Assignment 01

Isabelle Dahlström
Bachelor in Software Development
isabelle.dahlstromm@gmail.com

January 2025

```java
public class Main {
    Run | Debug
    public static void main(String[] args) {
        String task1 = "Done";
        String task2 = "Done";
        String task3 = "Done";

        // Check if all tasks are done
        if (task1.equals(anObject:"Done") && task2.equals(anObject:"Done") && task3.equals(anObject:"Done")) {
            System.out.println(x:"🎉 Hooray! I'm done with A01! 🎉");
        } else {
            System.out.println(x:"Keep going!");
        }
    }
}
```

Contents

1 Introduction

This report documents the process I followed creating the three tasks for Assignment 01. I will reflect on my work, sharing my thoughts and experiences as I tackled larger programming projects in Java for the first time. Additionally, I will discuss the steps taken to create functional and user friendly applications and how the pillars of OOP theory take a part in the programs.

The first part of this report focuses on Task 01, which emphasized the basics of Java programming: understanding Classes and Objects and how they interact. The second part discusses my process constructing Task 02, where inheritance and compositions were key concepts, along with the implementation of collections. Finally, the last part of the assignment, Task 03, centered on Classes, Objects, Interfaces and Abstract Classes. This part also contained File Handling in different approaches.

2 Task 01

For Task 01, the main focus was on the fundamentals of Java classes and their interaction. Classes containing methods and properties were used in combination with conditions and iterations to create a smaller functional program. A menu system was created for Requirement 1, and for Requirement 2, both a Rock, Paper, Scissors game and a Dice Game were developed.

2.1 Working process

The first implementation I worked on was the menu, consisting of five simple switch cases. Menu option '1' printed my avatar, requiring only a method call with my class name. Option '2' displayed the local time and today's date using the "LocalDateTime" and "DateTimeFormatter" classes. Option 'm' reprinted the menu by calling the respective method, making it easy to navigate.

Next, I moved on to the two games: Rock, Paper, Scissors and DiceGame21, where things got more interesting.

For Rock, Paper, Scissors, I created a class with methods to handle different parts of the game. Since there were two players—myself and my avatar Kitty—I made a method for my move input, validating that it was one of 'r', 's', 'p', or 'q' (to quit) and exactly one character long as shown in Figure 8. Kitty's move was determined by a method that randomly selected a character from an array containing the valid options. To decide the winner, I compared the moves and formatted the results into new String variables for better readability. I implemented an if-else structure to evaluate all possible outcomes. While I felt there might be a more efficient approach, I stuck with what I understood as a Java beginner. Points were tracked using instance variables, with scores displayed after each round.

```java
public String getChoice(Scanner scanner) {
    System.out.print(s:"Your choice: ");

    String choice = scanner.nextLine().toLowerCase();
    if ("rspq".contains(choice) && choice.length() == 1) {
        return choice;
    } else {
        System.out.println(x:"Invalid choice. Please enter 'r', 's', 'p', or 'q'.");
        return getChoice(scanner);
    }
}
```

Figure 1: Conditions for user input

DiceGame21 was built similarly, with methods for Kitty's turn, her dice rolls, and the player's turn. The game involved keeping track of the sum of dice rolls. If a player exceeded 21, they would lose, and Kitty would gain a point. Otherwise, points were awarded based on who had the higher score, with ties favoring Kitty. A "displayScore" method presented the scores after each round, providing a clear summary of the game's progress.

2.2 Learning outcome

This task helped me become more comfortable working with classes and objects. I learned how to create a class, objects and method, as well as how to implement decision structures and properly construct a switch case. Initially, it was challenging because I was so accustomed to Python, but as I learned more about how Java works I was impressed by its structure and features. I got to learn the differences in programming in Python verses Java and I could use a lot of my previous Python knowledge even though it has its differences. Additionally, I gained valuable knowledge in reading UML diagrams, which helped me identify the necessary fields, classes and methods required to complete the task. The UML diagram guided me in constructing the program and provided a solid foundation for learning how to develop a program independently.

2.3 TIL;

Today, I gained a deeper understanding of how to create objects and methods in Java. I now understand how getters and setters work and how to use them effectively. Additionally, I learned how to create an array of characters and how to randomly pick a character from it.

2.4 UML Diagram

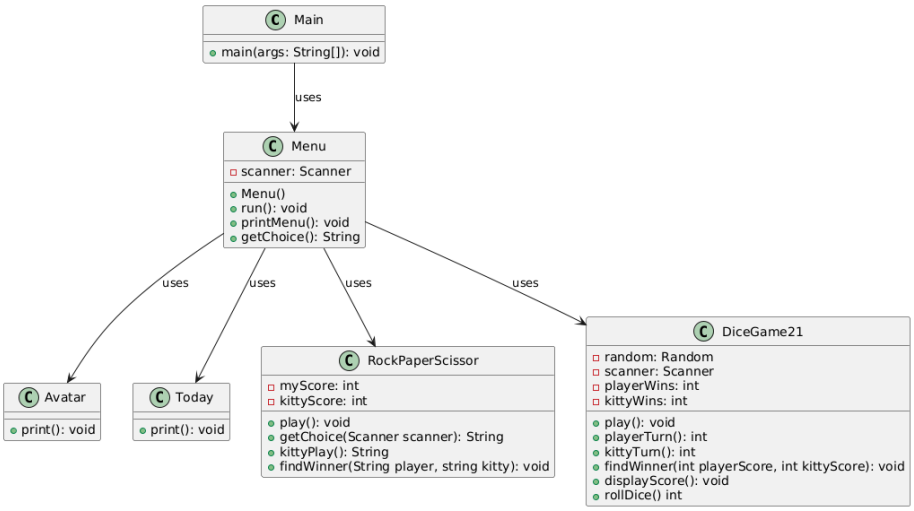


Figure 2: UML Diagram for Task 1

3 Task 02

The main focus of Task 02 is to further develop skills in working with classes and objects, with an emphasis on using inheritance and composition. This program also incorporates collections alongside the classes. I have completed 14 requirements, building a management system for employees in a team. The user can create a team and add both regular employees and super-employees. Regular employees have an ID, a salary, and a work position, while super-employees have an ID, superpowers, and work positions. This task was also construed using the automation tool Gradle and contains Java documentation.

3.1 Working process

When structuring the Menu class, my goal was to make it independent from the application code to increase code reusability. The outcome was somewhat independent; the Menu class contains the logic for displaying and navigating the menu. It also handles inputs and includes essential methods like displayMenu() and runMenu(), which are crucial for the program to function. The runMenu() method is built with a loop and a switch statement. Some error handling is also included to ensure the input is valid.

At first, creating the class diagram was challenging, but as I learned more about inheritance and composition, it became easier to understand how the program should be structured.

```java
public String toString() {
    return " Emp(" + getId() + "): " + getName() + " (" + work + ")";
}


public static int getNewEmployeeId() {
    return globalEmpId++;
}
```

Figure 3: The method toString() and getNewId()

The concept of dependency injection is used within the Menu class. Instead of creating a Team object inside the class, it accepts the Team object via its constructor. The Main class creates a Team object and passes it into the Menu constructor. This approach is efficient because it allows you to easily replace the Team with a different implementation, promoting reusable code.

A challenging part of this task was understanding the difference between inheritance and composition. It took me a while to grasp these concepts fully. Overall, seeing the connection between all the classes was difficult and took considerable time.

Writing JavaDocs for the first time was a great opportunity to practice identifying the most important parts of the code. Initially, it was hard to know what to include in the documentation, but over time it became easier. Using JavaDoc was relatively straightforward and helped provide a better understanding of the methods.

I decided to complete all the optional parts of this task, which included creating unique employee IDs, generating a salary report, and providing JavaDoc comments for all classes, methods, and properties. The employee IDs were implemented using a counter in the Employee class. Each time an employee is created, the counter increments, assigning a unique ID to each employee. As shown in Figure 8, the getNewId() method increments the globalEmpId (initially set to 1) every time it is called. In the toString() method, the ID is included in the string representation of the employee.

3.2 Optional parts

The optional menu choice "7) Salary report" required additional work. I created a method called salaryReport() that returns a string. To build the salary report, I used the built-in StringBuilder class, allowing me to append each part of the report to construct a final string. As seen in Figure 8 method uses a for-each loop to iterate through each employee in the member array, formatting their salary in a readable way. It also contains an integer, totalSalary, which starts at 0. With each iteration, the employee's salary is added to this total.

Finally, the total salary is appended to the StringBuilder and formatted for presentation. A date formatter is also used to represent the date on which the report was generated. When selecting menu option 7, the salaryReport() method is called on the team object, which is an instance of the Team class. The return value of this method is then printed to the console using the System.out.println() method.

```
for (Employee emp : member) {
    String formatSalary = String.format(format:"%d", emp.getSalary());
    sb.append(String.format(format:" (%d) %-20s %10s\n", emp.empID, emp.name, formatSalary));

    totalSalary += emp.getSalary();
}
```

Figure 4: The for each loop in the salaryReport() method

3.3 TIL;

Today, I learned how to install and use a build tool called Gradle. Gradle automates the build process for Java projects, meaning it automatically compiles your Java files. At first, I didn't understand the benefits of using it, but once I figured out how to work with it, everything clicked. Now, I prefer using Gradle for larger programs because it simplifies the build process.

I also learned about inheritance in Java and how to use it. Inheritance is a powerful OOP concept that allows code reuse, so you don't have to write the same code multiple times. I got to practice using method overriding and the super keyword, which further deepened my understanding of how inheritance works in object-oriented programming.

3.4 UML Diagram

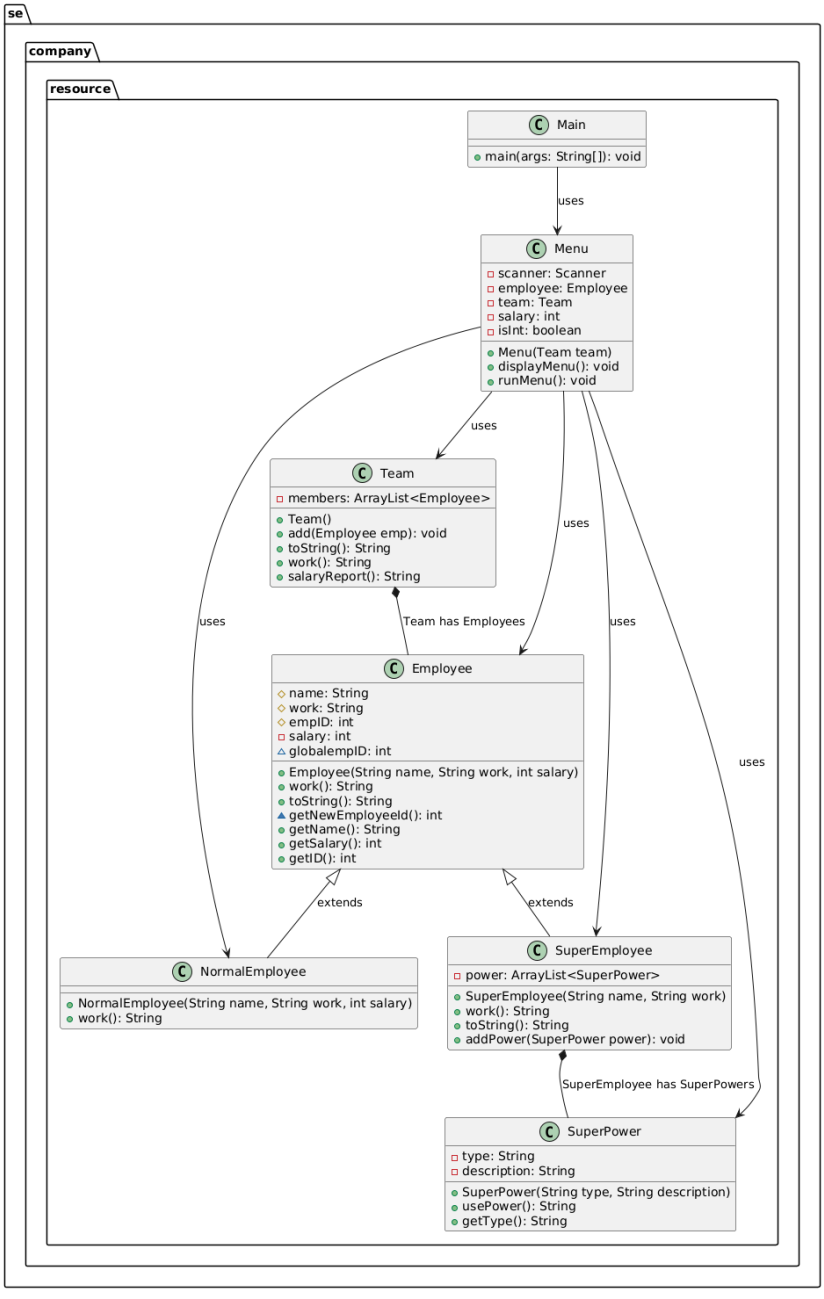


Figure 5: UML Diagram for Task 2

4 Task 03

The final and most challenging task focuses on concepts such as classes, objects, interfaces, and abstract classes, integrating collections and file handling. The application is a game that allows the user to interact with a 10x10 grid, where the player navigates in an attempt to reach their home before being caught by a wolf. The game includes a menu with 10 options, with the last two options dedicated to saving and loading the game state. This task requires a understanding of OOP principles, along with the ability to manage game state through file operations.

4.1 Working process

The provided UML diagram made the working process significantly easier. The diagram allowed for a better understanding of the program almost immediately. At first glance, you could determine that the Position class should be created first because of its hierarchy, it is not dependent on any other class.

Some methods were a bit more challenging to grasp in terms of their purpose. For instance, in the Position class, I initially struggled to understand why another constructor was needed. The Forest class, on the other hand, contained numerous methods, and it was crucial to implement them in the correct order to avoid confusion during the process.

Overall, I am satisfied with the final product, even if there is room for improvement. For example, I could have worked on implementing a smarter wolf that follows the player instead of moving randomly. Additionally, the wolf could be programmed to recognize when it moves out of the game area. However, these features seemed a bit too challenging for me as a beginner, so I decided to leave them out for now.

I also found the concept of abstraction challenging to understand initially. It took me some time to grasp why it was necessary and how to use it effectively.

4.2 Differences between inheritance, composition, interface and abstract classes

This task gave me a better understanding of the object-oriented programming pillars and the differences between inheritance, composition, interfaces, and abstract classes. Inheritance represents an "is-a" relationship, which is used to inherit features such as fields and methods from another class. Essentially, inheritance in Java means creating new classes based on existing ones [3].

By using inheritance, programmers can create flexible and reusable code, which is a crucial skill for becoming a good developer and creating effective programs. To implement inheritance, the keyword extends is used in the class that inherits from another.

Let's consider an example from my script: the class Castle extends the abstract class AbstractMoveableItem. This means that AbstractMoveableItem is the Parent class, and Castle is the Child class. They work together through the AbstractMoveableItem constructor, which takes three arguments: description,

graphic, and position. In the Castle class, these parameters are passed to the superclass using the super keyword. This means that you can create additional classes that use these arguments, for example, in my Cat class.

Composition, on the other hand, represents a "has-a" relationship. This technique is also employed for code reusability, but it achieves this by using an instance variable that refers to other objects[1]. A straightforward example to illustrate composition is a car and its engine. A car cannot properly function without an engine, and an engine cannot operate without the car. The engine is an integral part of the car, and the two are inherently dependent on each other.

Moving on to interfaces and abstraction, these concepts are often considered similar because both aim to hide implementation details. Abstract classes and interfaces cannot be instantiated directly. An interface can be seen as an abstract type that defines the behavior a class must adhere to[2]. It does not contain method implementations but rather specifies the methods that any implementing class is required to define. Essentially, interfaces serve as a "contract" for the classes that use the implements keyword.

Abstraction shares a similar purpose, as it focuses on exposing only the essential details to the user while concealing the underlying implementation. A helpful analogy for abstraction is a TV remote: you can use all the buttons without understanding the intricate mechanics behind their functions. In Java, both abstract classes and abstract methods exist, but you cannot create objects from an abstract class. To use abstraction, you employ the abstract keyword.

4.3 Optional parts

For the optional part, I decided to add more game characters for the user to choose from. In the menu class, I implemented a simple input validation system that allows users to select between playing as a Cat, Dog, or Robot (seen in Figure 8). This was relatively straightforward to implement. I created additional character classes similar to the Robot class and added the necessary logic to the menu for input handling and validation.

```
do {
    System.out.print(s:"Select your player, Cat (c), Dog (d) or Robot (r): ");
    String playerChoice = scanner.nextLine();

    if (playerChoice.equalsIgnoreCase(anotherString:"c")) {
        player = new Cat(new Position(x:1, y:1));
        forest.addPlayerItem(player);
        System.out.println(
                "Selected player: " + player.getDescription() + " " + player.getGraphic() + "\n");
        break;
    } else if (playerChoice.equalsIgnoreCase(anotherString:"r")) {
        player = new Robot(new Position(x:1, y:1));
        forest.addPlayerItem(player);
        System.out.println(
                "Selected player: " + player.getDescription() + " " + player.getGraphic() + "\n");
        break;
    } else if (playerChoice.equalsIgnoreCase(anotherString:"d")) {
        player = new Dog(new Position(x:1, y:1));
        forest.addPlayerItem(player);
        System.out.println(
                "Selected player: " + player.getDescription() + " " + player.getGraphic() + "\n");
        break;
    } else {
        System.out.println(x:"Please select 'c', 'd', or 'r'.");
```

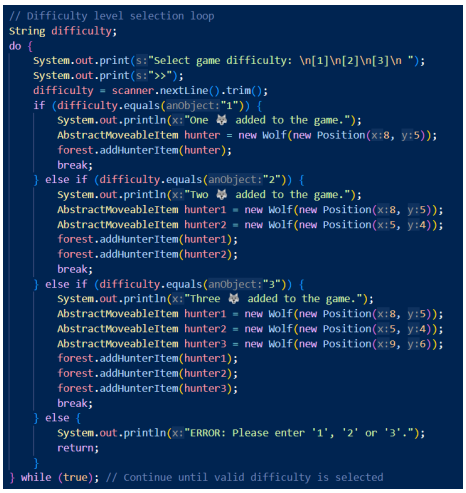
Figure 6: Menu option 6 with player characters to choose from.

I also wanted to make the game more challenging. Instead of just speeding up the wolf, I decided to introduce an option for the player to select the difficulty level. After selecting the character, the user can choose the difficulty of the game (1, 2, or 3). To implement this, I created multiple wolf objects based on the user's selected difficulty. This was done using a loop combined with if-statements. The changes are visualized in Figure 7.

After implementing this, I needed a place to store the wolves, so I added a list in the Forest class. This list is looped through in the getGamePlan method, where a for-each loop adds all the wolves to the game plan.

Additionally, I modified the movePlayer method. Instead of just moving a single wolf, I needed to account for all the wolves in the game. This meant looping through the list of wolves, checking each one's position, and moving it randomly. The logic ensures that all wolves move whenever the player moves, making the game more complex and interactive.

To handle this dynamic movement of wolves, I created the setPosition method in the AbstractMoveableItem class. Since the position field in the Forest class is private, I needed a way to change the wolves' positions easily. The setPosition method allows me to update the position of each wolf after it moves, making sure that the wolves are displayed correctly on the game board and can interact with the player.



```java
// Difficulty level selection loop
String difficulty;
do {
    System.out.print(s:"Select game difficulty: \n[1]\n[2]\n[3]\n ");
    System.out.print(s:">>");
    difficulty = scanner.nextLine().trim();
    if (difficulty.equals(anObject:"1")) {
        System.out.println(x:"One 🐺 added to the game.");
        AbstractMoveableItem hunter = new Wolf(new Position(x:8, y:5));
        forest.addHunterItem(hunter);
        break;
    } else if (difficulty.equals(anObject:"2")) {
        System.out.println(x:"Two 🐺 added to the game.");
        AbstractMoveableItem hunter1 = new Wolf(new Position(x:8, y:5));
        AbstractMoveableItem hunter2 = new Wolf(new Position(x:5, y:4));
        forest.addHunterItem(hunter1);
        forest.addHunterItem(hunter2);
        break;
    } else if (difficulty.equals(anObject:"3")) {
        System.out.println(x:"Three 🐺 added to the game.");
        AbstractMoveableItem hunter1 = new Wolf(new Position(x:8, y:5));
        AbstractMoveableItem hunter2 = new Wolf(new Position(x:5, y:4));
        AbstractMoveableItem hunter3 = new Wolf(new Position(x:9, y:6));
        forest.addHunterItem(hunter1);
        forest.addHunterItem(hunter2);
        forest.addHunterItem(hunter3);
        break;
    } else {
        System.out.println(x:"ERROR: Please enter '1', '2' or '3'.");
        return;
    }
} while (true); // Continue until valid difficulty is selected
```

Figure 7: Implementation of game difficulty.

4.4 TIL;

For this task, I learned a lot about OOP theory. One key concept I learned was abstraction — what it means, how to implement it, and why it's important to use (I've written more about this in subsection 4.2). Initially, this concept was challenging, and there were many new ideas to grasp. However, I eventually got a clearer understanding.

Additionally, I decided to implement more error handling than before, which turned out to be great practice. My ability to create input validation loops has improved significantly and now feels much easier. I also learned how to use a Map in Java, and I became more proficient with for-each loops. Finally, the concept of constructors also became much clearer to me throughout this task.

4.5 UML Diagram

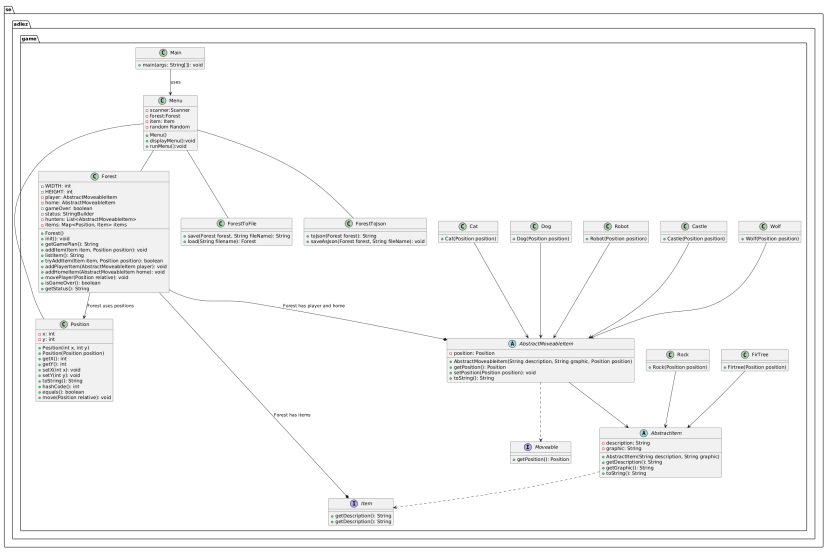


Figure 8: UML Diagram for Task 3

5 Bonus Points

For this assignment, I completed all the optional parts. In Task 1, I successfully created an application that includes both a Rock Paper Scissors game and a Dice Game. In Task 2, I implemented unique IDs for the employees and added an extra menu option to generate a salary report. In the final task, where we had more flexibility to enhance the game, I added extra character options and three different difficulty levels. The difficulty levels are determined by the number of hunters/wolves in the game. For more details, you can refer to the sections for each task.

6 Summary

Completing this project allowed me to learn a lot of new skills. To start, I became more comfortable with the fundamentals of Java, including creating classes and objects with methods. As I worked through each task, things became clearer, and I got into the right way of thinking when using an object-oriented programming language. At first, it was overwhelming, but as I completed task after task, looking back at Task 1, it seemed much simpler. It was clear that my learning progressed each week.

The four pillars of Java OOP also felt overwhelming, especially abstraction. It took time to fully grasp this concept, but with lots of practice, reading course

materials, and consulting various websites, things began to make sense. I found that the best way to learn the theory was to apply it in practice—not just read about it. The more I worked on smaller programs and extra tasks, the more comfortable I became with the concepts. I also realized how much my knowledge of Python helped me with Java, even though the two languages have their differences. They share many similarities, which made the transition smoother.

In summary, this assignment pushed me to learn Java properly. Without a solid understanding of Java's basics, I wouldn't have been able to progress to the more complex tasks. Overall, it was a fun challenge, and I've grown to appreciate the language more. Java is more complex than Python, and it feels more like "real" programming, which I value. While the theory can be difficult at first, once everything starts to click, Java proves to be a very clever language. I now understand why it is so widely used and appreciated by developers

6.1 TIL;

The most important thing I learned was gaining a solid understanding of the concept of OOP. For me, it was a new way of thinking about programming. I learned how all classes and objects work together and how they are interconnected. Additionally, getting familiar with error handling was a valuable lesson. It's crucial to implement error handling to prevent programs from crashing, and I now see how essential it is for creating robust applications. I believe these are some of the most essential concepts to grasp when developing high-quality software.

References

[1] GeeksforGeeks. Composition in java, 2024. Accessed: 2024-12-25.

[2] GeeksforGeeks. Interfaces in java, 2024. Accessed: 2024-12-25.

[3] GeeksforGeeks. Inheritance in java, n.d. Accessed: 2024-12-24.