, which stands for the main component of the hierarchy. The addSupervisee function assigns the new employee as a supervisee to the supervisor after conducting a recursive search for that supervisor using the given name. This indicates that in the structure, the new employee is now the supervisor's offspring node.

The Supervisor class depicts a node in the hierarchy, and it can oversee numerous Employee instances. Each employee entity has the ability to oversee others, meaning it can have supervisees of its own.

-Why would removing an employee be more complicated than adding one?

Because it may have an effect on the organization's hierarchy and the connections among its members, dismissing an employee from a hierarchical organization can be more difficult than hiring one. Each employee who reports to a particular employee must be given a new boss if that employee is fired. This may necessitate reorganizing the structure and have an impact on the entire company. In addition, it's possible that there are continuing tasks or other dependencies that need to be finished or reassigned before the employee can be let go. However, adding a new employee usually only necessitates assigning them to a current boss and updating the structure as necessary.

- How is this a recursive data structure?

Because the Employee class is a component of the Supervisor class, which itself may contain Employee instances, this data structure is iterative. As a result, there is a structure of employees with supervisors and lower-level workers.

-Why can’t we get an employee by number?

The workers are not explicitly numbered in the provided design. Each worker is recognized by their distinctive name. As a result, we are unable to identify an employee by number because no such naming system exists.

In other words, there is no employee number one or employee number 10. Staffs are categorized according to their name, rule and etc.

-What's the difference between the NonManagerEmployee and Supervisor count methods?

In short, the Supervisor count method counts both itself and all of its employees, whereas the NonManagerEmployee count method only counts itself.