# Java Abstraction — Complete Notes

## 1. What is Abstraction?

Abstraction is an object-oriented programming concept that focuses on showing only the essential features of an object while hiding the internal implementation details. In Java, abstraction allows you to model real-world entities by exposing *what* an object does (its behavior) rather than *how* it does it.

### Key ideas

- **Focus on the interface/behavior** rather than implementation.
- **Hide complexity** from the user of a class.
- Achieved in Java using **abstract classes** and **interfaces**.

---

## 2. Why use Abstraction?

- **Simplicity:** Users interact with a simple interface instead of complex code.
- **Reusability and maintainability:** Implementation can change without affecting code that uses the abstraction.
- **Security:** Internal details can be hidden to prevent misuse.
- **Polymorphism:** Abstraction is the groundwork that enables polymorphism in OOP.

---

## 3. Abstract Class (syntax & rules)

An abstract class is declared using the `abstract` keyword and can have both abstract methods (without body) and concrete methods (with body).

```
public abstract class Vehicle {
    private String model;

    public Vehicle(String model) {
        this.model = model;
    }

    // abstract method (no body)
    public abstract void start();

    // concrete method
    public void showModel() {
        System.out.println("Model: " + model);
    }
}
```

### Rules & characteristics

- You **cannot instantiate** an abstract class directly.
- It **can have constructors**, fields, and concrete methods.
- Subclass must implement all abstract methods, otherwise the subclass must also be declared `abstract`.
- Useful when you want a common base with shared code and some methods left to subclasses.

## 4. Interface (syntax & rules)

An interface is a contract that classes can implement. It declares methods that implementing classes must provide. Since Java 8 and later, interfaces can also contain `default`, `static`, and (since Java 9) `private` methods.

```
public interface Drivable {
    void accelerate();           // implicitly public abstract

    default void horn() {        // default method with body
        System.out.println("Beep beep!");
    }

    static void maintenanceNote() { // static method
        System.out.println("Check oil regularly.");
    }
}
```

### Rules & characteristics

- Methods are `public` by default; prior to Java 9 they were implicitly `public abstract`.
- Since Java 8: `default` and `static` methods allowed; since Java 9: `private` methods allowed.
- Interfaces **cannot** have instance fields (only `public static final` constants).
- A class can implement **multiple interfaces**, enabling multiple inheritance of type.

## 5. Abstract class vs Interface (comparison)

| Feature | Abstract class | Interface |
|---|---:|---|
| Multiple inheritance | No (class can extend one class) | Yes (class can implement many interfaces) |

| Feature | Abstract class | Interface |
|---|---:|---|
| Fields | Can have instance variables (state) | Only constants (`public static final`) |
| Methods with body | Yes (concrete methods) | Yes: `default`, `static`, (and `private` since Java 9) |
| Constructor | Yes | No (interfaces have no constructors) |
| Use when | Shared code + enforced methods | Contract for behavior, multiple inheritance of type |

## 6. When to choose which?

- Use an **abstract class** when:
- You need to provide common implementation to subclasses.
- You want to share state (fields) or constructors.
- There is an `is-a` relationship and you expect subclasses to share code.

- Use an **interface** when:

- You only need to define a contract (method signatures) and possibly default behavior.
- Multiple types should share behavior, or you need multiple inheritance of type.
- You want to design to an interface for looser coupling and better testability.

## 7. Polymorphism with Abstraction

Code can be written against abstract types (abstract class or interface) allowing swapping of concrete implementations at runtime.

```
public interface Payment {
    void pay(double amount);
}

public class CreditCard implements Payment {
    public void pay(double amount) { System.out.println("Paid by card: " +
amount); }
}

public class Upi implements Payment {
    public void pay(double amount) { System.out.println("Paid via UPI: " +
amount); }
```

```java
}

// Usage
Payment p = new CreditCard();
p.pay(100);
p = new Upi();
p.pay(200);
```

## 8. Real-world analogies

- **Remote control** (interface): You press buttons (methods) but don't know how the TV processes the signal (implementation).
- **Vehicle abstract class**: All vehicles have common features like number of wheels and a method `start()` but each vehicle implements `start()` differently.

## 9. Advanced topics & Java versions

- **Java 8**: Added `default` and `static` methods in interfaces — allowed adding methods to interfaces without breaking implementing classes.
- **Java 9**: Added `private` methods inside interfaces to help share code between `default` methods.
- **Functional interfaces**: An interface with a single abstract method (SAM). Can be used with lambda expressions and method references. Marked optionally with `@FunctionalInterface`.

```java
@FunctionalInterface
public interface Calculator {
    int compute(int a, int b);
}

// lambda
Calculator add = (x, y) -> x + y;
int res = add.compute(3, 5); // 8
```

## 10. Common pitfalls & gotchas

- Forgetting to implement all abstract methods in a concrete subclass.
- Relying too heavily on abstract classes when interfaces would give better flexibility.
- Adding non-default methods to interfaces in pre-Java 8 code breaks implementing classes.
- Assuming interfaces can hold mutable instance state — they cannot (only constants).

## 11. Example: Abstract class with constructor and fields

```
abstract class Animal {
    String name;

    public Animal(String name) {
        this.name = name;
    }

    public abstract void makeSound();

    public void info() {
        System.out.println("Animal: " + name);
    }
}

class Dog extends Animal {
    public Dog(String name) { super(name); }
    public void makeSound() { System.out.println("Woof"); }
}

// Usage
Animal a = new Dog("Buddy");
a.info(); // Animal: Buddy
a.makeSound(); // Woof
```

---

## 12. Example: Interface multiple inheritance

```
interface Flyable { void fly(); }
interface Swimmable { void swim(); }

class Duck implements Flyable, Swimmable {
    public void fly() { System.out.println("Duck flying"); }
    public void swim() { System.out.println("Duck swimming"); }
}
```

---

## 13. Exam-style short points (quick revision)

- Abstraction hides implementation and shows only required details.
- Use `abstract` keyword for abstract classes and methods.
- Interfaces provide contracts; classes use `implements` to adopt them.

- Java 8 `default` and `static` methods in interfaces; Java 9 `private` methods added.
- Concrete class must implement all abstract methods.
- Abstract classes can have constructors and fields.
- Prefer interfaces for API design and flexibility; prefer abstract classes when sharing code/state.

---

## 14. Practice questions

1. Write an abstract class `Shape` with an abstract method `area()` and create `Circle` and `Rectangle` subclasses.
2. Create an interface `Logger` with a `log(String message)` method and two implementations: `ConsoleLogger` and `FileLogger`.
3. Explain how default methods in interfaces help in API evolution.
4. Convert a legacy base class design to an interface-based design and explain the pros and cons.

---

## 15. Quick reference — syntax

- Abstract class:

```java
public abstract class MyBase {
    public abstract void doSomething();
}
```

- Interface:

```java
public interface MyContract {
    void doSomething(); // implicitly public abstract
}
```

---

## 16. Further reading suggestions

- Java language specification (sections on classes and interfaces)
- Oracle Java tutorials (Object-Oriented Programming Concepts)
- Articles on SOLID principles (Interface Segregation, Dependency Inversion)

---

## 17. Summary (one-liner)

Abstraction in Java is about defining *what* operations are possible while hiding *how* those operations are implemented — achieved primarily via abstract classes and interfaces to design clean, maintainable, and flexible code.

*End of notes.*