Interfaces and Abstract Classes for

Master of Science in Information Technology

Software Design and Programming

Joseph DiBiasi, Estell Moore, James McKenna, Anthony Martuscelli

University of Denver College of Professional Studies

10/22/2025

Faculty: Nirav Shah, M.S.

Director: Cathie Wilson, M.S.

Dean: Michael J. McGuire, MLS

Abstract

Interfaces and Abstract classes are both concepts that are super useful in Object Oriented Programming. Both of these concepts are used in various languages and their use cases are very useful when trying to implement specific behavior with objects. Abstract classes are more important when sharing common code across subclasses. While interfaces are just as important as this allows us to implement all of our methods from using the interface class. This is just as important as this will tell our subclasses what that class should do but not how to do it. Both of these concepts will be utilized and shown in our Player Interface and Player abstract classes to show the differences between the two and why each is important for specific behaviors and when using them in inheritance.

CONTENTS

Introduction

Interfaces and Abstract classes are fundamental principles that are huge in Java, C++, Python and many other object oriented languages. The mechanics for abstracts to define behaviors which allow other classes to extend or implement is also a crucial component when it comes to developing code. In our Player Portfolio System, first we have a Player interface, this will allow us to have a specified behavior for specific methods that other classes must implement. For our abstract class this will contain our abstract and concrete methods, this will give a more strict behavior and allow us to have public and private properties. Our interface will allow us to use more common behaviors and code that can be supplemented to both our NBA and NFL Players, while our abstract classes will allow us to have set abstract methods specifically for their sports and their stats. Both of these concepts will allow us to demonstrate inheritance and will also define very specific and strict behavior.

Design Decisions

When implementing interfaces and abstracts in the classes, multiple designs needed to be taken into consideration. Interfaces are collections of methods that can be implemented by other classes, and serves as a grouping mechanism to access common behaviors (GeeksForGeeks. 2025.). Abstracts are classes that can be subclassed to implement the abstract methods to utilize the commonalities of code between closely related classes (Oracle. 2025.). In this assignment, we were required to write an interface and an abstract class, then implement and extend each respectively.

To outline the differences between the two, the abstract class called PlayerA was extended with the two other concrete classes called NFLPlayerAbstract and NBAPlayerAbstract. In the two sport classes, the constructor utilizes *super* to call the constructor from the PlayerA to initialize the name and stat fields from the abstract class. The approach in the interface class called Player, was to implement it in the NFLPlayer and NBAPlayer classes. These classes implement the getter methods from Player, which also supports polymorphism between the NBA versus the NFL classes.

The decision to utilize the @Override annotation was just to ensure compile-time safeties when dealing with interfaces and abstracts. This annotation ensures that methods from the interface and abstract class are overridden by the method in the subclasses. This ensures there's no duplicate code among commonalities, and gives enough flexibility for each of the other classes to manipulate how it utilizes the methods.

Assignment Difficulties

This week, the project presented the team with several challenges relating to design structure and implementation. One of the main difficulties encountered during design was how to effectively separate the functions between the interface and abstract classes. Since both parts required similar player registration, it took us some additional effort to ensure that redundancy was avoided.

Another challenge we faced was properly implementing the Player interface and PlayerA Abstract. Ensuring each subclass correctly overwrote the required methods required a really high level of attention to details. Small inconsistencies like mismatched methods did initially

cause us some issues with compilation and did require some debugging on the part of the team collectively.

Getting to the unit testing, this added a whole other layer of complexity into this week. Creating tests that validated both of our implementations proved to be quite some work. Simulation of player data was also required as we needed to confirm that each subclass returned values for name, stats, and sport.

Lastly, we did run into some issues with code structure and keeping things clean. The class design took a few iterations to nail down. This week really highlighted how difference inheritance structures can achieve similar results. Similarly, how these structures can become confusing if they are not planned accordingly. Ultimately, this week reinforced the importance of consistent naming, testing, and most importantly, design planning when working with numerous inheritance approaches.

Conclusion

While utilizing abstract classes and interfaces can be difficult to learn initially, they are well worth the effort because they can help hide complex interactions within the code base and instead expose only the points necessary for implementation. Abstract classes provide for direct inheritance and are a way of reusing methods and fields within subclasses without repeating code. These subclasses can then take on different forms and behaviors as needed, either defined within themselves or taken from their superclass. Interfaces are similar but are defined by a looser coupling of only defining behaviors that are then implemented within the class using the interface. This style of loose coupling allows for multiple interfaces to be implemented by a

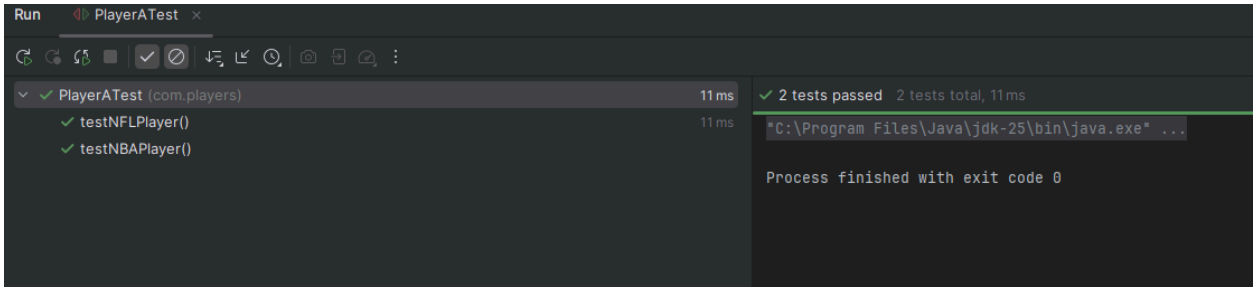
class whereas a class can only inherit one abstract class. While abstract classes and interfaces may differ in their approach to defining relationships, both are useful tools in development and aid in the crucial Object-Oriented Programming tenets of abstraction and polymorphism.

References

GeeksForGeeks. 2025. “Java Interface.” *GeeksforGeeks*. GeeksForGeeks. October 10. https://www.geeksforgeeks.org/java/interfaces-in-java/.

Oracle. 2025. “Abstract Methods and Classes.” 2025. *Abstract Methods and Classes (The JavaTM Tutorials > Learning the Java Language > Interfaces and Inheritance)*. Accessed October 26. https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html.

Appendix A: Player Abstract Unit Test Passing



Appendix B: Player Interface Unit Test Passing