C++ week 5

Khaula Molapo

1.

In a video game, abstract classes are useful for creating a base structure for different types of characters. Imagine a game with heroes, villains, and neutral characters. All of them share common features like health, attack power, and movement, but each behaves differently. An abstract class called 'Character' could define these shared attributes and declare abstract methods like 'attack()' and 'defend()'. Since 'Character' is abstract, it cannot be instantiated directly, but it ensures that all subclasses must implement the required behaviors. For example, a 'Warrior' class might implement 'attack()' with a sword, while a 'Mage' might use magic. The benefit of using an abstract class is consistency; all characters follow the same template but have freedom to implement unique actions. This structure makes the game easier to expand. Adding a new character type only requires creating a new subclass with its own behaviors.

2.

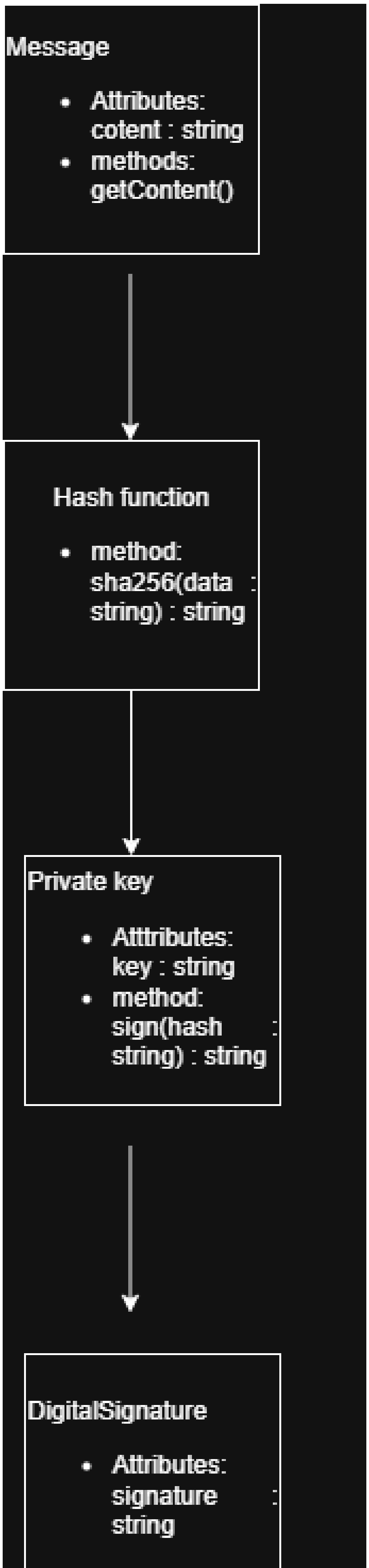
Pure virtual functions are special functions in C++ declared with '= 0', which makes a class abstract. They act as placeholders that derived classes must implement, enforcing a consistent interface across subclasses. For example, in an abstract class 'Shape', a pure virtual function 'draw()' ensures every subclass like 'Circle' or 'Rectangle' defines how it should be drawn. This mechanism is essential for polymorphism, as it allows code to use base class pointers or references while relying on subclass behavior. Pure virtual functions encourage modular design, reduce redundancy, and provide a foundation for building flexible, extensible software systems.

3.

For practice, I coded a simple interface-like structure in C++ using an abstract class. The class 'Shape' contained a pure virtual function 'draw()'. Then I created two subclasses: 'Circle' and 'Rectangle', each providing its own version of 'draw()'. Reflection: This exercise highlighted how interfaces promote flexibility. Instead of worrying about how each shape is drawn, the program can store them in an array of 'Shape*' and call 'draw()' without knowing the specific type. This approach reduces code duplication and improves maintainability. I realized that abstract classes act as powerful contracts that enforce structure while enabling customization.

4.

In Draw.io, I designed a simple UML diagram to represent the abstract class and its derived classes. At the top, the abstract class 'Character' is shown with italic text, containing attributes like 'health' and 'attackpower', plus abstract methods 'attack()' and 'defend()'. Arrows point downward to three subclasses: 'Warrior', 'Mage', and 'Archer'. Each subclass contains its own unique implementation of the abstract methods, such as 'attack with sword' or 'attack with bow'. The diagram clearly shows inheritance, with one base abstract class and multiple concrete classes. This visualization helps explain structure and relationships at a glance.


Message
Attributes: cotent : string
methods: getContent()
Hash function
method: sha256(data : string) : string
Private key
Atttributes: key : string
method: sign(hash : string) : string
DigitalSignature
Atttributes: signature : string