Kristianstad University
SE-291 88 Kristianstad
Sweden
+46 44 250 30 00
www.hkr.se

Assignment A01

Object-Oriented Programming in Java, DA114F HT24

Timur Ramazanov

1 Task 01: Get going with Java classes

1.1 Thought process

- **Understanding Requirements:** Analyzed task requirements to identify key components such as classes, terminal I/O, conditions, iterations, and exception handling.
- **Planning:** Designed class structure with clear responsibilities, defined properties, and methods for specific tasks.
- **Implementation:**
  - Developed each class incrementally, focusing on simplicity and logical operations.
  - Used terminal I/O for user interaction
- **Testing:**
  - Tested incrementally, verifying correctness of conditions, iterations, and edge cases.
  - Deliberately input invalid data to validate exception handling.
- **Debugging:** Fixed errors by refining logic, adding validation checks, and improving exception handling.
- **Finalization:** Reviewed and commented code for clarity, with a final round of testing to ensure all requirements were met.

1.2 Following the UML diagrams

Following the UML diagrams significantly helped in structuring the code and defining the relationships between classes. A well-detailed scheme does half the work for you. The provided diagram was sufficiently clear and effective for a project of this simplicity.

1.3 Difficulties learning Java as a new language

Having prior experience with C++ and other object-oriented languages made learning Java relatively straightforward. OOP concepts are universally shared, with each language implementing them in a similar way. Syntax-wise, Java is no more challenging than other strictly typed languages. Its syntax closely resembles that of the C family, and I only needed to look up a few minor details while completing this task.

1.4 My solution

When coding my solution, I started by implementing the bare minimum to make the program compilable, adding print statements to outline the logic to be executed. Once the core functionality was in place, I proceeded to complete the remaining features. Afterward, I tested various scenarios to ensure everything worked as expected.

The final step was refactoring and splitting the code into well-structured components. I believe this step is the most crucial, as many developers stop working once the program runs without issues. However, it's important to consider future use. If someone, including yourself, needs to revisit the code, a well-structured and documented program will be much easier to understand and maintain. Leaving behind a messy but functional codebase benefits no one in the long run.

Initially, I had all the logic for the dice game implemented within a single function. However, after refactoring the code, I decided to split the logic into several private helper functions.

```
// DiceGame21.java
private void playRound() {
    playerTurn();
    if (isBullseye()) {
        return;
    }
    botTurn();
    determineWinner();
    displayScore();
}
```

This approach improved the organization and readability of the code, making it easier to maintain and understand.

1.5 My UML diagram

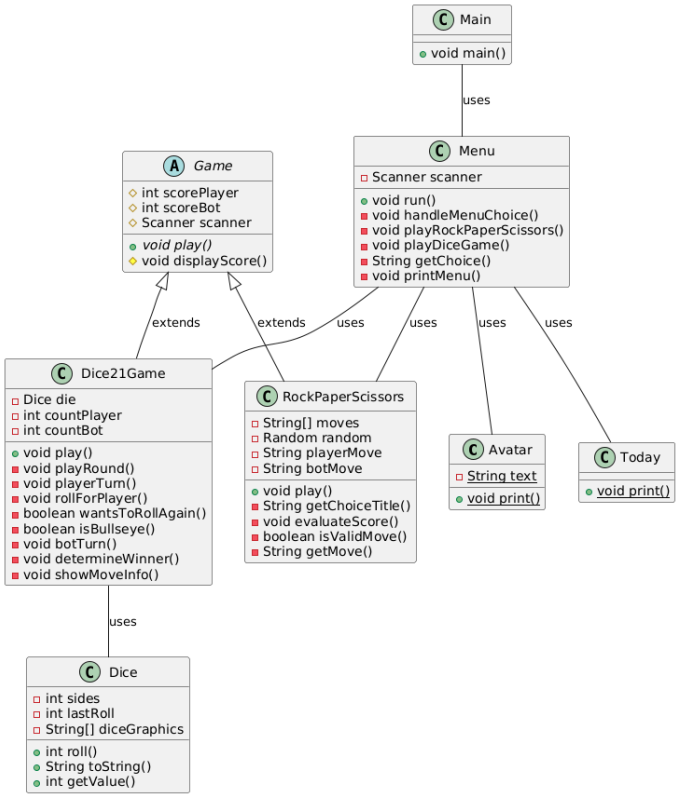

Figure 1: Implementation of Task 01

1.6 TIL

Today, I learned more about string formatting and how to align values in formatted output. Additionally, I explored a new switch format, known as the switch expression with the arrow syntax. This format simplifies switch statements and eliminates the need for break statements, as shown in the following example:

```
// RockPaperScissors.java
return switch(cmd) {
    case "r" -> "rock";
    case "s" -> "scissor";
    case "p" -> "paper";
    default -> "Unknown";
};
```

2 Task 02: Classes, Objects, Inheritance and Composition

2.1 Menu Class

In the given task, there was a requirement to make the **Menu** class independent from the app's logic. I achieved this by using a **HashMap** to store actions, which can be modified externally. Later, in the **run()** method, the user is prompted to enter their choice, and the corresponding action from the **HashMap** is executed.

```
// Stores menu actions mapped to their keys
final private LinkedHashMap<String, Runnable> actions = new LinkedHashMap<>();

// Picks an actions and executes it
actions.get(choice).run();
```

This approach ensures that the Menu class is fully independent from the app's logic, making it reusable and "plug-and-play" in any other application. By externalizing the action definitions, the menu can be easily integrated and customized in different contexts.

2.2 Following the class diagram

Following the diagram was straightforward, as it was created using standardized symbols. The graphics provided all the necessary information, making it easy to understand and implement the structure.

2.3 Dependency injection

When one object uses another object as its property, it can be passed as a parameter in the parent's constructor or via a setter method. This approach is particularly useful when an object acts as a wrapper to organize other objects. In this project, I used setter injection to add members to the team and assign superpowers to superemployees.

```
// The created user is added to the team
NormalEmployee john = new NormalEmployee("John", "IT", 1000);
team.add(john);
```

2.4 Challenging parts

The most challenging part of this task was managing formatted output. In Java, concatenating strings creates new objects instead of modifying the existing ones, which can lead to performance

issues, especially when dealing with large-scale outputs. In our case, there were many outputs related to the team, and each output required iterating through all the members, which became more computationally intensive as the number of members increased. To address this, I used StringBuilder, which is more efficient. However, ensuring that all the spacings were correct took considerable time and effort.

2.5 Writing JavaDoc

Javadoc is a powerful tool for team development, as it generates unified documentation with searchable links. In large applications, where the code may be used by others, specifying everything in Javadoc becomes essential. I always make an effort to document everything that might take time to understand, whether it's a class, function, or even a specific line of code.

2.6 Solving optional part

2.6.1 An enhanced UML class diagram

I followed the diagram closely and implemented all the methods exactly as specified.

2.6.2 Unique employee ID

1. Added static variable **GLOBAL_ID** to **Employee** class to store the latest added user's ID
2. In the Employee's constructor get new ID and assign it to a new object instance

   ```
   this.id = getNewEmployeeId();
   ```

3. The function **getNewEmployeeId** increases **GLOBAL_ID** by one so that no objects with the same ID are created

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

4. To simplify specifying an employee's type in the output, I added a **prefix** property, which contains the employee's type and ID. Normal employees have the prefix **Emp(ID)**, while super employees use the prefix **Sup(ID)**.

2.6.3 Salary report

To generate the salary report, I created a method called **salaryReport** that iterates over the employees to retrieve their salaries. The most challenging part was aligning the values correctly in the output. To address this, I used formatted strings with a specified minimum width.

```
String.format("(%d) %-21s %5d\n", emp.getId(), emp.getName(), emp.getSalary())
```

2.6.4 JavaDoc

To create the Javadoc, I first added comments to all functions and then used ChatGPT to refine the spelling and enhance the vocabulary.

2.7 TIL

Today, I learned about **LinkedHashMap**. While creating the **Menu** class, I initially used a **HashMap** to store the menu options. However, I encountered an issue: a standard **HashMap**

does not maintain the order of its entries, which was not suitable for my use case. I needed the options to remain in the exact order they were added. To resolve this, I used a **`LinkedHashMap`**, which preserves the insertion order of its entries, similar to Python's **`OrderedDict`**.

3 Task 03: Interface, implements and file management

3.1 Following the diagram

As I mentioned earlier, following the given diagram is half the job done. Coding while referencing it saves a lot of time and is highly beneficial, as it allows you to review the entire structure before starting implementation.

3.2 Difference between inheritance, composition, interface and abstract classes

- Inheritance is a mechanism where one class (child or subclass) derives properties and behaviors from another class (parent or superclass).
- Composition is a design principle where a class contains references to other objects (instances) to reuse their functionalities.
- An interface is a contract that defines methods a class must implement without providing the method implementations themselves.
- An abstract class is a class that cannot be instantiated and is meant to be subclassed. It can contain both fully implemented methods and abstract methods (without implementation).

3.3 Proposed modifications

In the given diagram, the hunter (**`Wolf`**) is depicted as an **`AbstractMovableItem`**, which is conceptually correct. However, hunters possess a unique behavior that distinguishes them: hunting. To address this, I introduced a new interface called **`Hunting`** and a subclass called **`AbstractHunter`**. The interface defines that an object should have a **`void`** method **`hunt`**. The class extends **`AbstractMovableItem`** and includes a **`hunt`** method that encapsulates all the logic for pursuing prey.

By doing so, the **`AbstractMovableItem hunter`** in the **`Forest`** class could be replaced with **`AbstractHunter hunter`**, making the design more flexible and specific. This change allows for a clear separation of concerns, as it isolates the hunting logic within the **`AbstractHunter`** class while still inheriting all the movement-related functionality from **`AbstractMovableItem`**.

3.4 Satisfaction with the results

I am happy with the current state of my code, but I plan to add more features in the future, such as implementing a character selection system, allowing players to choose their character for a more personalized gameplay experience.

3.5 The hardest part

The most challenging part for me was ensuring that the application doesn't break when a user is null or when there is no home. Handling these edge cases is crucial in application development, as no one likes bugs that disrupt the gameplay. Additionally, the hunting logic was tricky due to the many different scenarios involving the player and the hunter. It was essential to account for all possible cases and make the right decisions in each situation.

3.6 Solving optional part

I have not solved the optional part yet, but would really like to finish it later together with a menu before starting the game.

3.7 TIL

Today, I learned how to work with object serialization in Java. Prior to this, I had only done it in Python, where the process is more straightforward. In addition to serialization, I also learned how to convert Java objects to JSON objects using Gson. This knowledge is crucial, as preserving the state of an application is a common practice in real-world software development.