Abstraction in Java

Definition:
Abstraction is one of the fundamental principles of Object-Oriented Programming (OOP). It involves hiding the complex implementation details of a system and exposing only the essential features or functionalities to the user. In Java, abstraction is achieved using abstract classes and interfaces.

Why Use Abstraction?

- **Simplification**: By focusing on what an object does rather than how it does it, abstraction simplifies complex systems.
- **Security**: Hides the internal implementation, preventing external objects from accessing the internal workings of the system.
- **Modularity**: Encourages modular design, where implementation details can be changed without affecting other parts of the system.

Example: Abstraction in Java Using Abstract Classes

Let's consider an example of a payment processing system that demonstrates abstraction using an abstract class.

Scenario: Payment Processing System

In this scenario, you have different types of payment methods like Credit Card and PayPal. Each payment method has specific processing steps, but the basic operation of processing a payment is common across all payment methods.

Abstract Class: `PaymentProcessor`

- This class provides a common template for all payment methods.
- It contains an abstract method `processPayment()`, which must be implemented by any subclass.
- It also provides a non-abstract method `printReceipt()`, which is common for all payment methods.

Concrete Classes: `CreditCardPayment` and `PayPalPayment`

- These classes extend the `PaymentProcessor` abstract class and provide specific implementations for `processPayment()`.

Code Example

```
// Abstract class
abstract class PaymentProcessor {
    // Abstract method (must be implemented by subclasses)
    abstract void processPayment(double amount);

    // Concrete method (common implementation for all subclas
ses)
    void printReceipt(double amount) {
        System.out.println("Receipt: Payment of $" + amount +
" has been processed.");
    }
}

// Concrete class implementing the abstract method
class CreditCardPayment extends PaymentProcessor {
    @Override
    void processPayment(double amount) {
        System.out.println("Processing credit card payment of
$" + amount);
    }
}

// Another concrete class implementing the abstract method
class PayPalPayment extends PaymentProcessor {
    @Override
    void processPayment(double amount) {
        System.out.println("Processing PayPal payment of $" +
amount);
    }
}
```

```
public class AbstractionExample {
    public static void main(String[] args) {
        PaymentProcessor payment1 = new CreditCardPayment();
        payment1.processPayment(150.0);
        payment1.printReceipt(150.0);

        PaymentProcessor payment2 = new PayPalPayment();
        payment2.processPayment(200.0);
        payment2.printReceipt(200.0);
    }
}
```

Explanation

1. **Abstract Class `PaymentProcessor`:**
   - Contains an abstract method `processPayment(double amount)`, which defines a contract that must be fulfilled by subclasses.
   - Contains a concrete method `printReceipt(double amount)`, which provides a common functionality that can be used by all subclasses.
2. **Concrete Classes `CreditCardPayment` and `PayPalPayment`:**
   - These classes extend the `PaymentProcessor` abstract class and provide specific implementations for the `processPayment(double amount)` method.
3. **Main Class `AbstractionExample`:**
   - Demonstrates how different payment methods can be used interchangeably, thanks to the abstraction provided by the `PaymentProcessor` abstract class.
   - The concrete implementations of `processPayment()` are hidden from the user, who only interacts with the high-level `PaymentProcessor` abstraction.

Output

```
Processing credit card payment of $150.0
Receipt: Payment of $150.0 has been processed.
```

```
Processing PayPal payment of $200.0
Receipt: Payment of $200.0 has been processed.
```

Summary

Abstraction in Java allows you to define the "what" (through abstract methods) while hiding the "how" (the actual implementation) from the user. This approach simplifies complex systems by providing a clear interface for interaction while encapsulating the underlying details. In the provided example, the `PaymentProcessor` abstract class defines a common structure for different payment methods, allowing them to be used interchangeably while keeping the specific payment processing details hidden.