Abstract Class

Abstract classes that share common state and behavior in Java; constructors, the Template Method pattern, and abstract class vs interface.

Intermediate 50 min

Last updated: November 9, 2026
Read online: https://www.learnforgex.com/en/topics/abstract-class

---

Abstract Class

In Java, an **abstract class** is a class that can never be instantiated directly, but that bundles together the shared state and behavior a set of subclasses will inherit and complete. Unlike an interface's "contract only" philosophy, an abstract class can say both "you must do this" (abstract methods) and "I've already done this for you" (concrete methods, fields, a constructor). In this lesson we'll go end-to-end: starting from the basic `abstract` keyword, through the role constructors play, to the real-world use of the Template Method pattern, all the way to the exact mirror image of the `Interface vs Abstract Class` comparison we saw in the Interface lesson.

What Is an Abstract Class?

An abstract class is declared by adding the `abstract` keyword to a class declaration. That single word guarantees two things at once: this class can **never be instantiated directly** (`new AbstractClass()` won't compile), and this class is allowed to contain bodyless — that is, **abstract** — methods. The only way to make an abstract class concrete is to write a subclass that `extends` it and implements every one of its abstract methods.

Why Does It Exist?

Let's start with a real problem: imagine writing `Dog`, `Cat`, and `Bird` classes separately — all three have a `name` field, a `sleep()` behavior (they all sleep the same way), and a `makeSound()` behavior (each makes a different sound). Copying `name` and `sleep()` into all three classes means both duplication and the risk of updating three places whenever a bug needs fixing. But you also can't define `makeSound()` **fully** in one shared place either — every animal makes a different sound.

An abstract class solves exactly this middle ground: you write the shared `name` and `sleep()` **once**, in the parent class; the `makeSound()` that has to be unique per subclass is left as just

a **signature**, with the body deferred to each subclass. The result: no duplicated code, yet every animal can still make its own sound.

History

Like interfaces, abstract classes have existed since Java's very first release (JDK 1.0, 1996) — as one of the fundamental building blocks of object-oriented programming (OOP), combining code sharing with abstraction from the language's earliest days. For years the line between abstract class and interface was sharp: an interface could never hold a body, an abstract class could hold both abstract and concrete methods. Java 8's (2014) addition of `default` methods to interfaces (which we covered in the Interface lesson's "History" section) blurred that line considerably — an interface could now provide a method with a body too. But as we'll see in "Abstract Class vs. Interface," some differences — like having a constructor and instance fields — remain permanent, and keep the abstract class indispensable.

Writing Your First Abstract Class

An abstract class is declared with `abstract class` instead of `class`; a bodyless method inside it is likewise marked `abstract`. A concrete subclass that extends it must provide an `@Override` for every abstract method:

```
abstract class Animal {
    protected String name;

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

    abstract void makeSound();

    void sleep() {
        System.out.println(name + " is sleeping... Zzz");
    }
}

class Dog extends Animal {
    Dog(String name) {
        super(name);
    }

    @Override
    void makeSound() {
        System.out.println(name + " says: Woof!");
    }
}

class FirstAbstractClassDemo {
    public static void main(String[] args) {
        // Animal animal = new Animal("Generic"); // compile error: Animal is abstract;
cannot be instantiated

        Dog dog = new Dog("Rex");
        dog.makeSound(); // Rex says: Woof!
        dog.sleep();     // Rex is sleeping... Zzz -- inherited straight from Animal
    }
}
```

Trying to instantiate `Animal` directly with `new Animal("Generic")` fails to compile — just like the commented-out line shows. `Dog`, on the other hand, extends `Animal` and implements `makeSound()`, so it instantiates without any trouble — and it can use `sleep()` without writing a single line of it, purely by inheriting it from `Animal`.

> An abstract class is **not required** to have even a single abstract method — a class with zero abstract methods can still be blocked from direct instantiation just by the `abstract` keyword alone; we'll see exactly that in the next section.

Abstract Class vs. Concrete Class

The most commonly confused point here: what decides whether a class can be instantiated directly is **not** whether it has an abstract method, but whether the class **itself** is marked `abstract`:

```
// Shape has ZERO abstract methods -- every method already has a body -- but
// the class itself is still marked `abstract`, so it still cannot be
// instantiated directly. It's the `abstract` keyword on the class, not the
// presence of an abstract method, that blocks `new`.
abstract class Shape {
    double area() {
        return 0;
    }
}

class Square extends Shape {
    private final double side;

    Square(double side) {
        this.side = side;
    }

    @Override
    double area() {
        return side * side;
    }
}

class AbstractVsConcreteExample {
    public static void main(String[] args) {
        // Shape shape = new Shape(); // compile error: Shape is abstract; cannot be
instantiated

        Shape shape = new Square(4);
        System.out.println(shape.area()); // 16.0
    }
}
```

`Shape` doesn't have a single abstract method — `area()` has a complete body. Even so, `Shape` can't be instantiated directly, because of the `abstract` keyword on the class declaration itself. This is a deliberate way for an API designer to say "this class should only ever be used as a base class, no one should create a direct instance of it" — even without a single abstract method.

Abstract Methods

Abstract methods in an abstract class are bodyless, just like in an interface (recall the "Abstract Methods" section in the Interface lesson) — but there's an important difference: **another abstract class that extends this one is not required to immediately implement the abstract methods it inherits**:

```
abstract class Vehicle {
    abstract void start();
}

// MotorVehicle does NOT implement start() -- and that's perfectly legal,
// because MotorVehicle itself is still declared `abstract`. Only a CONCRETE
// (non-abstract) subclass is forced to implement every inherited abstract
// method; an abstract subclass is free to defer some (or all) of them
// further down the hierarchy.
abstract class MotorVehicle extends Vehicle {
    abstract void refuel();
}

class Car extends MotorVehicle {
    @Override
    void start() {
        System.out.println("Car engine started");
    }

    @Override
    void refuel() {
        System.out.println("Car refueled with gasoline");
    }
}

class AbstractMethodExample {
    public static void main(String[] args) {
        Car car = new Car();
        car.start();
        car.refuel();
    }
}
```

Expected Output

```
Car engine started
Car refueled with gasoline
```

`MotorVehicle` doesn't implement `start()`, which it inherits from `Vehicle` — and this compiles perfectly fine, because `MotorVehicle` itself is also `abstract`. Only a **concrete** class — `Car` here — is forced to implement an abstract method, and it has to implement both `start()` and the `refuel()` it adds on its own.

> If a subclass doesn't implement an abstract method it inherited from its parent, and isn't itself marked `abstract`, you'll get a compile error. The only way to say "I'm not making this class concrete yet" is to mark the class `abstract` too — the compiler allows no third option.

Concrete Methods

Besides abstract methods, an abstract class can also hold fully-bodied — that is, **concrete** — methods; this is one of the most valuable things about abstract classes that many people don't realize:

```
abstract class Animal {
    protected String name;

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

    abstract void makeSound();

    // A CONCRETE method -- it has a body right here in the abstract class.
    // Every subclass gets this behavior for free and never has to write it
    // again, unlike makeSound(), which each subclass MUST provide itself.
    void sleep() {
        System.out.println(name + " is sleeping... Zzz");
    }
}

class Dog extends Animal {
    Dog(String name) {
        super(name);
    }

    @Override
    void makeSound() {
        System.out.println(name + " says: Woof!");
    }
}

class Cat extends Animal {
    Cat(String name) {
        super(name);
    }

    @Override
    void makeSound() {
        System.out.println(name + " says: Meow!");
    }
}

class ConcreteMethodExample {
    public static void main(String[] args) {
        Dog dog = new Dog("Rex");
        Cat cat = new Cat("Whiskers");

        dog.makeSound(); // Rex says: Woof!
        cat.makeSound();  // Whiskers says: Meow!

        // Neither Dog nor Cat wrote a single line of sleep() -- both reuse
        // the exact same inherited implementation.
        dog.sleep(); // Rex is sleeping... Zzz
        cat.sleep(); // Whiskers is sleeping... Zzz
    }
}
```

Expected Output

```
Rex says: Woof!
Whiskers says: Meow!
Rex is sleeping... Zzz
Whiskers is sleeping... Zzz
```

`sleep()` is defined with a **full body** right inside `Animal` — both `Dog` and `Cat` use it purely by inheritance, without writing a single line of it themselves. `makeSound()`, on the other hand, stays abstract, so both have to provide their own implementation. This duality — share some methods, defer others to the subclass — is exactly the core value of an abstract class we described in "Why Does It Exist?"

Fields

An abstract class, just like a regular class, can hold **instance fields** — this is one of its most fundamental differences from interfaces, which can only declare constants (recall "Constant Fields" in the Interface lesson). Subclasses typically reach these fields directly, through `protected` access:

```
abstract class Employee {
    // `protected` -- visible to subclasses (even in other packages), but not
    // part of the public API. This is the access level abstract classes
    // typically use for state that subclasses are expected to read directly.
    protected String name;
    protected double baseSalary;

    Employee(String name, double baseSalary) {
        this.name = name;
        this.baseSalary = baseSalary;
    }

    abstract double calculateSalary();
}

class Manager extends Employee {
    private final double bonus;

    Manager(String name, double baseSalary, double bonus) {
        super(name, baseSalary);
        this.bonus = bonus;
    }

    @Override
    double calculateSalary() {
        // Reads baseSalary directly -- it's inherited state, not something
        // Manager has to ask for through a getter.
        return baseSalary + bonus;
    }
}

class FieldsExample {
    public static void main(String[] args) {
        Manager manager = new Manager("Ada", 5000, 1500);
        System.out.println(manager.calculateSalary()); // 6500.0
    }
}
```

`Manager` reaches `baseSalary` directly inside `calculateSalary()`, without going through a getter — because `baseSalary` is a real **instance field** inherited from `Employee`, not merely a constant. Every instance of an `Employee` subclass (every `Manager`, every future role that might be written) gets its **own copy** of `name` / `baseSalary` — unlike interface constants, there's no single shared value here; every object has its own state.

> `protected` fields are convenient for subclasses, but they weaken encapsulation — a subclass can modify the parent's internal state directly, without going through any getter/setter. We'll cover when to prefer a `private` field plus `protected` getters/setters over a plain `protected` field in "Best Practices."

Constructors

An abstract class **can** have a constructor — and usually must, since it needs to initialize the instance fields it holds. The only difference is that this constructor can never be called directly with `new`; it only ever runs indirectly, through a `super(...)` call inside a subclass's constructor:

```
abstract class Account {
    protected String owner;
    protected double balance;

    // An abstract class CAN have a constructor -- you just can never call it
    // with `new` directly. It only ever runs as part of building a concrete
    // subclass instance, to initialize the shared state the subclass inherits.
    Account(String owner, double balance) {
        this.owner = owner;
        this.balance = balance;
        System.out.println("Account constructor running for " + owner);
    }

    abstract double interestRate();
}

class SavingsAccount extends Account {
    SavingsAccount(String owner, double balance) {
        super(owner, balance); // must be the first statement in the subclass constructor
        System.out.println("SavingsAccount constructor running");
    }

    @Override
    double interestRate() {
        return 0.05;
    }
}

class ConstructorExample {
    public static void main(String[] args) {
        SavingsAccount account = new SavingsAccount("Ada", 1000);
        // Output order proves Account's constructor runs FIRST:
        // Account constructor running for Ada
        // SavingsAccount constructor running
        System.out.println(account.interestRate()); // 0.05
    }
}
```

The first line of `SavingsAccount`'s constructor is `super(owner, balance)` — this is a **mandatory rule** in Java: the first statement of every subclass constructor (even if not written explicitly) always calls one of the parent class's constructors. Notice the order in the output: `Account`'s constructor runs **before** `SavingsAccount`'s — the parent's state has to be fully set up before the subclass does any of its own additional work.

> If you never write `super(...)` in a subclass constructor, the compiler implicitly tries to call the parent's **no-argument** constructor. If the parent (like `Account` here) doesn't have a no-argument constructor, that implicit call fails, and the subclass is forced to call `super(...)` explicitly with the correct arguments.

Inheritance

A class can `extends` only **one** class (abstract or not) — this is the exact same single-inheritance restriction we covered in the Interface lesson's "Why Does It Exist?" Once you write `Dog extends Animal`, `Dog` has fully used up its one chance to inherit from a class — but it's still free to `implements` as many interfaces as it likes (we'll do exactly that in "An Abstract Class Implementing an Interface"). Abstract class hierarchies can also span multiple levels, as we saw with `Vehicle` → `MotorVehicle` → `Car` in "Abstract Methods" — each level is free to defer part of the abstraction further down.

Overriding Abstract Methods and Polymorphism

The `@Override` annotation isn't mandatory when implementing an abstract method, but it's strongly recommended — it lets the compiler immediately catch a mistake if you get the signature wrong (say, the wrong parameter type). When several subclasses override the same abstract method differently, you get the same **polymorphism** we saw with interfaces:

```
abstract class Animal {
    protected String name;

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

    abstract void makeSound();
}

class Dog extends Animal {
    Dog(String name) {
        super(name);
    }

    @Override
    void makeSound() {
        System.out.println(name + " says: Woof!");
    }
}

class Cat extends Animal {
    Cat(String name) {
        super(name);
    }

    @Override
```

```
    void makeSound() {
        System.out.println(name + " says: Meow!");
    }
}

class Bird extends Animal {
    Bird(String name) {
        super(name);
    }

    @Override
    void makeSound() {
        System.out.println(name + " says: Tweet!");
    }
}

class OverridingAndPolymorphismExample {
    public static void main(String[] args) {
        // Polymorphism through the abstract type reference: the loop below
        // never needs to know whether each element is a Dog, a Cat, or a
        // Bird -- it only knows it's holding an Animal.
        Animal[] animals = { new Dog("Rex"), new Cat("Whiskers"), new Bird("Tweety") };

        for (Animal animal : animals) {
            animal.makeSound();
        }
        // Rex says: Woof!
        // Whiskers says: Meow!
        // Tweety says: Tweet!
    }
}
```

Expected Output

```
Rex says: Woof!
Whiskers says: Meow!
Tweety says: Tweet!
```

`Dog`, `Cat`, and `Bird` each override `makeSound()` differently — the loop inside `OverridingAndPolymorphismExample` only ever sees each object as an `Animal`, calling `makeSound()` without ever knowing which concrete class it's holding. Which implementation actually runs is decided at runtime based on the object's real class (dynamic dispatch), exactly like the `Shape` example in the Interface lesson.

Modifier and Access Rules for Abstract Methods

An abstract method's access modifier can be `public` or `protected`, but it can **never be `private`** — and `abstract` can never be combined with a handful of other keywords:

```
abstract class Shape {
    public abstract double area();          // legal: public abstract

    protected abstract double perimeter();  // legal: protected abstract

    // private abstract double helper();
    // ILLEGAL: a private method can never be overridden, which directly
    // contradicts what `abstract` requires (a subclass MUST override it).

    // static abstract double unit();
    // ILLEGAL: static methods aren't resolved polymorphically (no dynamic
    // dispatch), so they can't be "overridden" the way abstract requires.

    // abstract final double diagonal();
    // ILLEGAL: `final` forbids overriding, `abstract` requires it --
    // a direct contradiction.
}

// abstract final class Circle { }
// ILLEGAL for the exact same reason: a class can't be simultaneously
// "must be extended to be useful" (abstract) and "can never be extended"
// (final) at the same time.

class Square extends Shape {
    private final double side;

    Square(double side) {
        this.side = side;
    }

    @Override
    public double area() {
        return side * side;
    }

    @Override
    protected double perimeter() {
        return 4 * side;
    }
}

class ModifierRulesExample {
    public static void main(String[] args) {
        Shape shape = new Square(3);
        System.out.println(shape.area()); // 9.0

        // shape.perimeter();
        // Not accessible here: perimeter() is `protected`, and this call
        // site is outside Shape's package and outside any subclass of Shape.
    }
}
```

Every one of these bans directly contradicts what `abstract` itself means: `abstract` **requires** a method to be overridden; a `private` method is already invisible to every subclass (so it can't be overridden); a `static` method isn't resolved polymorphically (you'd be hiding it, not

overriding it); and a `final` method is explicitly forbidden from being overridden. All three flatly contradict "must be overridable," so writing them next to `abstract` is an immediate compile error — the exact same contradiction applies at the class level too, with `abstract final class`.

> The error message you get for trying `private abstract` or `static abstract` ("illegal combination of modifiers") can be confusing at first glance — but the cause is always one of the contradictions above. When you see it, first ask which modifier is blocking overridability.

An Abstract Class Implementing an Interface

An abstract class can connect to one (or several) interfaces with `implements` — and the exact same rule from the Interface lesson applies here too: **only subclasses are required to be concrete**, the abstract class itself doesn't have to implement the interface's methods right away:

```
interface Auditable {
    String auditLog();
}

// An abstract class that implements an interface is NOT required to provide
// bodies for that interface's methods -- it can leave auditLog() unimplemented
// and defer it to its own concrete subclasses, exactly the way it defers its
// own abstract methods like content().
abstract class Document implements Auditable {
    protected String title;

    Document(String title) {
        this.title = title;
    }

    abstract String content();
    // auditLog() from Auditable is inherited as still-abstract here --
    // Document compiles fine without implementing it.
}

class Report extends Document {
    Report(String title) {
        super(title);
    }

    @Override
    String content() {
        return "Q3 sales figures...";
    }

    @Override
    public String auditLog() {
        return "Report '" + title + "' was accessed";
    }
}

class AbstractImplementsInterfaceExample {
    public static void main(String[] args) {
        Document document = new Report("Q3 Report");
        System.out.println(document.content());  // Q3 sales figures...
        System.out.println(document.auditLog());  // Report 'Q3 Report' was accessed
    }
}
```

Expected Output

```
Q3 sales figures...
Report 'Q3 Report' was accessed
```

`Document` implements `Auditable` but never writes `auditLog()` — just like it defers its own abstract method `content()`, it defers `auditLog()` to its subclass (`Report`). `Report` has to implement both: `Document`'s own abstract method `content()`, and the `auditLog()` it inherited from `Auditable` — the compiler makes no distinction between the two, both just pile up in the same "unimplemented abstract method" list.

Abstract Class vs. Interface

Even though default methods have brought interfaces much closer to abstract classes since Java 8 (as mentioned in the Interface lesson's "History"), the permanent differences between them can be summarized as follows:

- **Constructor:** An abstract class has one; an interface can never have one.
- **Instance fields (mutable state):** An abstract class has them; an interface can only declare `public static final` constants.
- **Multiple inheritance:** A class can `extends` only one abstract class, but can `implements` as many interfaces as it likes.
- **Method access modifiers:** An abstract class's concrete methods can be `public` / `protected` / `private`; interface methods are implicitly always `public`.
- **Purpose:** An abstract class exists to share the implementation of closely related types (an "is-a" relationship); an interface exists to offer a contract that even completely unrelated types can conform to (a "can-do" relationship).

The practical rule of thumb: if what you're sharing is **state and/or shared implementation** (genuinely related types — like an `Animal` hierarchy), choose an abstract class; if what you're sharing is only a **capability contract** (you want even completely unrelated types to be able to conform — like `Comparable`, `Auditable`), choose an interface. Using **both together** is entirely normal, too — that's exactly what we did with `Document implements Auditable` in the previous section, and we'll build on it with a larger example in "Appendix: Mini Project — Combining an Abstract Class and an Interface (Payment Processor)."

> When designing a class hierarchy, ask yourself: "Might some completely unrelated type, tomorrow, also want to conform to this contract?" If the answer is yes (like `Serializable`, `Comparable`), reach for an interface; if the answer is "no, this only makes sense as a member of my `Animal` family," reach for an abstract class.

Template Method Pattern

The most classic and most widely used design pattern built on abstract classes is **Template Method**: a parent class defines the **fixed skeleton** of an algorithm (the order of its steps), leaving some (or all) of the steps abstract and deferred to subclasses:

```
abstract class DataProcessor {
    // The "template method" -- it defines the FIXED skeleton of the
    // algorithm, marked `final` so no subclass can ever change the order
    // (or skip a step); only the individual steps below are customizable.
    final void process() {
        validate();
        transform();
        save();
    }

    abstract void validate();

    abstract void transform();

    // A default step with a body -- subclasses may override it if they need
    // to, but most won't have to.
    void save() {
        System.out.println("Saved.");
    }
}

class CsvProcessor extends DataProcessor {
    @Override
    void validate() {
        System.out.println("Validating CSV structure...");
    }

    @Override
    void transform() {
        System.out.println("Transforming CSV rows...");
    }
}

class TemplateMethodExample {
    public static void main(String[] args) {
        DataProcessor processor = new CsvProcessor();
        processor.process();
        // Validating CSV structure...
        // Transforming CSV rows...
        // Saved.
    }
}
```

Expected Output

```
Validating CSV structure...
Transforming CSV rows...
Saved.
```

Notice that `process()` is marked `final` — this is a deliberate design decision: `CsvProcessor` (or any future `DataProcessor` subclass) can never change the **order** of the steps, only fill in the **content** of `validate()` and `transform()`. `save()`, meanwhile, isn't abstract — it's a default step with a body (playing a role very similar to a `default` interface method): a subclass can override it if it needs to, or leave it alone entirely.

> Template Method is a concrete example of what's sometimes called the "Hollywood Principle": "Don't call us, we'll call you." Control of the flow (when and in what order `process()` runs) stays with the parent class; the subclass only steps in at the specific points the parent calls (`validate()` , `transform()`) — it never manages the flow itself.

Real-World Use Cases

Abstract classes are a common design tool inside the JDK itself — especially throughout the collections framework:

```
import java.util.AbstractList;

// java.util.AbstractList is a real, JDK-shipped abstract class: implement
// just get(int) and size(), and it gives you a fully working, read-only
// List implementation -- iterator(), contains(), indexOf(), toString(),
// even the for-each loop -- all built on top of those two methods.
class ReadOnlyRange extends AbstractList<Integer> {
    private final int start;
    private final int end;

    ReadOnlyRange(int start, int end) {
        this.start = start;
        this.end = end;
    }

    @Override
    public Integer get(int index) {
        return start + index;
    }

    @Override
    public int size() {
        return end - start;
    }
}

class ReadOnlyListExample {
    public static void main(String[] args) {
        ReadOnlyRange range = new ReadOnlyRange(5, 10);

        System.out.println(range);              // [5, 6, 7, 8, 9] -- toString() came for
free
        System.out.println(range.contains(7));   // true -- contains() came for free too

        for (int n : range) { // the for-each loop came for free as well
            System.out.print(n + " ");
        }
        System.out.println();
    }
}
```

Expected Output

```
[5, 6, 7, 8, 9]
true
5 6 7 8 9
```

`java.util.AbstractList` is a real, JDK-shipped abstract class: implement only `get(int)` and `size()`, and `iterator()`, `contains()`, `indexOf()`, `toString()`, even for-each loop support all come **for free** — all built once, inside `AbstractList`, on top of those two methods. `java.util.AbstractMap` and `java.util.AbstractQueue` follow the same philosophy: the JDK writes the complex part of the collection interfaces (`List`, `Map`, `Queue`) once, as an abstract class, and leaves you to implement only a handful of core methods.

The same pattern shows up frequently in the Spring framework too — base classes like `AbstractController` take on the fixed part of the HTTP request-handling flow (logging, error handling, writing the response), much like a Template Method, leaving you to fill in only the single method containing your actual business logic.

Best Practices

- If you want a class to be used **only as a base class**, mark it `abstract` even if it has no abstract methods at all (see "Abstract Class vs. Concrete Class").
- In the Template Method pattern, make the method that defines the algorithm's fixed skeleton `final` — let subclasses fill in the steps, not reorder them (see "Template Method Pattern").
- Prefer a `private` field plus `protected` getters/setters over a plain `protected` field where possible — this prevents subclasses from unexpectedly corrupting the parent's internal state (see the warning in "Fields").
- If what you're sharing is only a contract (no state, no shared implementation), prefer an interface over an abstract class (see "Abstract Class vs. Interface").
- Design an abstract class's constructor knowing it will only ever be called from subclasses — a package-private or `protected` constructor usually communicates that intent better than a `public` one.

Common Mistakes

1. Assuming a class doesn't need `abstract` just because it has no abstract methods. Whether it has abstract methods is irrelevant if you want to block direct instantiation — the `abstract` keyword alone is enough (see "Abstract Class vs. Concrete Class").

2. Assuming a mid-level abstract class must immediately implement every abstract method it inherits. Only a concrete class is bound by that requirement; an abstract class is free to defer abstract methods further down the hierarchy (see "Abstract Methods").

3. Forgetting `super(...)` in a subclass constructor and assuming the parent has a no-argument constructor. If the parent's only constructor takes parameters, failing to call `super(...)` explicitly with the right arguments is a compile error (see the tip in "Constructors").

4. Trying to combine an abstract method with `private`, `static`, or `final`. All three directly contradict overridability, and none can be written alongside `abstract` (see "Modifier and Access Rules for Abstract Methods").

5. Reaching for an abstract class "just in case," knowing there will only ever be one concrete subclass. If there's no genuinely shared state/implementation, and no second subclass is expected, you've added an unnecessary layer of abstraction — a classic violation of the YAGNI ("you aren't gonna need it") principle.

6. Forgetting to mark the Template Method's skeleton method `final`, letting a subclass (accidentally or deliberately) change the order of the steps. This makes the pattern fragile — `final` is the only thing that guarantees the ordering (see "Template Method Pattern").

Summary, Cheat Sheet, and Glossary

The abstract class has been one of Java's fundamental OOP tools since JDK 1.0, combining shared state and behavior with abstraction. Key takeaways:

- Once the `abstract` keyword is added to a class, that class **can never be instantiated directly**, whether or not it has any abstract methods
- An abstract class can hold both abstract (bodyless) and concrete (bodied) methods; only **subclasses**, not intermediate abstract classes, are required to be concrete
- Unlike an interface, an abstract class can hold **instance fields** and a **constructor** — this is the most permanent difference between the two
- The first statement of a subclass constructor always (explicitly or implicitly) calls the parent's constructor — the parent's state is set up **before** the subclass's own
- `abstract` can never be combined with `private` / `static` / `final` — all three contradict overridability
- An abstract class can implement an interface, and can defer that interface's methods to its own subclasses too
- Template Method pattern: define a fixed step order in a `final` method, leave the content of the steps to abstract/overridable methods
- The JDK itself (`AbstractList`, `AbstractMap`, `AbstractQueue`) and Spring (`AbstractController`) use this pattern heavily

Quick reference:

```
// Basic definition
abstract class Animal {
    protected String name;                  // instance field -- interfaces can't have this

    Animal(String name) {                    // constructor -- interfaces can't have this
        this.name = name;
    }

    abstract void makeSound();                // abstract method -- no body

    void sleep() {                            // concrete method -- has a body
        System.out.println(name + " sleeping");
    }
}

class Dog extends Animal {
    Dog(String name) { super(name); }         // super() is always the first statement

    @Override
    void makeSound() { System.out.println("Woof!"); }
}

// Template Method
abstract class Pipeline {
    final void run() {                        // order is fixed -- final
        step1();
        step2();
    }
    abstract void step1();
    abstract void step2();
}

// Abstract class + interface together
interface Auditable { String auditLog(); }
abstract class Document implements Auditable {
    // No need to implement auditLog() here -- deferred to a subclass
    abstract String content();
}
```

Glossary

Abstract class — A class that can never be instantiated directly, shares common state /behavior, and imposes a contract on its subclasses through abstract methods.

Abstract method — A bodyless method marked `abstract`, which must be implemented by some concrete subclass.

Concrete method/class — A method with a full body; or a class that has implemented all of its abstract methods and can be instantiated directly.

`super(...)` — The call from a subclass constructor to the parent class's constructor; mandatory as the (explicit or implicit) first statement of every subclass constructor.

Template Method pattern — A design pattern that defines an algorithm's fixed step order in a `final` method, leaving the content of the steps to abstract/overridable methods.

Dynamic dispatch — Deciding, at runtime rather than compile time, which implementation of a method call actually runs, based on the object's real class.

`java.util.AbstractList` / `AbstractMap` / `AbstractQueue` — Real-world, JDK-shipped abstract classes in the collections framework that grant a full implementation once you implement just a handful of core methods.

Appendix: Mini Project — A Report-Generation Pipeline with Template Method

Let's take what we learned in "Template Method Pattern" into a real scenario: different report types (sales, inventory) all go through the same four steps (`validate` → `process` → `save` → `log`), but the content of each step varies by report type:

```
// A reusable Template Method skeleton: every report goes through the same
// four fixed steps, in the same fixed order -- only validate() and
// process() are mandatory for a subclass to fill in; save() and log() are
// optional hooks with sensible defaults.
abstract class ReportPipeline {
    final void run() {
        validate();
        process();
        save();
        log();
    }

    abstract void validate();

    abstract void process();

    void save() {
        System.out.println("[save] report written to disk");
    }

    void log() {
        System.out.println("[log] pipeline finished for " + getClass().getSimpleName());
    }
}
```

```
class SalesReportPipeline extends ReportPipeline {
    @Override
    void validate() {
        System.out.println("[validate] checking sales data...");
    }

    @Override
    void process() {
        System.out.println("[process] aggregating sales totals...");
    }
    // save() and log() are not overridden -- both use ReportPipeline's defaults.
}

class InventoryReportPipeline extends ReportPipeline {
    @Override
    void validate() {
        System.out.println("[validate] checking inventory counts...");
    }

    @Override
    void process() {
        System.out.println("[process] computing stock levels...");
    }

    @Override
    void log() {
        // This pipeline needs a different log step -- and since log() was a
        // concrete (not abstract) hook, overriding it is optional, not required.
        System.out.println("[log] inventory pipeline audited and archived");
    }
}

class ReportPipelineDemo {
    public static void main(String[] args) {
        new SalesReportPipeline().run();
        // [validate] checking sales data...
        // [process] aggregating sales totals...
        // [save] report written to disk
        // [log] pipeline finished for SalesReportPipeline

        System.out.println("---");

        new InventoryReportPipeline().run();
        // [validate] checking inventory counts...
        // [process] computing stock levels...
        // [save] report written to disk
        // [log] inventory pipeline audited and archived
    }
}
```

Expected Output

```
[validate] checking sales data...
[process] aggregating sales totals...
[save] report written to disk
[log] pipeline finished for SalesReportPipeline
---
[validate] checking inventory counts...
[process] computing stock levels...
[save] report written to disk
[log] inventory pipeline audited and archived
```

`SalesReportPipeline` only fills in the two mandatory steps (`validate`, `process`), using the defaults for `save()` / `log()`. `InventoryReportPipeline` fills in the same two mandatory steps but also overrides `log()` for its own needs — since `save()` isn't abstract, that override is entirely optional. Neither can ever change `run()` itself: the order of the steps, as emphasized in "Template Method Pattern," is always the same, thanks to `final`.

> In a real report pipeline, `save()` would likely write to a database or a file system — we deliberately simplified it to `System.out.println` in this mini project so the real point (the step order being fixed, which steps are mandatory vs. optional) stays clear.

Appendix: Mini Project — Combining an Abstract Class and an Interface (Payment Processor)

The final mini project combines the ideas from "An Abstract Class Implementing an Interface" and "Abstract Class vs. Interface": the `PaymentProcessor` abstract class carries both the shared state/algorithm (the abstract class's job) and the `Auditable` contract (the interface's job) at once:

```
interface Auditable {
    String auditTrail();
}

// Combines an abstract class (shared state + a fixed charging algorithm)
// with an interface (an unrelated "can-do" contract, Auditable) --
// PaymentProcessor implements Auditable but, like Document in
// AbstractImplementsInterfaceExample.java, leaves it to its own concrete
// subclasses to actually provide auditTrail().
abstract class PaymentProcessor implements Auditable {
    protected double totalProcessed = 0;

    // final: every processor charges the exact same way -- amount plus
    // whatever fee its own calculateFee() decides on -- only the fee
    // formula itself varies per processor.
    final double charge(double amount) {
        double fee = calculateFee(amount);
        totalProcessed += amount;
        return amount + fee;
    }

    abstract double calculateFee(double amount);
}

class CreditCardProcessor extends PaymentProcessor {
    @Override
    double calculateFee(double amount) {
        return amount * 0.029;
    }

    @Override
    public String auditTrail() {
        return "CreditCard: total processed = " + totalProcessed;
    }
}

class BankTransferProcessor extends PaymentProcessor {
    @Override
    double calculateFee(double amount) {
        return 1.50; // flat fee, regardless of amount
    }

    @Override
    public String auditTrail() {
        return "BankTransfer: total processed = " + totalProcessed;
    }
}
```

```
class PaymentProcessorDemo {
    public static void main(String[] args) {
        PaymentProcessor creditCard = new CreditCardProcessor();
        System.out.println(creditCard.charge(100)); // 102.9
        System.out.println(creditCard.auditTrail()); // CreditCard: total processed = 100.0

        PaymentProcessor bankTransfer = new BankTransferProcessor();
        System.out.println(bankTransfer.charge(500)); // 501.5
        System.out.println(bankTransfer.auditTrail()); // BankTransfer: total processed =
500.0

        // Both are used purely through the PaymentProcessor/Auditable
        // contracts here -- neither variable's declared type ever mentions
        // CreditCardProcessor or BankTransferProcessor directly.
    }
}
```

It's no accident that `charge(...)` is `final` — every payment processor totals the amount with the fee returned by `calculateFee(...)` in **exactly the same way**; only the fee-calculation **formula** varies from processor to processor (another face of the Template Method pattern). `auditTrail()`, on the other hand, comes from `Auditable` and is never implemented by `PaymentProcessor` itself — just like with `Document`/`auditLog()`, that responsibility is deferred directly to the concrete subclasses (`CreditCardProcessor`, `BankTransferProcessor`).

> Both variables in `PaymentProcessorDemo` are typed as `PaymentProcessor` — neither one mentions `CreditCardProcessor` or `BankTransferProcessor`. This shows that the "program to a contract, not an implementation" principle from the Interface lesson's "Why Does It Exist?" holds just as well in an example where an abstract class and an interface are used **together**.