Interfaces vs Abstract Classes

Let's consider a scenario where you are developing a video game, and you have different types of characters in the game, such as heroes, enemies, and non-playable characters (NPCs). Each of these character types has different abilities and behaviors, but they all share a common functionality of being able to move and attack.

In this case, you could define an interface called Movable that includes methods like moveForward(), moveBackward(), moveLeft(), and moveRight(). Similarly, you could define another interface called Attackable that includes methods like attack() and defend().

By using interfaces, you can have separate classes for each type of character (e.g., Hero, Enemy, NPC), and each class can implement the appropriate interfaces based on their specific abilities. For example, a Hero class can implement both Movable and Attackable interfaces, while an Enemy class may only implement the Movable interface.

This approach provides flexibility and loose coupling in your code. It allows you to define different behaviors for each type of character while ensuring that they all adhere to the common contract defined by the interfaces. Additionally, you can easily extend or modify the behavior of any character type by implementing additional interfaces or modifying existing ones without changing the underlying class hierarchy.

On the other hand, if you were to use abstract classes instead of interfaces, you would be limited to a single inheritance hierarchy. Abstract classes are useful when you want to provide a base implementation and common functionality for a specific group of related classes. However, in scenarios where you have unrelated classes that need to share common behavior, interfaces offer a more flexible and modular solution.

Abstract Classes vs Interfaces

There are several scenarios where you would choose to use an abstract class over an interface. Here's an example:

Let's say you are building a banking application, and you have different types of accounts, such as savings accounts, checking accounts, and investment accounts. Each of these account types has some common functionality, such as depositing and withdrawing money, checking the account balance, and calculating interest. However, each account type may have its own specific implementation details.

In this case, you could define an abstract class called Account that provides a base implementation for the common functionality. The Account class can include methods like deposit(), withdraw(), getBalance(), and calculateInterest(), along with any necessary fields and additional helper methods.

Subclasses representing specific account types, such as SavingsAccount, CheckingAccount, and InvestmentAccount, can then extend the Account abstract class and provide their own implementations for any additional methods or properties specific to their respective account types. For example, the SavingsAccount class might implement a method called applyInterestRate().

By using an abstract class, you can provide a reusable implementation for common functionality, reduce code duplication, and enforce a consistent structure across different account types. Subclasses can inherit and override methods from the abstract class as needed, providing specialized behavior while ensuring adherence to the common contract defined by the abstract class.

In this scenario, using an interface wouldn't be as suitable because interfaces only define contracts without any default implementation. Abstract classes allow you to provide a default implementation for common functionality, reducing redundant code across subclasses.

Additionally, abstract classes can have instance variables and constructors, which can be useful when you need to share state or define common behavior that relies on specific fields. Interfaces, on the other hand, only define method signatures and cannot have instance variables or constructors.

To summarize, you would use an abstract class over an interface when you want to provide a base implementation for common functionality, share state or behavior across subclasses, and when you need constructors or instance variables in addition to method signatures.