a1a # a1b # defensive-coding # java-fundamentals # oop

Interface Contracts: Designing for Collaboration

TCSS 305 Programming Practicum

This guide explains what interfaces are, why they matter for professional software development, and how to implement them correctly. You'll learn how interfaces enable teams to work independently and why defensive coding enforces the "contract" between code components.

What is an Interface?

An **interface** in Java is a blueprint that defines what a class can do without specifying how it does it. It declares method signatures—the names, parameters, and return types—but contains no implementation code.

```
public interface Item {
    String getName();
    BigDecimal getPrice();
}
```

This interface says: "Any class that implements `Item` must have a `getName()` method that returns a `String` and a `getPrice()` method that returns a `BigDecimal`."

Think of it like a job description versus the person doing the job:

Job Description (Interface)	Employee (Implementation)
"Must process customer orders"	How they process orders (phone, computer, paper)
"Must generate sales reports"	How they generate reports (Excel, SQL, hand-written)
"Must respond to customer inquiries"	How they respond (email, chat, phone call)

The job description defines **what** needs to be done. The employee decides **how** to do it. Similarly, an interface defines what a class must do, and the implementing class provides the actual code.

Same Interface, Different Implementations

Here's a concrete example. Imagine a game that needs to save progress:

```
public interface GameSaver {
    void saveGame(GameState state);
}
```

The interface says WHAT: "save the game state." But HOW? That depends on the implementation:

```
// Saves to a local file
public class FileSaver implements GameSaver {
    public void saveGame(GameState state) {
        // Write state to disk
    }
}

// Saves to a database
public class DatabaseSaver implements GameSaver {
    public void saveGame(GameState state) {
        // INSERT into database
    }
}

// Saves to cloud storage via HTTP
public class CloudSaver implements GameSaver {
    public void saveGame(GameState state) {
        // POST to cloud API
    }
}
```

The calling code doesn't care HOW the game is saved—it just calls `saver.saveGame(state)`. You could swap `FileSaver` for `CloudSaver` and the rest of the application works unchanged. That's the power of interfaces.

> **Note**
>
> You've already used interfaces without realizing it. When you use `List<String>`, that's an interface. `ArrayList` and `LinkedList` are different implementations of the same interface. Your code works with either because they both fulfill the `List` contract.

Why Interfaces Matter: The Team Analogy

Here's where interfaces become powerful—and where most students first understand their real value.

Two Teams, One Project

Imagine a large software project with two teams working on the UW Bookstore application:

Team A: Shopping Cart Team - Building the shopping cart functionality - Needs to add items to a cart, calculate totals, apply discounts - Works with `Item` objects

Team B: Inventory Team - Building the inventory management system - Creates and manages all the store items - Produces `Item` objects

These teams work in different buildings. They started the same week and have a deadline in two months. They never meet, never talk, never share code during development.

The Magic Moment

On deployment day, Team A's shopping cart code and Team B's inventory code are combined for the first time.

It works perfectly. No bugs. No incompatibilities. No "your code broke my code."

How is this possible?

The Interface is the Contract

Both teams coded to the `Item` **interface**:

```
// Team A's code (Shopping Cart)
public class Cart {
    public void add(Item item, int quantity) {
        // Cart doesn't care if this is a StoreItem, BulkItem, or any other
Item
        // It only cares that item has getName() and getPrice()
        BigDecimal price = item.getPrice();
        // ... add to cart
    }
}

// Team B's code (Inventory)
public class StoreItem implements Item {
    private String myName;
    private BigDecimal myPrice;

    @Override
    public String getName() { return myName; }

    @Override
    public BigDecimal getPrice() { return myPrice; }
}
```

Team A doesn't know or care how `StoreItem` works internally. They only know it implements `Item`. Team B doesn't know or care how `Cart` uses their items. They only know they're providing the methods defined in `Item`.

The interface is the **contract** between them.

> **Important**
>
> This is how professional software development works. Teams at companies like Amazon, Google, and Microsoft work on different parts of large systems simultaneously. They integrate seamlessly because everyone codes to agreed-upon interfaces. The interface IS the specification.

The Interface as a Contract

The legal analogy helps clarify what a "contract" means in programming:

Like a Legal Contract

Legal Contract	Interface Contract
Defines obligations for both parties	Defines obligations for implementer and caller
Penalties for breaking terms	Exceptions for violating contract
Written in precise legal language	Written in method signatures and Javadoc
Enforceable by law	Enforceable by compiler and runtime

Two Sides of the Contract

The Implementer's Promise: - "I will provide all methods declared in the interface" - "My methods will behave as documented in the Javadoc" - "I will throw the documented exceptions for invalid input"

The Caller's Promise: - "I will provide valid arguments as specified" - "I will handle the documented exceptions" - "I will not assume behavior beyond what's documented"

If either side breaks their promise, the contract is violated—and an exception is the result.

Anatomy of the Item Interface

Let's examine a simplified version of the `Item` interface you'll implement:

```
/**
 * Represents a single item for sale in the bookstore.
 */
public interface Item {

    /**
     * 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();
}
```

Every part of this interface is important:

Element	Purpose
Interface declaration	Defines the type that classes will implement
Method signatures	The methods every implementer must provide
Javadoc comments	The contract details—exactly how methods must behave
`@return` tags	What the caller can expect to receive
`@throws` tags (when present)	Preconditions the caller must satisfy

The Javadoc IS the contract specification. It tells you exactly what behavior is required.

Preconditions and Postconditions

Contracts have two types of conditions:

Preconditions: What Must Be True Before

Preconditions are requirements the **caller** must satisfy before calling a method. If the caller violates a precondition, the method throws an exception.

In a constructor for `StoreItem`: - The name cannot be null - The name cannot be empty - The price cannot be null - The price cannot be negative

These are documented with `@throws` in Javadoc:

```
/**
 * Constructs a new StoreItem 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
 */
public StoreItem(String theName, BigDecimal thePrice) {
    // ...
}
```

Postconditions: What Will Be True After

Postconditions are guarantees the **implementer** makes about the state after a method completes. They tell the caller what they can expect.

- `getName()` will return the item's name (never null)
- `getPrice()` will return a non-negative price

These are documented in the method description and `@return` tags.

> **Tip**
>
> When implementing an interface, read every `@throws` tag carefully. Each one is a precondition you must enforce in your code.

Defensive Coding: Enforcing the Contract

Now we arrive at a crucial question many students ask:

"If someone passes null for a name, why not just use 'Unknown' as a default? Why throw an exception?"

The "Fail Fast" Philosophy

The answer is fundamental to professional software development: **hiding bugs is worse than exposing them.**

Consider this scenario:

```
// Somewhere in a large application, someone writes:
Item item = new StoreItem(getUserName(), getPrice());
cart.add(item);
```

If `getUserName()` returns `null` due to a bug, what should happen?

Option A: "Fix" the input (BAD)

```
public StoreItem(String name, BigDecimal price) {
    myName = (name == null) ? "Unknown" : name;  // Silently "fix" the problem
    // ...
}
```

The item is created with name "Unknown". The cart works. The application seems fine.

Then, three months later, a customer complains: "Why does my receipt say 'Unknown' instead of my order name?" A developer spends hours tracking down where "Unknown" came from. The original bug—`getUserName()` returning null—happened far away in the code and long ago in time.

Option B: Throw an exception (GOOD)

```
public StoreItem(String name, BigDecimal price) {
    Objects.requireNonNull(name, "Name cannot be null");  // Fail immediately
    // ...
}
```

The application crashes immediately with a clear error message pointing to the exact line where the null was passed. The bug is obvious, located precisely, and can be fixed in minutes.

> ⚠ **Warning**
>
> Silently "fixing" bad input hides bugs. The bug still exists—it just surfaces later, far from its source, making debugging a nightmare. The contract is clear: "Give me valid input, or I'll tell you it's wrong."

The Defensive Coding Pattern

Here's the standard pattern for enforcing preconditions in a constructor:

```
public StoreItem(String theName, BigDecimal thePrice) {
    // Check for null first (throws NullPointerException implicitly)
    // Then check for other invalid values (throws IllegalArgumentException
explicitly)
    if (Objects.requireNonNull(theName, "Name cannot be null").isEmpty()) {
        throw new IllegalArgumentException("Name cannot be empty");
    }
    if (Objects.requireNonNull(thePrice, "Price cannot be null")
            .compareTo(BigDecimal.ZERO) < 0) {
        throw new IllegalArgumentException("Price cannot be negative");
    }

    myName = theName;
    myPrice = thePrice;
}
```

This code: 1. Uses `Objects.requireNonNull()` to check for null (throws `NullPointerException` if null) 2. Uses explicit checks for other invalid conditions (throws `IllegalArgumentException`) 3. Only assigns values after all validation passes

The Exception: equals(null)

There are rare cases where we tolerate null input. The most common is `equals()`:

```
@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;  // Don't throw, just return false
    }
    // ...
}
```

This is a Java convention. Comparing anything to `null` returns `false` rather than throwing an exception. But this is the exception, not the rule—and it's documented behavior.

> **Note**
>
> The key difference: `equals(null)` is *designed* to accept null and has documented behavior for it. Constructors are *not* designed to accept invalid data—that's a contract violation.

Reading an Interface to Understand Your Requirements

When you implement an interface, the interface plus its Javadoc IS your specification. Here's how to read it:

Step 1: Read Every Method Signature

Look at what methods you must implement. Note the parameter types and return types—these cannot change.

Step 2: Read Every Javadoc Comment

The description tells you what the method should do. This is the behavior you must provide.

Step 3: Pay Special Attention to @throws

Each `@throws` tag is a precondition you must enforce:

```
/**
 * @throws NullPointerException if theName is null
 * @throws IllegalArgumentException if theName is empty
 */
```

This tells you: your code must check for null, check for empty, and throw the appropriate exceptions.

> **Gen AI & Learning: Using AI to Explain Interface Contracts**
>
> AI assistants can help you understand unfamiliar interfaces by explaining what each method does, what the `@throws` tags require, and why certain preconditions exist. Try prompting: "Explain this interface contract and what I need to implement." However, **you** must write the implementation code—using AI to generate your implementation defeats the learning purpose and may violate academic integrity policies. The goal is understanding the "why" behind contracts, not bypassing that understanding.

Step 4: Understand the Tests

The provided unit tests verify you met the contract. If a test fails, you either: - Didn't implement the required behavior, or - Didn't enforce a precondition, or - Used the wrong exception type

---

Looking Ahead: Sealed Interfaces

In later assignments, you'll encounter `sealed` interfaces:

```
public sealed interface Item permits StoreItem, StoreBulkItem {
    // ...
```

```
}
```

The `sealed` keyword restricts which classes can implement the interface. This creates a closed, well-defined type system where the compiler knows all possible types.

For now, just know this exists. You'll learn more about sealed interfaces in Assignment 1C, where they enable powerful pattern matching features.

Javadoc on Overridden Methods

A common question: "Do I need to write Javadoc for methods I'm implementing from an interface?"

When You Can Inherit Javadoc

If your implementation follows the interface's contract exactly, you can omit Javadoc—it will be inherited from the interface. IntelliJ even shows the inherited documentation when you hover over the method.

```
@Override
public String getName() {
    return myName;  // Javadoc inherited from Item interface
}
```

When You MUST Write Your Own Javadoc

If your implementation behaves differently from the parent contract, you **must** document that difference.

The classic example is `toString()`. The `Object.toString()` contract states it returns:

> "a string representation of the object... should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method."

The default format is `ClassName@hexHashCode` (e.g., `StoreItem@1a2b3c4d`).

But we override it to return something useful:

```
/**
 * Returns a String representation of this Item for debugging purposes.
 * <p>
 * Format: "StoreItem[name='itemName', price=X.XX]"
 * <p>
 * This differs from the default Object.toString() format to provide
```

```
     * more useful debugging information.
     *
     * @return a String representation of this Item
     */
    @Override
    public String toString() {
        return "StoreItem[name='" + myName + "', price=" + myPrice + "]";
    }
```

Since our `toString()` doesn't follow the default `ClassName@hash` format, we document what it *actually* returns.

> **Tip**
>
> Rule of thumb: If your implementation does exactly what the interface/parent says, inherit the Javadoc. If it does something different or more specific, write your own.

The @Override Annotation

What Are Annotations?

Annotations are a form of metadata in Java—information about code that isn't part of the code's logic itself. They're marked with the `@` symbol and provide instructions to the compiler, tools, or runtime.

You may have already seen annotations:

- `@Override` — tells the compiler you intend to override a method
- `@Test` — tells JUnit this method is a test case
- `@Deprecated` — warns that a method shouldn't be used anymore

Annotations don't change what your code *does*—they provide information *about* your code that tools can use.

Why @Override Matters

The `@Override` annotation tells the compiler: "I intend to override a method from my superclass or interface." The compiler then verifies this is actually true.

Without @Override—silent bugs:

```
// Programmer thinks they're overriding equals()
public boolean equals(StoreItem other) {  // WRONG parameter type!
```

```
    return myName.equals(other.myName);
}
```

This code compiles without error. But `equals(StoreItem)` is **overloading**, not overriding—the parameter must be `Object`. The real `equals(Object)` from `Object` is never overridden, so your class won't work correctly in collections.

With @Override—compiler catches the mistake:

```
@Override  // ERROR: Method does not override method from its superclass
public boolean equals(StoreItem other) {
    return myName.equals(other.myName);
}
```

The compiler immediately flags the error because there's no `equals(StoreItem)` in any superclass to override.

The Rule: Always Use @Override

Use `@Override` every time you:

- Implement a method from an interface
- Override a method from a parent class
- Override methods from `Object` (`equals`, `hashCode`, `toString`)

```
public class StoreItem implements Item {

    @Override  // Implementing interface method
    public String getName() {
        return myName;
    }

    @Override  // Overriding Object.equals()
    public boolean equals(Object obj) {
        // ...
    }

    @Override  // Overriding Object.hashCode()
    public int hashCode() {
        // ...
    }

    @Override  // Overriding Object.toString()
    public String toString() {
        // ...
    }
}
```

> **Tip**
>
> `@Override` costs nothing and catches subtle bugs. Make it a habit—add it every time you override a method, without exception.

Common Mistakes

1. Not Reading the Javadoc Carefully

Problem: You implement a method based on the name alone, missing important behavioral requirements.

Solution: The Javadoc IS the specification. Read every word, especially `@throws` tags.

2. Forgetting to Validate Input

Problem: Your constructor accepts null or negative values silently.

Solution: Validate all parameters at the beginning of the constructor. Use `Objects.requireNonNull()` for null checks.

3. Using the Wrong Exception Type

Problem: Throwing `IllegalArgumentException` for null when `NullPointerException` is specified.

Solution: Use the exact exception type documented in the `@throws` tag. `NullPointerException` for null, `IllegalArgumentException` for other invalid values.

4. Silently "Fixing" Bad Input

Problem: Using default values instead of throwing exceptions for invalid input.

Solution: Throw exceptions for contract violations. Fail fast, fail loud.

5. Adding Methods Not in the Interface

Problem: Creating public methods that aren't part of the interface.

Solution: Only implement the methods the interface requires (plus standard `Object` methods like `equals`, `hashCode`, `toString`).

6. Changing Method Signatures

Problem: Changing parameter names is fine, but changing types breaks the contract.

Solution: Match the exact return types and parameter types from the interface.

Summary

Concept	Key Point
Interface	Defines what a class can do, not how
Contract	Both caller and implementer have obligations
Preconditions	What must be true before calling (caller's responsibility)
Postconditions	What will be true after calling (implementer's guarantee)
Defensive coding	Enforce preconditions with exceptions
Fail fast	Expose bugs immediately, don't hide them
Javadoc	The specification IS the contract

Interfaces enable independent teams to work together without direct communication. By coding to the interface—the shared contract—components integrate seamlessly. This is the foundation of large-scale software development.

Further Reading

External Resources

- Oracle Java Tutorial: Interfaces - Official Java documentation on interfaces
- Effective Java, Item 20: Prefer interfaces to abstract classes - Joshua Bloch's classic book
- Baeldung: Java Interfaces - Practical tutorial with examples
- Design by Contract - The formal theory behind interface contracts
- Objects.requireNonNull() Javadoc - Official documentation for null checking

---

References

Primary Texts:

- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. Item 20: Prefer interfaces to abstract classes; Item 49: Check parameters for validity; Item 64: Refer to objects by their interfaces.
- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals* (12th ed.). Oracle Press. 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 Java Tutorial: Interfaces — Official interface documentation
- Oracle JDK 25: java.util.Objects — Objects.requireNonNull() and related methods
- Oracle JDK 25: Sealed Classes — Sealed interface documentation

Design Principles:

- Meyer, B. (1992). Applying "Design by Contract". *IEEE Computer*, 25(10), 40-51. — Original Design by Contract paper
- Design by Contract (Wikipedia) — Overview of contract-based design

---

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