a1c # java-fundamentals # oop

Inheritance Hierarchies: The Three-Tier Pattern

TCSS 305 Programming Practicum

This guide explains why professional Java applications use a three-tier inheritance pattern: Interface, Abstract Class, and Concrete Class. You will learn what belongs at each level, how to make design decisions about code placement, and how this pattern balances code reuse with flexibility.

Related Guides

- Interface Contracts — Deeper exploration of interfaces as contracts, including preconditions, postconditions, and defensive coding
- Sealed Types and Records — Understanding the `sealed` keyword and `permits` clause used in this hierarchy

---

Why Three Tiers?

When designing object-oriented systems, you face a fundamental tension: **flexibility versus code reuse**.

- **Interfaces** give maximum flexibility but no code reuse
- **Concrete classes** give maximum code reuse but minimal flexibility
- **Abstract classes** provide a middle ground

The three-tier pattern combines all three to get the best of each world.

The Pattern at a Glance

```
Interface (contract)
    └── Abstract Class (shared implementation)
            ├── Concrete Class A (specific behavior)
            └── Concrete Class B (specific behavior)
```

Each tier has a specific responsibility.

Tier	Responsibility	Contains
Interface	Defines WHAT	Method signatures, constants, default methods
Abstract Class	Implements HOW (partially)	Shared fields, shared methods, template methods
Concrete Class	Implements HOW (completely)	Specific behavior, final implementations

Important

This pattern is not theoretical. You will implement it directly in Assignment 1c with the `Item` interface, `AbstractItem` abstract class, and `StoreItem` / `StoreBulkItem` concrete classes.

Tier 1: The Interface (WHAT)

An interface defines the **contract**. It specifies what operations a type supports without dictating how those operations are implemented.

What Belongs in an Interface

- Method signatures (required behavior)
- Constants (values all implementations share)
- Default methods (optional convenience methods with default behavior)
- Static methods (utility methods related to the type)

Note

Default methods and static methods in interfaces are more advanced features that we won't work with early in the quarter, if at all. Focus on method signatures and constants for now.

The Item Interface Example

```
/**
 * Represents a single item for sale in the bookstore.
```

```
 */
public sealed interface Item permits AbstractItem {

    /**
     * Returns the name for this Item.
     *
     * @return the name for this Item
     */
    String getName();

    /**
     * Returns the unit price for this Item.
     *
     * @return the unit price for this Item
     */
    BigDecimal getPrice();

    /**
     * Calculates the total price for the given quantity.
     *
     * @param quantity the number of items
     * @param useMembershipPricing whether to apply membership discounts
     * @return the total price
     * @throws IllegalArgumentException if quantity is negative
     */
    BigDecimal calculateTotal(int quantity, boolean useMembershipPricing);

    /**
     * Returns a formatted description suitable for display.
     *
     * @return a formatted description (e.g., "Pen, $2.00")
     */
    String getFormattedDescription();
}
```

Notice what the interface does NOT contain:

- Field declarations for name or price
- Implementation of any methods
- Details about how formatting works
- Information about bulk pricing

The interface says "an Item can tell you its name, price, total, and formatted description." It does not say HOW any of this happens.

> **Tip**
>
> When designing an interface, ask: "What must EVERY implementation be able to do?" Only include operations that apply universally to all implementations.

Tier 2: The Abstract Class (HOW, Partially)

An abstract class provides **shared implementation**. It implements the parts of the interface that are common to all concrete classes.

What Belongs in an Abstract Class

- Fields shared by all subclasses
- Constructor(s) that initialize shared state
- Method implementations that are identical across subclasses
- Protected helper methods and constants for subclass use
- Abstract methods that subclasses must implement differently

The AbstractItem Example

```
/**
 * Provides common functionality for all Item implementations.
 */
public sealed abstract class AbstractItem implements Item
        permits StoreItem, StoreBulkItem {

    /** Currency formatter for US locale. */
    protected static final NumberFormat CURRENCY_FORMAT =
            NumberFormat.getCurrencyInstance(Locale.US);

    // Private fields for name and price go here

    /**
     * Constructs an AbstractItem with the specified name and price.
     *
     * @param theName the name of this item
     * @param thePrice the unit price of this item
     * @throws NullPointerException if theName or thePrice is null
     * @throws IllegalArgumentException if theName is empty or thePrice is
negative
     */
    protected AbstractItem(final String theName, final BigDecimal thePrice) {
        // Validation and field assignment - you implement this
    }

    @Override
    public String getName() {
        // Return the item's name
    }

    @Override
    public BigDecimal getPrice() {
        // Return the item's price
    }
```

```
    // calculateTotal() and getFormattedDescription() are NOT implemented here
    // because they differ between StoreItem and StoreBulkItem
}
```

Notice the class declaration:

```
public sealed abstract class AbstractItem implements Item
        permits StoreItem, StoreBulkItem
```

The `sealed` modifier combined with `permits` creates a closed hierarchy. Only `StoreItem` and `StoreBulkItem` can extend `AbstractItem`—the compiler enforces this restriction. This design decision reflects that the bookstore has exactly two pricing models: simple pricing and bulk pricing.

Key Design Decisions

Why Is the Constructor Protected?

The constructor is `protected` because:

1. Abstract classes cannot be instantiated directly
2. Only subclasses should call this constructor
3. `protected` allows subclass access while preventing external instantiation

```
// This would be an error - can't instantiate abstract class
Item item = new AbstractItem("Pen", new BigDecimal("2.00"));  // Compile
error!

// Subclasses call super() to use the protected constructor
public class StoreItem extends AbstractItem {
    public StoreItem(String name, BigDecimal price) {
        super(name, price);  // Calls protected AbstractItem constructor
    }
}
```

Why Is CURRENCY_FORMAT Protected Static Final?

- `protected` - subclasses need access for formatting
- `static` - one formatter shared by all instances (no need to create per-instance)
- `final` - the formatter reference should never change

Why Implement getName() and getPrice() Here?

Every item type returns name and price the same way. There is no variation. Implementing these in `AbstractItem` means:

- Less code in each concrete class

- Guaranteed consistent behavior
- Single point of maintenance

Why NOT Implement calculateTotal() Here?

Different item types calculate totals differently:

- `StoreItem`: simple multiplication (price x quantity)
- `StoreBulkItem`: complex logic with bulk discounts

Since the algorithm differs, the method stays abstract (not implemented in `AbstractItem`).

> **Note**
>
> A method belongs in the abstract class if and only if ALL subclasses would implement it identically. If there is ANY variation, leave it for the concrete classes.

Private Fields and Child Class Access

A common source of confusion: if `AbstractItem` declares `myName` as `private`, how do subclasses like `StoreItem` access it?

The Rule: Inherited but Not Accessible

When a parent class declares a field as `private`:

- **The field exists** in every child object (memory is allocated for it)
- **The field is not accessible** directly by the child's code
- **The child must use accessors** (`getName()`, `getPrice()`) to read the values

```
public final class StoreItem extends AbstractItem {

    @Override
    public String toString() {
        // WRONG: Cannot access private field from parent
        return "StoreItem[name='" + myName + "']";  // Compile error!

        // CORRECT: Use the accessor method
        return "StoreItem[name='" + getName() + "']";  // Works!
    }

    @Override
    public boolean equals(Object obj) {
        // ...
        // Use getName() and getPrice(), not myName and myPrice
        return Objects.equals(getName(), other.getName())
                && Objects.equals(getPrice(), other.getPrice());
```

```
    }
}
```

Why Not Use Protected Fields?

You might wonder: "Why not just make `myName` and `myPrice` protected so subclasses can access them directly?"

This is tempting but breaks encapsulation. Consider the problems:

Problem	Explanation
Implementation lock-in	If `AbstractItem` stores price as `BigDecimal myPrice`, every subclass depends on that exact representation. Changing to store cents as a `long` would break all subclasses.
No validation on access	With direct field access, you cannot add logging, lazy initialization, or computed values later.
Equivalent to public	Any class that extends yours (even malicious ones in an unsealed hierarchy) gets full access. Protected is only marginally better than public.

By keeping fields `private` and providing accessors, the parent class maintains control:

```
// Parent can change internal representation without breaking children
public abstract class AbstractItem {
    private long myPriceInCents;  // Changed from BigDecimal!

    public BigDecimal getPrice() {
        // Children still call getPrice() and get a BigDecimal
        return BigDecimal.valueOf(myPriceInCents, 2);
    }
}
```

> **The Principle**
>
> **Private fields + public/protected accessors** is the standard pattern. Use `protected` for methods that subclasses need to call or override, but keep fields `private`. This preserves encapsulation while enabling inheritance.

Practical Impact in Your Code

When implementing `toString()`, `equals()`, and `hashCode()` in `StoreItem` or `StoreBulkItem`:

- Use `getName()` and `getPrice()` to access inherited state
- Use your own fields directly (e.g., `myBulkQuantity` in `StoreBulkItem`)
- Never try to access `myName` or `myPrice` directly—the compiler will stop you anyway

---

Tier 3: The Concrete Classes (HOW, Completely)

Concrete classes provide **complete, specific implementations**. They implement all remaining abstract methods and add any type-specific behavior.

The StoreItem Example

```
/**
 * Represents a simple item with standard pricing.
 * Membership status has no effect on StoreItem pricing.
 */
public final class StoreItem extends AbstractItem {

    // Additional fields IF needed

    public StoreItem(final String theName, final BigDecimal thePrice) {
        super(theName, thePrice);  // Delegate validation to AbstractItem
    }

    @Override
    public BigDecimal calculateTotal(final int quantity,
                                     final boolean useMembershipPricing) {
        // Validate quantity, then calculate: price × quantity
        // Membership has no effect on StoreItem pricing
    }

    @Override
    public String getFormattedDescription() {
        // Return formatted string like "Pen, $2.00"
    }

    // toString(), equals(), hashCode() - implement using getName() and
getPrice()
}
```

The StoreBulkItem Example

```
/**
 * Represents an item with bulk pricing available for members.
 */
public final class StoreBulkItem extends AbstractItem {

    // Additional fields IF needed
```

```
    public StoreBulkItem(final String theName, final BigDecimal thePrice,
                         final int theBulkQuantity, final BigDecimal
theBulkPrice) {
        super(theName, thePrice);  // Validate name and price via parent
        // Validate and store bulk-specific fields
    }

    @Override
    public BigDecimal calculateTotal(final int quantity,
                                     final boolean useMembershipPricing) {
        // For members: apply bulk pricing (sets of bulkQuantity at bulkPrice)
        // For non-members: simple multiplication
    }

    @Override
    public String getFormattedDescription() {
        // Return formatted string like "Notebook, $3.00 (5 for $12.00)"
    }

    // toString(), equals(), hashCode() - include bulk fields
}
```

Why Are Concrete Classes `final`?

The `final` keyword prevents further inheritance. This is intentional.

```
public final class StoreItem extends AbstractItem {
    // Cannot be extended
}
```

The `final` keyword prevents further inheritance for several reasons:

1. **Design clarity** — The hierarchy is complete. `StoreItem` is not meant to be a base for further specialization.
2. **Invariant protection** — Subclasses could break the assumptions made in `equals()` and `hashCode()`.
3. **Performance** — The JVM can optimize final classes more aggressively.
4. **Sealed hierarchy** — Classes extending a sealed parent must declare `final`, `sealed`, or `non-sealed`. Since we don't want further extension, `final` is the appropriate choice. (See Sealed Types and Records for details.)

> ⚠ **Warning**
>
> Joshua Bloch recommends in *Effective Java* Item 19: "Design and document for inheritance or else prohibit it." If you do not explicitly design a class for extension, make it `final`.

Calling `super()` from Constructors

When a subclass constructor runs, it must first initialize its parent. This is done by calling `super()`.

The Call Chain

```
// When you write:
StoreItem pen = new StoreItem("Pen", new BigDecimal("2.00"));

// The following happens:
// 1. JVM allocates memory for StoreItem
// 2. StoreItem constructor calls super(theName, thePrice)
// 3. AbstractItem constructor validates and stores name/price
// 4. Control returns to StoreItem constructor
// 5. StoreItem completes any additional initialization
```

Rules for `super()`

1. **Implicit if omitted** — If you do not write `super()`, Java inserts `super()` (no-arg) automatically.
2. **Must match parent signature** — Arguments to `super()` must match a parent constructor.
3. **Code before `super()` is allowed (JDK 25+)** — You can process parameters or initialize fields before calling `super()`, as long as you don't access `this` or call instance methods.

```
// Explicit super() call with arguments
public StoreItem(String name, BigDecimal price) {
    super(name, price);
    // Additional initialization here
}

// NEW in JDK 25: Processing before super() is now allowed
// (Demonstration only - not needed in A1C)
public StoreItem(String name, BigDecimal price) {
    String storeName = "UW Bookstore: " + name;
    BigDecimal normalizedPrice = price.setScale(2, RoundingMode.HALF_UP);
    super(storeName, normalizedPrice);  // Valid in JDK 25+
}
```

Flexible Constructor Bodies (JEP 513)

Prior to JDK 25, `super()` was required to be the first statement. JDK 25 relaxed this rule, allowing a **prologue** (statements before `super()`) as long as those statements don't reference `this`. This enables cleaner initialization patterns. See JEP 513 for details.

Validation in the Parent

Notice that `AbstractItem` handles validation. This means:

- Subclass constructors do not duplicate validation logic
- All items get consistent validation
- Changes to validation rules happen in one place

```
// In AbstractItem
protected AbstractItem(String theName, BigDecimal thePrice) {
    // Validation here applies to ALL subclasses
    if (Objects.requireNonNull(theName).isEmpty()) {
        throw new IllegalArgumentException("Name cannot be empty");
    }
    // ...
}

// In StoreItem - no need to re-validate name/price
public StoreItem(String theName, BigDecimal thePrice) {
    super(theName, thePrice);  // Parent validates
    // StoreItem-specific initialization (if any)
}
```

Code Reuse vs. Flexibility

The three-tier pattern balances two competing goals.

Code Reuse (DRY Principle)

DRY = "Don't Repeat Yourself"

Without an abstract class, every concrete class would duplicate:

- Field declarations for name and price
- Validation logic in constructors
- Implementations of `getName()` and `getPrice()`
- The `CURRENCY_FORMAT` constant

This duplication creates maintenance problems. If validation rules change, you must update multiple classes.

Flexibility (Open-Closed Principle)

The interface allows you to:

- Add new item types without changing existing code
- Program to the `Item` interface, not concrete classes
- Swap implementations without affecting client code

```
// Client code works with ANY Item implementation
public void printReceipt(List<Item> items) {
    for (Item item : items) {
        System.out.println(item.getFormattedDescription());
    }
}

// This works with StoreItem, StoreBulkItem, or any future Item type
```

The Tradeoff

Approach	Reuse	Flexibility
Interface only	None	Maximum
Abstract class only	Maximum	Limited
Three-tier pattern	High	High

The three-tier pattern gives you:

- **Reuse** through the abstract class (shared code)
- **Flexibility** through the interface (polymorphism)
- **Specificity** through concrete classes (specialized behavior)

Making Design Decisions

When adding a new method or field, ask these questions.

Where Does This Code Belong?

Put It in the Interface If:

- It defines behavior ALL implementations must support
- It is part of the public API that clients depend on
- It does not require implementation details

Put It in the Abstract Class If:

- It implements shared behavior identically for all subclasses
- It provides helper functionality for subclasses
- It manages shared state (fields)

Put It in the Concrete Class If:

- It varies between implementations
- It uses type-specific fields or logic
- It overrides parent behavior for a specific case

Example: Adding a `getCategory()` Method

Suppose we need items to report their category.

Option 1: All items have the same category logic

Put it in `AbstractItem`:

```
// In AbstractItem
private final String myCategory;

protected AbstractItem(String name, BigDecimal price, String category) {
    // ... existing validation ...
    myCategory = category;
}

public String getCategory() {
    return myCategory;
}
```

Option 2: Different item types determine category differently

Keep it abstract, implement in concrete classes:

```
// In Item interface
String getCategory();

// In StoreItem
@Override
public String getCategory() {
    return "Standard";
}

// In StoreBulkItem
@Override
public String getCategory() {
```

```
    return "Bulk";
}
```

The Assignment 1c Hierarchy

In Assignment 1c, you implement this exact pattern.

```
Item (sealed interface) ← PROVIDED
└── AbstractItem (sealed abstract class) ← YOU COMPLETE
    ├── StoreItem (final) ← YOU IMPLEMENT
    └── StoreBulkItem (final) ← YOU IMPLEMENT
```

Your Implementation Tasks

Component	Your Task
`Item`	Provided. Study the contract.
`AbstractItem`	Complete the stub. Implement constructor, `getName()`, `getPrice()`.
`StoreItem`	Full implementation. Simple pricing (no bulk).
`StoreBulkItem`	Full implementation. Bulk pricing for members.

Code Organization Summary

Method/Field	Location	Reason
`getName()`	AbstractItem	Same for all items
`getPrice()`	AbstractItem	Same for all items
`myName`, `myPrice`	AbstractItem	Shared state
`CURRENCY_FORMAT`	AbstractItem	Shared constant
Constructor validation	AbstractItem	Consistent for all

Method/Field	Location	Reason
`calculateTotal()`	StoreItem, StoreBulkItem	Different algorithms
`getFormattedDescription()`	StoreItem, StoreBulkItem	Different formats
`myBulkQuantity`, `myBulkPrice`	StoreBulkItem only	Type-specific state

Common Mistakes

1. Duplicating Code Across Concrete Classes

Problem: Writing the same validation logic in both `StoreItem` and `StoreBulkItem`.

Solution: Put shared logic in `AbstractItem`. Let `super()` handle it.

2. Forgetting to Call `super()`

Problem: Not calling the parent constructor, causing fields to be uninitialized.

```
// BAD: Forgot super() - myName and myPrice never set!
public StoreItem(String name, BigDecimal price) {
    // Missing: super(name, price);
}
```

Solution: Always call `super()` with required arguments as the first line.

3. Making the Abstract Class Constructor Public

Problem: Using `public` instead of `protected` for abstract class constructors.

```
// BAD: Public constructor suggests you can instantiate AbstractItem
public AbstractItem(String name, BigDecimal price) { ... }
```

Solution: Use `protected` to indicate the constructor is for subclasses only.

4. Implementing Varying Behavior in Abstract Class

Problem: Implementing `calculateTotal()` in `AbstractItem` when algorithms differ.

Solution: Keep methods with varying implementations abstract or unimplemented at the abstract level.

5. Not Making Concrete Classes Final

Problem: Leaving concrete classes open for extension without designing for it.

Solution: Make concrete classes `final` unless you explicitly design and document them for extension.

Summary

Concept	Key Point
Interface	Defines WHAT operations a type supports (contract)
Abstract Class	Provides HOW for shared behavior (partial implementation)
Concrete Class	Provides HOW for specific behavior (complete implementation)
Protected Constructor	Signals constructor is for subclasses only
`super()` **Call**	Must be first statement in subclass constructor
`final` **Classes**	Prevent inheritance when not designed for extension
Code Reuse	Put shared code in abstract class
Flexibility	Program to interfaces, not implementations

The three-tier pattern is foundational to professional Java development. It appears throughout the Java standard library (`Collection` -> `AbstractCollection` -> `ArrayList`) and in virtually every enterprise application. Master this pattern and you will understand how large-scale systems are organized.

Further Reading

External Resources

- Oracle Java Tutorial: Abstract Methods and Classes - Official abstract class documentation
- Oracle Java Tutorial: Interfaces - Official interface documentation
- Baeldung: Abstract Classes in Java - Practical tutorial with examples
- Baeldung: Guide to Inheritance - Comprehensive inheritance overview

---

References

Primary Texts:

- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. Item 19: Design and document for inheritance or else prohibit it; Item 20: Prefer interfaces to abstract classes.
- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals* (12th ed.). Oracle Press. Chapter 5: Inheritance; Chapter 6: Interfaces, Lambda Expressions, and Inner Classes.
- Reges, S., & Stepp, M. (2020). *Building Java Programs* (5th ed.). Pearson. Chapter 9: Inheritance and Interfaces.

Language Documentation:

- Oracle JDK 25: Abstract Methods and Classes — Official abstract class documentation
- Oracle JDK 25: Interfaces — Official interface documentation
- Oracle JDK 25: Sealed Classes — Sealed class and interface documentation

Design Principles:

- Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall. — Open-Closed Principle (OCP)
- Hunt, A., & Thomas, D. (1999). *The Pragmatic Programmer*. Addison-Wesley. — DRY (Don't Repeat Yourself) principle

---

This guide is part of TCSS 305 Programming Practicum, School of Engineering and Technology, University of Washington Tacoma.