Abstraction with Examples

Abstraction is a core concept in Java that helps to simplify complex systems by focusing on high-level functionalities rather than implementation details. Let's break down the idea of abstraction and how it is implemented in Java through classes and interfaces.

What is Abstraction?

Abstraction allows you to define the structure of a system without providing all the implementation details. Instead, it focuses on the essential aspects, leaving the specific implementations to be handled in more specialized classes. This concept is often used in situations where the implementation of certain methods is unknown or needs to be different in subclasses.

Example of Abstraction: A Graphic Software

Imagine you are designing a graphic software. One of the basic classes in this software would be the `Shape` class. Every shape, whether it's a circle, rectangle, or triangle, shares common features like color, name, and boundary style. Moreover, each shape needs to have methods such as `draw()`, `getArea()`, and `move()`.

However, when creating the base `Shape` class, you might not know how these methods will be implemented. For instance, you don't know how to draw a specific shape or calculate its area in the `Shape` class itself. This is where abstraction comes in.

Using Abstract Methods

To handle this situation, you can define the methods in the `Shape` class as abstract. An abstract method is one that is declared but has no implementation in the base class. By marking these methods as abstract, you indicate that the derived classes (like `Rectangle`, `Circle`, or `Triangle`) will provide their own specific implementations of these methods.

For example:

```
abstract class Shape {

   String name;

   String color;



   // Abstract methods

   abstract void draw();

   abstract double getArea();

}
```

Abstract Classes and Interfaces in Java

Java achieves abstraction primarily through abstract classes and interfaces.

1. Abstract Classes:
    - An abstract class can have both abstract methods (without implementation) and normal methods (with implementation). It serves as a blueprint for other classes.
    - You cannot create instances of abstract classes directly, but you can create objects of classes that extend these abstract classes.
2. Interfaces:
    - An interface is similar to an abstract class, but all the methods in an interface are abstract by default. In recent Java versions, interfaces can also have default methods with implementation, but they still focus on defining behavior rather than implementation.
    - Interfaces are often used to define common functionalities that can be shared across different classes.

Both abstract classes and interfaces are powerful tools for achieving abstraction, each with its own use cases. Abstract classes are ideal when you have some common

functionality shared by all subclasses, while interfaces are better suited for defining common behavior that can be implemented by classes from different hierarchies.

Real-World Example: Employee Management System

Consider an employee management software. You have a base `Employee` class, and then specific types of employees like `SDE`, `HR`, and `Manager`. In the `Employee` base class, you might define methods like `raiseSalary()` or `changeDepartment()`, but you don't know exactly how these methods will be implemented for every type of employee.

In this case, you can declare these methods as abstract in the `Employee` class. Each specific employee type will then implement these methods according to their needs. This allows you to define the structure of employee behavior while leaving the details to be implemented by subclasses.

Concrete vs Abstract Classes

- Abstract Classes: Provide a high-level blueprint with some methods declared but not implemented.
- Concrete Classes: Are fully implemented classes where methods are defined and the behavior is complete. For example, `Rectangle` or `Circle` are concrete classes that implement the abstract methods from the `Shape` class.