**Interfaces and Abstract Classes in Java**

In Java, both **interfaces** and **abstract classes** are used to achieve abstraction, but they are different in their structure and use cases.

**1. Abstract Class**

An **abstract class** is a class that cannot be instantiated directly. It can have both abstract methods (methods without implementation) and concrete methods (methods with implementation). Abstract classes are used to provide a base class with common behavior for subclasses while allowing specific implementations to be defined by subclasses.

**Key Features of Abstract Class:**

* **Abstract methods**: These methods do not have a body and must be implemented by subclasses.
* **Concrete methods**: These methods can have a body and can be inherited by subclasses.
* **Can have constructors**: Abstract classes can have constructors, which can be called by the subclass's constructor.
* **Can have instance variables**: Abstract classes can have fields and regular methods.

**Example of Abstract Class:**

Let's create a system with an abstract class Employee, and two subclasses FullTimeEmployee and PartTimeEmployee that implement specific behavior.

// Abstract class

abstract class Employee {

String name;

int age;

// Constructor

Employee(String name, int age) {

this.name = name;

this.age = age;

}

// Abstract method (no body)

abstract void calculateSalary();

// Concrete method

void displayDetails() {

System.out.println("Name: " + name);

System.out.println("Age: " + age);

}

}

// Subclass 1 - FullTimeEmployee

class FullTimeEmployee extends Employee {

double monthlySalary;

// Constructor

FullTimeEmployee(String name, int age, double monthlySalary) {

super(name, age);

this.monthlySalary = monthlySalary;

}

// Implementation of abstract method

@Override

void calculateSalary() {

System.out.println("Full-time employee salary: $" + monthlySalary);

}

}

// Subclass 2 - PartTimeEmployee

class PartTimeEmployee extends Employee {

double hourlyRate;

int hoursWorked;

// Constructor

PartTimeEmployee(String name, int age, double hourlyRate, int hoursWorked) {

super(name, age);

this.hourlyRate = hourlyRate;

this.hoursWorked = hoursWorked;

}

// Implementation of abstract method

@Override

void calculateSalary() {

System.out.println("Part-time employee salary: $" + (hourlyRate \* hoursWorked));

}

}

public class AbstractClassDemo {

public static void main(String[] args) {

Employee fullTime = new FullTimeEmployee("Alice", 30, 5000);

Employee partTime = new PartTimeEmployee("Bob", 25, 20, 80);

// Display details and calculate salary

fullTime.displayDetails();

fullTime.calculateSalary();

System.out.println();

partTime.displayDetails();

partTime.calculateSalary();

}

}

**Explanation:**

* The abstract class Employee defines an abstract method calculateSalary() which must be implemented by any subclass.
* Both FullTimeEmployee and PartTimeEmployee provide their own implementation of calculateSalary().
* The displayDetails() method is a concrete method that both subclasses inherit.

**Output:**

Name: Alice

Age: 30

Full-time employee salary: $5000.0

Name: Bob

Age: 25

Part-time employee salary: $1600.0

**2. Interface**

An **interface** is a reference type in Java, similar to a class, but it can only contain method declarations (abstract methods) and constant variables. Interfaces cannot contain concrete methods unless they are marked as default (in Java 8 and beyond). A class implements an interface by providing concrete implementations of all its methods.

**Key Features of Interface:**

* **Abstract methods**: All methods in an interface are implicitly abstract (unless they are default or static methods).
* **No constructors**: Interfaces cannot have constructors, as they cannot be instantiated directly.
* **Multiple inheritance**: A class can implement multiple interfaces, providing a form of multiple inheritance.
* **Can only contain constants and method signatures**: No instance variables are allowed, only public static final constants.

**Example of Interface:**

In this example, we create an interface Payment to represent different payment methods. Both CreditCardPayment and PayPalPayment classes will implement this interface.

// Interface

interface Payment {

// Abstract method to process payment

void processPayment(double amount);

}

// Implementing class 1 - CreditCardPayment

class CreditCardPayment implements Payment {

String cardNumber;

CreditCardPayment(String cardNumber) {

this.cardNumber = cardNumber;

}

// Implementing the abstract method from Payment interface

@Override

public void processPayment(double amount) {

System.out.println("Processing payment of $" + amount + " through Credit Card: " + cardNumber);

}

}

// Implementing class 2 - PayPalPayment

class PayPalPayment implements Payment {

String email;

PayPalPayment(String email) {

this.email = email;

}

// Implementing the abstract method from Payment interface

@Override

public void processPayment(double amount) {

System.out.println("Processing payment of $" + amount + " through PayPal account: " + email);

}

}

public class InterfaceDemo {

public static void main(String[] args) {

// Creating instances of CreditCardPayment and PayPalPayment

Payment creditCardPayment = new CreditCardPayment("1234-5678-9876-5432");

Payment payPalPayment = new PayPalPayment("user@example.com");

// Processing payments

creditCardPayment.processPayment(100.0);

payPalPayment.processPayment(150.0);

}

}

**Explanation:**

* The Payment interface defines an abstract method processPayment() which is implemented by the classes CreditCardPayment and PayPalPayment.
* CreditCardPayment and PayPalPayment provide specific implementations of the processPayment() method.

**Output:**

Processing payment of $100.0 through Credit Card: 1234-5678-9876-5432

Processing payment of $150.0 through PayPal account: user@example.com

**Key Differences Between Abstract Class and Interface**

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| **Method Type** | Can have both abstract and concrete methods | All methods are implicitly abstract |
| **Multiple Inheritance** | A class can inherit only one abstract class | A class can implement multiple interfaces |
| **Constructors** | Can have constructors | Cannot have constructors |
| **Fields** | Can have instance variables (fields) | Only constants (public static final) |
| **Method Implementation** | Can provide default behavior in concrete methods | Cannot provide method implementations |
| **Inheritance** | Used for "is-a" relationship | Used for "can-do" relationship |
| **Access Modifiers** | Methods can have any access modifier (public, private, etc.) | Methods are implicitly public |

**When to Use Abstract Class vs. Interface**

* **Use an abstract class** when you have a common base class with shared code that should be inherited. Abstract classes are suitable when:
  + You want to provide default behavior.
  + You have a base class that should not be instantiated directly.
  + You have fields and constructors in the base class.
* **Use an interface** when you want to define a contract for classes to follow without imposing any implementation. Interfaces are suitable when:
  + You need to achieve multiple inheritance.
  + You only want to specify the method signatures (a contract).
  + You want classes from different inheritance trees to share common behavior.

# ****Default and Static Methods in Interfaces: Java 8 and Beyond****

In **Java 8**, two significant features were added to interfaces: **default methods** and **static methods**. These changes brought more flexibility to interfaces, allowing them to provide method implementations directly within the interface. This allowed interfaces to have behavior (something they couldn’t do before) and made it easier to evolve interfaces while maintaining backward compatibility.

## ****1. Default Methods in Interfaces****

### ****What is a Default Method?****

A **default method** is a method in an interface that has a body (implementation) within the interface itself. This is in contrast to abstract methods, which only have method signatures without implementations. Default methods allow interfaces to provide default behavior to their implementing classes.

### ****Syntax of Default Methods:****

The default keyword is used to define a default method in an interface.

interface MyInterface {

// Abstract method

void abstractMethod();

// Default method

default void defaultMethod() {

System.out.println("This is the default method implementation.");

}

}

### ****Key Features of Default Methods:****

* **Method Implementation**: Default methods can have a body (implementation) within the interface itself.
* **Overridable**: Classes implementing the interface can override the default method to provide their own implementation if needed.
* **Backward Compatibility**: Default methods were introduced to maintain backward compatibility when adding new methods to existing interfaces. Without default methods, adding a new method to an interface would break the existing code that implements that interface.

### ****When to Use Default Methods:****

* When you want to provide a default behavior in an interface that can be shared across all implementing classes.
* When you want to add new methods to an interface without breaking the existing implementations of that interface.
* When providing common functionality in the interface that is likely to be shared across multiple classes.

### ****Example of Default Method:****

In this example, an interface Payment has a default method processTransaction() to process a payment, while the implementing class overrides it with a more specific implementation:

interface Payment {

// Abstract method

void processPayment(double amount);

// Default method

default void processTransaction() {

System.out.println("Processing transaction...");

}

}

class CreditCardPayment implements Payment {

String cardNumber;

CreditCardPayment(String cardNumber) {

this.cardNumber = cardNumber;

}

// Implementing the abstract method

@Override

public void processPayment(double amount) {

System.out.println("Processing payment of $" + amount + " through Credit Card: " + cardNumber);

}

// Optional: Override the default method if needed

@Override

public void processTransaction() {

System.out.println("Processing Credit Card transaction...");

}

}

class PayPalPayment implements Payment {

String email;

PayPalPayment(String email) {

this.email = email;

}

// Implementing the abstract method

@Override

public void processPayment(double amount) {

System.out.println("Processing payment of $" + amount + " through PayPal account: " + email);

}

// The default method is used here without overriding

}

public class DefaultMethodExample {

public static void main(String[] args) {

Payment creditCard = new CreditCardPayment("1234-5678-9876-5432");

Payment paypal = new PayPalPayment("user@example.com");

// Process payment and transaction

creditCard.processPayment(100.0);

creditCard.processTransaction(); // Overridden method in CreditCardPayment

System.out.println();

paypal.processPayment(150.0);

paypal.processTransaction(); // Default method in PayPalPayment

}

}

**Output:**

Processing payment of $100.0 through Credit Card: 1234-5678-9876-5432

Processing Credit Card transaction...

Processing payment of $150.0 through PayPal account: user@example.com

Processing transaction...

## ****2. Static Methods in Interfaces****

### ****What is a Static Method?****

A **static method** in an interface is similar to static methods in classes. It belongs to the interface itself, not to instances of classes that implement the interface. Static methods are used when you want to define utility methods or helper methods that are specific to the interface and do not depend on instance data.

### ****Syntax of Static Methods:****

Static methods are defined using the static keyword in the interface.

interface MyInterface {

// Static method

static void staticMethod() {

System.out.println("This is a static method in the interface.");

}

}

### ****Key Features of Static Methods:****

* **Belongs to the Interface**: Static methods in interfaces are accessed using the interface name.
* **Cannot Be Overridden**: Unlike instance methods (including default methods), static methods cannot be overridden by implementing classes. They belong to the interface and are accessed using the interface name.
* **Utility Methods**: Static methods are often used to provide utility methods that are related to the interface but do not depend on instance data.

### ****When to Use Static Methods:****

* When you need to define a utility or helper method that is closely related to the interface but does not rely on instance-specific data.
* When you want to provide functionality that belongs to the interface, not to any particular object implementing the interface.

### ****Example of Static Method:****

In this example, we use a static method processTransaction() in the interface Payment to demonstrate its utility function:

interface Payment {

// Abstract method

void processPayment(double amount);

// Static method

static void processTransaction() {

System.out.println("Processing a generic payment transaction...");

}

}

class CreditCardPayment implements Payment {

String cardNumber;

CreditCardPayment(String cardNumber) {

this.cardNumber = cardNumber;

}

// Implementing the abstract method

@Override

public void processPayment(double amount) {

System.out.println("Processing payment of $" + amount + " through Credit Card: " + cardNumber);

}

}

class PayPalPayment implements Payment {

String email;

PayPalPayment(String email) {

this.email = email;

}

// Implementing the abstract method

@Override

public void processPayment(double amount) {

System.out.println("Processing payment of $" + amount + " through PayPal account: " + email);

}

}

public class StaticMethodExample {

public static void main(String[] args) {

Payment.processTransaction(); // Calling the static method

Payment creditCardPayment = new CreditCardPayment("1234-5678-9876-5432");

Payment payPalPayment = new PayPalPayment("user@example.com");

// Processing individual payments

creditCardPayment.processPayment(100.0);

payPalPayment.processPayment(150.0);

}

}

**Output:**

Processing a generic payment transaction...

Processing payment of $100.0 through Credit Card: 1234-5678-9876-5432

Processing payment of $150.0 through PayPal account: user@example.com

## ****Key Differences Between Default and Static Methods in Interfaces****

| **Feature** | **Default Method** | **Static Method** |
| --- | --- | --- |
| **Belongs to** | Instance of the class implementing the interface | The interface itself |
| **Accessed by** | Object of the implementing class | Interface name |
| **Overriding** | Can be overridden by implementing classes | Cannot be overridden |
| **Use Case** | Provides default implementation for instance methods | Utility methods or helper methods specific to the interface |
| **Invocation** | Invoked through an instance of the implementing class | Invoked using the interface name |