a1c # defensive-coding # java-fundamentals

Defensive Programming: Fail Fast, Fail Loud

TCSS 305 Programming Practicum

This guide explains defensive programming principles—how to validate input, when to throw exceptions, and why "failing fast" makes code more reliable and easier to debug. You'll learn patterns for protecting your code at its boundaries while trusting internal state.

Why Defensive Programming Matters

Every method makes assumptions about its inputs. When those assumptions are violated, something goes wrong. The question is: **when and how does it go wrong?**

Consider this scenario: a method receives `null` when it expects a valid object. Two approaches:

Approach A: Silently tolerate the problem

```
public StoreItem(String name, BigDecimal price) {
    myName = (name == null) ? "Unknown" : name;  // "Fix" the null
    myPrice = (price == null) ? BigDecimal.ZERO : price;
}
```

The object is created. It seems to work. Weeks later, a customer complains: "Why does my receipt say 'Unknown'?" A developer spends hours tracing through logs, database records, and code paths to find where the null originated.

Approach B: Fail immediately with a clear error

```
public StoreItem(String name, BigDecimal price) {
    myName = Objects.requireNonNull(name, "Name cannot be null");
    myPrice = Objects.requireNonNull(price, "Price cannot be null");
}
```

The application fails immediately with a stack trace pointing to the exact line where invalid data was passed. The bug is found in minutes, not hours.

> **Important**
>
> Defensive programming isn't about being paranoid—it's about making bugs **visible and locatable**. The earlier a bug manifests, the easier it is to fix. This principle is called **fail fast**.

The Fail-Fast Philosophy

Fail fast means detecting and reporting errors as soon as they occur, rather than allowing them to propagate through the system.

Why Fail Fast?

Silent Failure	Fail Fast
Bug hides in the system	Bug surfaces immediately
Symptoms appear far from cause	Error points to exact location
Debugging takes hours/days	Debugging takes minutes
Data corruption may occur	Invalid data never enters system
Multiple bugs may compound	Each bug caught individually

The Contract Perspective

Think of each method as having a **contract** with its callers:

- **Caller's obligation:** Provide valid arguments
- **Method's obligation:** Return correct results (given valid input)
- **Method's right:** Reject invalid input by throwing exceptions

When a caller violates the contract (passes invalid arguments), the method has every right—and responsibility—to refuse service. Throwing an exception is not being "mean" to the caller; it's enforcing the agreement that makes the system reliable.

Guide

Interface Contracts — Deeper exploration of contracts, preconditions, and postconditions.

Validation at Boundaries

Not all code needs validation. The key principle: **validate at trust boundaries**.

What Are Trust Boundaries?

A trust boundary is where data enters your code from an untrusted source:

Boundary Type	Examples	Trust Level
Public constructors	`new StoreItem(name, price)`	Untrusted—any code can call
Public accessors	`cart.add(item, qty)`, `item.setPrice(p)`	Untrusted—any code can call
Private methods	`calculateDiscount(rate)`, `formatCurrency(amt)`	Trusted—only called by your code

Public Methods: Validate Everything

Every public method is a potential entry point. Validate all parameters:

```
public class ShoppingCart {

    /**
     * Adds an item to this cart.
     *
     * @param theItem the item to add
     * @param theQuantity the quantity to add
     * @throws NullPointerException if theItem is null
     * @throws IllegalArgumentException if theQuantity is not positive
     */
    public void add(final Item theItem, final int theQuantity) {
        Objects.requireNonNull(theItem, "Item cannot be null");
        if (theQuantity <= 0) {
            throw new IllegalArgumentException("Quantity must be positive: " +
theQuantity);
        }
```

```
        // Now safe to proceed...
    }
}
```

Private Methods: Trust Internal State

Private methods are only called by your own code. If you've validated data at the public boundary, you don't need to re-validate in private helpers:

```
public class ShoppingCart {

    public void add(final Item theItem, final int theQuantity) {
        Objects.requireNonNull(theItem, "Item cannot be null");
        if (theQuantity <= 0) {
            throw new IllegalArgumentException("Quantity must be positive");
        }

        // Private method—no validation needed
        addToInternalList(theItem, theQuantity);
    }

    // Private: trusts that caller (add) already validated
    private void addToInternalList(final Item item, final int quantity) {
        // No null check needed—add() already verified
        // Implementation: add item to internal collection
    }
}
```

Tip

Validating in private methods is redundant and adds clutter. Trust that your public methods did their job. If you're tempted to validate in private methods, ask: "How could invalid data reach here?" If the answer is "only through a public method that validates," skip the redundant check.

Objects.requireNonNull(): The Standard Null Check

Java provides `Objects.requireNonNull()` specifically for null parameter validation. It's in the `java.util.Objects` class and has been the standard approach since Java 7.

Related Guide

The `Objects` class provides several utility methods beyond `requireNonNull()`. See The Objects Utility Class for `Objects.equals()`, `Objects.hash()`, and other useful methods.

Basic Usage

```
import java.util.Objects;

public StoreItem(String theName, BigDecimal thePrice) {
    myName = Objects.requireNonNull(theName, "Name cannot be null");
    myPrice = Objects.requireNonNull(thePrice, "Price cannot be null");
}
```

How It Works

`requireNonNull` is simple:

1. If the argument is not null, it returns the argument unchanged
2. If the argument is null, it throws `NullPointerException` with the provided message

```
// Equivalent manual code (but use requireNonNull instead)
if (theName == null) {
    throw new NullPointerException("Name cannot be null");
}
myName = theName;

// Better: single line with requireNonNull
myName = Objects.requireNonNull(theName, "Name cannot be null");
```

Explicit vs. Implicit NPE

You might wonder: "If I don't check for null, won't Java throw a `NullPointerException` anyway when I try to use it?"

Yes—but there's a critical difference:

Implicit NPE (no validation):

```
public StoreItem(String theName, BigDecimal thePrice) {
    myName = theName;  // No validation
    myPrice = thePrice;
}

public String getFormattedDescription() {
    // NPE thrown HERE when myName.length() is called
    return myName.toUpperCase() + ": $" + myPrice;
}
```

The object is created successfully with null inside. The NPE occurs later, in a different method, when you try to use the null value. The stack trace points to `getFormattedDescription()`, not to where the null was actually passed.

Explicit NPE (with validation):

```
public StoreItem(String theName, BigDecimal thePrice) {
    myName = Objects.requireNonNull(theName, "Name cannot be null");
    myPrice = Objects.requireNonNull(thePrice, "Price cannot be null");
}
```

The NPE is thrown immediately in the constructor, with a clear message, pointing exactly to where the invalid data entered the system.

> **Important**
>
> Explicit null checks with `Objects.requireNonNull()` make bugs **locatable**. The exception happens at the boundary where the contract was violated, not somewhere deep in the code later.

requireNonNull Variants

`Objects` provides several overloads:

```
// Basic check, default message
Objects.requireNonNull(value);
// Throws: NullPointerException (no message)

// With custom message
Objects.requireNonNull(value, "Description cannot be null");
// Throws: NullPointerException: Description cannot be null

// With lazy message (avoids string concatenation if not null)
Objects.requireNonNull(value, () -> "Item " + id + " cannot be null");
// Message only computed if exception is thrown
```

For most cases, the simple message version is best. Use the supplier version when the message involves expensive computation (like string concatenation with method calls).

IllegalArgumentException: Beyond Null

Null is just one type of invalid input. For other validation failures, throw `IllegalArgumentException`:

Common Validation Patterns

```
public StoreItem(String theName, BigDecimal thePrice) {
    // Null checks first (throw NullPointerException)
    Objects.requireNonNull(theName, "Name cannot be null");
```

```
    Objects.requireNonNull(thePrice, "Price cannot be null");

    // Semantic validation (throw IllegalArgumentException)
    if (theName.isEmpty()) {
        throw new IllegalArgumentException("Name cannot be empty");
    }
    if (theName.isBlank()) {
        throw new IllegalArgumentException("Name cannot be blank");
    }
    if (thePrice.compareTo(BigDecimal.ZERO) < 0) {
        throw new IllegalArgumentException("Price cannot be negative: " +
thePrice);
    }

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

Exception Type Guidelines

Condition	Exception	Example
Argument is `null`	`NullPointerException`	`Objects.requireNonNull(name)`
Argument violates semantic constraint	`IllegalArgumentException`	Empty string, negative number
Object state prevents operation	`IllegalStateException`	Calling `next()` on empty iterator
Index out of bounds	`IndexOutOfBoundsException`	Array/list index errors

Note

The Javadoc `@throws` tag documents which exception to throw for each condition. Follow the specification exactly—tests verify the correct exception type.

Including Context in Messages

Good exception messages help debugging. Include relevant values:

```
// Unhelpful
throw new IllegalArgumentException("Invalid quantity");
```

```
// Helpful
throw new IllegalArgumentException("Quantity must be positive, was: " +
quantity);

// Even better for complex validations
throw new IllegalArgumentException(
    String.format("Price must be between %s and %s, was: %s",
                  MIN_PRICE, MAX_PRICE, price));
```

Validation Order and Short-Circuit Patterns

When validating multiple conditions, order matters. Use these patterns:

Pattern 1: Null Checks First

Always check for null before accessing object methods:

```
// GOOD: Check null, then check isEmpty()
public StoreItem(String theName, BigDecimal thePrice) {
    Objects.requireNonNull(theName, "Name cannot be null");
    if (theName.isEmpty()) {  // Safe: theName is definitely not null
        throw new IllegalArgumentException("Name cannot be empty");
    }
    // ...
}

// BAD: isEmpty() would throw NPE if name is null
public StoreItem(String theName, BigDecimal thePrice) {
    if (theName.isEmpty()) {  // NPE if theName is null!
        throw new IllegalArgumentException("Name cannot be empty");
    }
    // ...
}
```

Pattern 2: Chained Validation with requireNonNull

Since `requireNonNull` returns its argument, you can chain validations:

```
public StoreItem(String theName, BigDecimal thePrice) {
    // requireNonNull returns theName if not null, so isEmpty() is safe
    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 pattern:

1. Checks for null first (returns the value if valid)
2. Immediately uses that value for the next check
3. Keeps related validations together on one logical "line"

Pattern 3: Fail on First Error

Don't continue validating after finding an error:

```
// GOOD: Return/throw immediately on first error
public StoreItem(String theName, BigDecimal thePrice) {
    if (Objects.requireNonNull(theName).isEmpty()) {
        throw new IllegalArgumentException("Name cannot be empty");
    }
    // Only reaches here if name is valid

    if (Objects.requireNonNull(thePrice).compareTo(BigDecimal.ZERO) < 0) {
        throw new IllegalArgumentException("Price cannot be negative");
    }
    // Only reaches here if price is also valid

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

This is efficient (stops at first error) and produces clear error messages (one problem at a time).

Extracting Validation to Helper Methods

When a constructor or method has many validation checks, the code can become long and hard to follow. A common pattern is extracting validation into a private helper method.

When to Extract

Consider extracting validation when:

- The constructor has many validation statements (5+)
- Code quality tools flag the method as too long

- The same validations apply to multiple constructors

The Pattern

```
public Employee(final String theName, final String theEmail,
                final BigDecimal theSalary) {
    validateArguments(theName, theEmail, theSalary);  // Delegate validation
    myName = theName;
    myEmail = theEmail;
    mySalary = theSalary;
}

private static void validateArguments(final String theName,
                                      final String theEmail,
                                      final BigDecimal theSalary) {
    // All null checks and semantic validation here...
}
```

This keeps the constructor focused on its core responsibility: validate, then assign. See the Complete Validation Example below for a full implementation.

Why This Doesn't Violate "Trust Private Methods"

Earlier, we said private methods shouldn't re-validate data that was already checked at a public boundary. So why is this different?

The key distinction is **where validation happens**, not **who performs it**:

Scenario	Description	Redundant?
Public method validates, then calls private helper	Private helper receives already-validated data	Yes—don't re-validate
Public constructor delegates to private validator	Private method **is** the validation	No—this is the boundary

The `validateArguments()` method isn't trusting already-validated data—it's **performing** the validation on behalf of the public constructor. Think of it as *moving* the validation logic into a helper, not duplicating it.

> **Tip**
>
> Use `private static` for validation helpers when they don't need instance state. This makes it clear the method doesn't depend on object construction order.

Checked vs. Unchecked Exceptions

Java has two categories of exceptions. Understanding when to use each is crucial.

Unchecked Exceptions (RuntimeException)

Do not require explicit handling. Used for programming errors—conditions that indicate bugs in the calling code.

```
// Unchecked: caller made a mistake
throw new NullPointerException("Name cannot be null");
throw new IllegalArgumentException("Quantity must be positive");
throw new IllegalStateException("Cart is closed");
```

When to throw unchecked exceptions:

- Invalid arguments (caller's bug)
- Null when non-null required (caller's bug)
- Invalid object state (indicates internal bug)
- Index out of bounds (caller's bug)

> **Tip**
>
> In TCSS 305, you'll primarily use unchecked exceptions for parameter validation. These include `NullPointerException`, `IllegalArgumentException`, and `IllegalStateException`.

Checked Exceptions (Exception, not RuntimeException)

Require explicit handling with try-catch or throws declaration. Used for recoverable conditions outside the program's control.

```
// Checked: caller should handle this
throw new IOException("File not found");
throw new SQLException("Database connection failed");
```

When to throw checked exceptions:

- External resource failures (files, network, database)
- Conditions the caller should anticipate and recover from
- API operations that might legitimately fail

The Decision Framework

Condition	Exception Type	Reason
Caller passed null	Unchecked (`NullPointerException`)	Bug in calling code
Caller passed invalid value	Unchecked (`IllegalArgumentException`)	Bug in calling code
File doesn't exist	Checked (`FileNotFoundException`)	Caller should handle gracefully
Network timeout	Checked (`SocketTimeoutException`)	Caller should retry or report

Gen AI & Learning: Exception Design Decisions

When designing APIs, choosing between checked and unchecked exceptions can be nuanced. AI assistants can help you explore the tradeoffs: "Should I throw a checked or unchecked exception when X happens?" Understanding *why* the Java library designers made certain choices (like `FileNotFoundException` being checked but `NumberFormatException` being unchecked) builds your intuition for API design.

Complete Validation Example

Here's a complete example showing all validation patterns in a real class, using the helper method pattern from the previous section:

```
import java.math.BigDecimal;
import java.util.Objects;

/**
 * Represents an employee with validated contact and salary information.
 */
public final class Employee {

    /** The employee's full name. */
    private final String myName;

    /** The employee's email address. */
    private final String myEmail;
```

```java
    /** The employee's annual salary. */
    private final BigDecimal mySalary;

    /**
     * Constructs an Employee with the specified information.
     *
     * @param theName the employee's full name
     * @param theEmail the employee's email address
     * @param theSalary the employee's annual salary
     * @throws NullPointerException if any argument is null
     * @throws IllegalArgumentException if theName is empty or blank,
     *         if theEmail doesn't contain '@', or if theSalary is negative
     */
    public Employee(final String theName, final String theEmail,
                    final BigDecimal theSalary) {
        validateArguments(theName, theEmail, theSalary);
        myName = theName;
        myEmail = theEmail;
        mySalary = theSalary;
    }

    /**
     * Validates all constructor arguments.
     *
     * @param theName the name to validate
     * @param theEmail the email to validate
     * @param theSalary the salary to validate
     * @throws NullPointerException if any argument is null
     * @throws IllegalArgumentException if validation fails
     */
    private static void validateArguments(final String theName,
                                          final String theEmail,
                                          final BigDecimal theSalary) {
        // Null checks with requireNonNull
        Objects.requireNonNull(theName, "Name cannot be null");
        Objects.requireNonNull(theEmail, "Email cannot be null");
        Objects.requireNonNull(theSalary, "Salary cannot be null");

        // Semantic validation with IllegalArgumentException
        if (theName.isEmpty()) {
            throw new IllegalArgumentException("Name cannot be empty");
        }
        if (theName.isBlank()) {
            throw new IllegalArgumentException("Name cannot be blank");
        }
        if (!theEmail.contains("@")) {
            throw new IllegalArgumentException(
                "Email must contain '@': " + theEmail);
        }
        if (theSalary.compareTo(BigDecimal.ZERO) < 0) {
            throw new IllegalArgumentException(
                "Salary cannot be negative: " + theSalary);
        }
    }

    // ... accessors omitted for brevity
}
```

Assignment Context: Constructor Validation

In TCSS 305 assignments, constructors are the primary validation boundary. The pattern is consistent:

1. **Check for null** using `Objects.requireNonNull()` with descriptive messages
2. **Check semantic constraints** with explicit if-statements
3. **Throw the documented exception type** (check the `@throws` Javadoc)
4. **Assign fields only after all validation passes**

The tests verify both normal operation and exception behavior:

```
// Tests that valid input works
@Test
void testConstructorWithValidArguments() {
    Employee emp = new Employee("Ada Lovelace", "ada@example.com", new
BigDecimal("75000"));
    assertEquals("Ada Lovelace", emp.getName(), "Name should match constructor
argument");
}

// Tests that null is rejected
@Test
void testConstructorRejectsNullName() {
    assertThrows(NullPointerException.class,
        () -> new Employee(null, "ada@example.com", new BigDecimal("75000")),
        "Constructor should reject null name");
}

// Tests that invalid email is rejected
@Test
void testConstructorRejectsInvalidEmail() {
    assertThrows(IllegalArgumentException.class,
        () -> new Employee("Ada Lovelace", "not-an-email", new
BigDecimal("75000")),
        "Constructor should reject email without @");
}
```

Common Mistakes

Mistake 1: Silently "Fixing" Invalid Input

Problem: Using default values instead of rejecting bad input.

```
// BAD: Hides bugs
public StoreItem(String name, BigDecimal price) {
```

```
    myName = (name == null) ? "Unknown" : name;
    myPrice = (price == null) ? BigDecimal.ZERO : price;
}
```

Solution: Throw exceptions for invalid input.

```
// GOOD: Exposes bugs immediately
public StoreItem(String name, BigDecimal price) {
    myName = Objects.requireNonNull(name, "Name cannot be null");
    myPrice = Objects.requireNonNull(price, "Price cannot be null");
}
```

Mistake 2: Wrong Exception Type

Problem: Using `IllegalArgumentException` for null.

```
// BAD: Wrong exception type
if (name == null) {
    throw new IllegalArgumentException("Name cannot be null");
}
```

Solution: Use the exception type specified in the API.

```
// GOOD: Correct exception type
Objects.requireNonNull(name, "Name cannot be null");  // Throws NPE
```

Mistake 3: Validating After Assignment

Problem: Assigning fields before validation completes.

```
// BAD: Object partially constructed if validation fails
public StoreItem(String name, BigDecimal price) {
    myName = name;  // Assigned before validation!
    Objects.requireNonNull(name);  // Too late
    myPrice = price;
}
```

Solution: Validate before any assignment.

```
// GOOD: Validate first, assign after
public StoreItem(String name, BigDecimal price) {
    Objects.requireNonNull(name, "Name cannot be null");
    Objects.requireNonNull(price, "Price cannot be null");
    myName = name;
    myPrice = price;
}
```

Mistake 4: Redundant Validation in Private Methods

Problem: Re-validating data that was already checked.

```
// WASTEFUL: addInternal is only called from add, which already validates
public void add(Item item, int qty) {
    Objects.requireNonNull(item);
    if (qty <= 0) throw new IllegalArgumentException();
    addInternal(item, qty);
}

private void addInternal(Item item, int qty) {
    Objects.requireNonNull(item);  // Redundant!
    if (qty <= 0) throw new IllegalArgumentException();  // Redundant!
    // ...
}
```

Solution: Trust the public boundary validation.

```
// CLEAN: Trust that add() validated
private void addInternal(Item item, int qty) {
    // Implementation: add item to internal collection
}
```

Mistake 5: No Message in Exception

Problem: Exception provides no context.

```
// BAD: No message
throw new IllegalArgumentException();
Objects.requireNonNull(name);
```

Solution: Always include descriptive messages.

```
// GOOD: Helpful messages
throw new IllegalArgumentException("Quantity must be positive: " + qty);
Objects.requireNonNull(name, "Name cannot be null");
```

Summary

Concept	Key Point
Fail fast	Detect and report errors immediately at boundaries

Concept	Key Point
Trust boundaries	Validate at public methods; trust private methods
Objects.requireNonNull()	Standard idiom for null checks; explicit better than implicit
IllegalArgumentException	For semantic violations (empty, negative, out of range)
NullPointerException	Specifically for null arguments
Validation order	Null checks first, then semantic checks
Private methods	Don't re-validate—trust the public boundary
Exception messages	Include context: what was expected, what was received

Defensive programming makes bugs visible. When something goes wrong, you want to know **immediately** and **exactly where**. Validate at boundaries, fail fast, and trust your internal code.

Further Reading

External Resources

- Objects.requireNonNull Javadoc - Official documentation for null checking
- Effective Java, Item 49: Check parameters for validity - Joshua Bloch's definitive guidance
- Baeldung: IllegalArgumentException vs NullPointerException - When to use which exception
- Defensive Programming (Wikipedia) - Overview of defensive techniques

---

References

Primary Texts:

- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. Item 49: Check parameters for validity; Item 72: Favor the use of standard exceptions; Item 73: Throw exceptions appropriate to the abstraction.

- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals* (12th ed.). Oracle Press. Chapter 7: Exceptions, Assertions, and Logging.

Language Documentation:

- Oracle JDK 25: java.util.Objects — Objects utility class including requireNonNull()
- Oracle JDK 25: NullPointerException — NullPointerException documentation
- Oracle JDK 25: IllegalArgumentException — IllegalArgumentException documentation

Design Principles:

- Fail-fast (Wikipedia) — Overview of the fail-fast design principle
- Design by Contract — The broader framework for preconditions and postconditions

---

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