

Java Abstract Class Cheat Sheet

What is an Abstract Class?

An abstract class is a class that cannot be instantiated and may contain abstract methods (methods without implementation). It serves as a blueprint for other classes to extend.

Basic Syntax

```
java

public abstract class ClassName {
    // Abstract method (no implementation)
    public abstract void methodName();

    // Concrete method (with implementation)
    public void concreteMethod() {
        // implementation
    }
}
```

Key Characteristics

- **Cannot be instantiated:** You cannot create objects directly from an abstract class
- **Can have abstract methods:** Methods without a body (must end with semicolon)
- **Can have concrete methods:** Regular methods with full implementation
- **Can have constructors:** Called when subclass is instantiated
- **Can have fields:** Both static and instance variables
- **Can have static methods:** Class-level methods
- **Uses `extends` keyword:** Subclasses extend the abstract class
- **Single inheritance:** A class can extend only one abstract class

Creating an Abstract Class

Simple Abstract Class

```
java
```

```

public abstract class Animal {
    // Abstract method - no implementation
    public abstract void makeSound();

    // Concrete method - has implementation
    public void sleep() {
        System.out.println("Zzz...");
    }
}

```

Abstract Class with Fields

```

java

public abstract class Shape {
    // Instance variables
    protected String color;
    protected double area;

    // Constructor
    public Shape(String color) {
        this.color = color;
    }

    // Abstract methods
    public abstract double calculateArea();
    public abstract double calculatePerimeter();

    // Concrete methods
    public String getColor() {
        return color;
    }

    public void display() {
        System.out.println("Color: " + color);
        System.out.println("Area: " + calculateArea());
    }
}

```

Extending Abstract Classes

Basic Extension

```
java
```

```
public class Dog extends Animal {  
    // Must implement all abstract methods  
    @Override  
    public void makeSound() {  
        System.out.println("Woof! Woof!");  
    }  
  
    // Can add own methods  
    public void fetch() {  
        System.out.println("Fetching ball...");  
    }  
}  
  
// Usage  
Dog dog = new Dog();  
dog.makeSound(); // Output: Woof! Woof!  
dog.sleep(); // Output: Zzz...  
dog.fetch(); // Output: Fetching ball...
```

Extension with Constructor

java

```

public class Circle extends Shape {
    private double radius;

    // Constructor calls parent constructor
    public Circle(String color, double radius) {
        super(color); // Call parent constructor
        this.radius = radius;
    }

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }

    @Override
    public double calculatePerimeter() {
        return 2 * Math.PI * radius;
    }
}

// Usage
Circle circle = new Circle("Red", 5.0);
circle.display(); // Shows color and calculated area

```

Abstract Class vs Interface

Feature	Abstract Class	Interface
Methods	Can have both abstract and concrete methods	All methods are abstract by default (until Java 8)
Variables	Can have instance variables	Only static final constants
Constructor	Can have constructors	Cannot have constructors
Access Modifiers	Can use any access modifier	Methods are public by default
Inheritance	Single inheritance (extends)	Multiple inheritance (implements)
When to Use	Related classes sharing common code	Unrelated classes sharing behavior

Common Patterns

Template Method Pattern

java

```

public abstract class Game {
    // Template method - defines algorithm structure
    public final void play() {
        initialize();
        startPlay();
        endPlay();
    }

    // Abstract methods to be implemented
    protected abstract void initialize();
    protected abstract void startPlay();
    protected abstract void endPlay();
}

public class Cricket extends Game {
    @Override
    protected void initialize() {
        System.out.println("Cricket Game Initialized");
    }

    @Override
    protected void startPlay() {
        System.out.println("Cricket Game Started");
    }

    @Override
    protected void endPlay() {
        System.out.println("Cricket Game Finished");
    }
}

// Usage
Game game = new Cricket();
game.play(); // Executes the template

```

Base Class with Common Logic

java

```
public abstract class Employee {  
    protected String name;  
    protected int id;  
    protected double baseSalary;  
  
    public Employee(String name, int id, double baseSalary) {  
        this.name = name;  
        this.id = id;  
        this.baseSalary = baseSalary;  
    }  
  
    // Abstract method - each employee type calculates differently  
    public abstract double calculateSalary();  
  
    // Concrete method - common for all employees  
    public void displayInfo() {  
        System.out.println("Name: " + name);  
        System.out.println("ID: " + id);  
        System.out.println("Salary: " + calculateSalary());  
    }  
}  
  
public class FullTimeEmployee extends Employee {  
    private double bonus;  
  
    public FullTimeEmployee(String name, int id, double baseSalary, double bonus) {  
        super(name, id, baseSalary);  
        this.bonus = bonus;  
    }  
  
    @Override  
    public double calculateSalary() {  
        return baseSalary + bonus;  
    }  
}  
  
public class PartTimeEmployee extends Employee {  
    private int hoursWorked;  
    private double hourlyRate;  
  
    public PartTimeEmployee(String name, int id, int hours, double rate) {  
        super(name, id, 0);  
        this.hoursWorked = hours;  
        this.hourlyRate = rate;  
    }  
}
```

```
@Override  
public double calculateSalary() {  
    return hoursWorked * hourlyRate;  
}  
}
```

Multiple Abstract Methods

java

```
public abstract class Vehicle {  
    protected String brand;  
    protected int year;  
  
    public Vehicle(String brand, int year) {  
        this.brand = brand;  
        this.year = year;  
    }  
  
    // Multiple abstract methods  
    public abstract void start();  
    public abstract void stop();  
    public abstract void accelerate(int speed);  
    public abstract int getCurrentSpeed();  
  
    // Concrete method  
    public void displayInfo() {  
        System.out.println(year + " " + brand);  
    }  
}  
  
public class Car extends Vehicle {  
    private int currentSpeed = 0;  
  
    public Car(String brand, int year) {  
        super(brand, year);  
    }  
  
    @Override  
    public void start() {  
        System.out.println("Car engine started");  
        currentSpeed = 0;  
    }  
  
    @Override  
    public void stop() {  
        System.out.println("Car stopped");  
        currentSpeed = 0;  
    }  
  
    @Override  
    public void accelerate(int speed) {  
        currentSpeed += speed;  
        System.out.println("Accelerating to " + currentSpeed + " km/h");  
    }  
}
```

```
@Override  
public int getCurrentSpeed() {  
    return currentSpeed;  
}  
}
```

Abstract Class Hierarchy

```
java  
// Top-level abstract class  
public abstract class Animal {  
    public abstract void makeSound();  
    public abstract void move();  
}  
  
// Mid-level abstract class  
public abstract class Mammal extends Animal {  
    // Implements some methods  
    @Override  
    public void move() {  
        System.out.println("Walking on land");  
    }  
  
    // Adds new abstract method  
    public abstract void giveBirth();  
}  
  
// Concrete class  
public class Cat extends Mammal {  
    @Override  
    public void makeSound() {  
        System.out.println("Meow!");  
    }  
  
    @Override  
    public void giveBirth() {  
        System.out.println("Giving birth to kittens");  
    }  
}
```

Static and Final Methods

```
java
```

```

public abstract class MathOperations {
    // Static method - belongs to class
    public static int add(int a, int b) {
        return a + b;
    }

    // Final method - cannot be overridden
    public final void printResult(int result) {
        System.out.println("Result: " + result);
    }

    // Abstract method
    public abstract int calculate(int a, int b);
}

public class Multiplication extends MathOperations {
    @Override
    public int calculate(int a, int b) {
        return a * b;
    }

    // Cannot override printResult() - it's final
}

// Usage
int sum = MathOperations.add(5, 3); // Static method call
Multiplication mult = new Multiplication();
int product = mult.calculate(4, 5);
mult.printResult(product);

```

Common Mistakes

✗ Cannot instantiate abstract class

```

java

// ERROR: Cannot instantiate abstract class
Animal animal = new Animal(); // Compilation error

```

✓ Correct way

```

java

// Create instance of concrete subclass
Animal animal = new Dog();

```

Forgetting to implement abstract methods

```
java

// ERROR: Must implement all abstract methods
public class Bird extends Animal {
    // Missing makeSound() implementation
} // Compilation error
```

Correct way

```
java

public class Bird extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Chirp!");
    }
}
```

Best Practices

1. Use abstract classes when:

- You want to share code among related classes
- You need constructors or instance variables
- You want to provide default behavior
- Classes have a clear "is-a" relationship

2. Naming conventions:

- Often end with "Base" or start with "Abstract" (optional)
- Use descriptive names: `AbstractShape`, `BaseController`

3. Design tips:

- Keep abstract classes focused and cohesive
- Don't make everything abstract - provide concrete methods for common logic
- Use protected access for fields that subclasses need
- Document the contract for abstract methods

4. When to use abstract class vs interface:

- **Abstract class:** Related classes with shared implementation
- **Interface:** Unrelated classes with shared contract

Complete Example

```
java
```

```
// Abstract class definition
public abstract class BankAccount {
    protected String accountNumber;
    protected String owner;
    protected double balance;

    public BankAccount(String accountNumber, String owner, double initialBalance) {
        this.accountNumber = accountNumber;
        this.owner = owner;
        this.balance = initialBalance;
    }

    // Abstract methods
    public abstract void withdraw(double amount);
    public abstract double calculateInterest();

    // Concrete methods
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Deposited: $" + amount);
        }
    }

    public double getBalance() {
        return balance;
    }

    public void displayAccountInfo() {
        System.out.println("Account: " + accountNumber);
        System.out.println("Owner: " + owner);
        System.out.println("Balance: $" + balance);
    }
}

// Savings account implementation
public class SavingsAccount extends BankAccount {
    private double interestRate;
    private double minimumBalance;

    public SavingsAccount(String accountNumber, String owner,
                         double initialBalance, double rate) {
        super(accountNumber, owner, initialBalance);
        this.interestRate = rate;
        this.minimumBalance = 100.0;
    }
}
```

```
@Override
public void withdraw(double amount) {
    if (balance - amount >= minimumBalance) {
        balance -= amount;
        System.out.println("Withdrawn: $" + amount);
    } else {
        System.out.println("Cannot withdraw: Minimum balance required");
    }
}

@Override
public double calculateInterest() {
    return balance * interestRate / 100;
}
}

// Checking account implementation
public class CheckingAccount extends BankAccount {
    private double overdraftLimit;

    public CheckingAccount(String accountNumber, String owner,
                          double initialBalance, double overdraft) {
        super(accountNumber, owner, initialBalance);
        this.overdraftLimit = overdraft;
    }

    @Override
    public void withdraw(double amount) {
        if (balance - amount >= -overdraftLimit) {
            balance -= amount;
            System.out.println("Withdrawn: $" + amount);
        } else {
            System.out.println("Exceeded overdraft limit");
        }
    }

    @Override
    public double calculateInterest() {
        return 0; // No interest on checking accounts
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        BankAccount savings = new SavingsAccount("SA001", "Alice", 1000, 3.5);
```

```

BankAccount checking = new CheckingAccount("CA001", "Bob", 500, 200);

savings.deposit(500);
savings.withdraw(200);
System.out.println("Interest: $" + savings.calculateInterest());

checking.withdraw(600); // Uses overdraft
savings.displayAccountInfo();
}

}

```

Quick Reference

java

// Declaration

```
public abstract class ClassName { }
```

// Abstract method

```
public abstract returnType methodName(parameters);
```

// Concrete method

```
public returnType methodName(parameters) {
```

// implementation

```
}
```

// Constructor

```
public ClassName(parameters) {
```

// initialization

```
}
```

// Extending

```
public class SubClass extends AbstractClass { }
```

// Implementing abstract method

```
@Override
```

```
public returnType methodName(parameters) {
```

// implementation

```
}
```