LECTURE 5

# Exceptions & Design by Contract

Advanced Object-Oriented Programming

Duration: 4 Hours | SmartShelf v0.5: Robust Error Handling

# Today's Agenda

4 Hours - Making SmartShelf Robust & Reliable

**Hour 1**

### Exception Fundamentals

Checked vs unchecked, throw/catch mechanics, exception hierarchy

**Hour 2**

### Custom Exception Hierarchy

SmartShelfException, ItemNotFoundException, rich context, error codes

**Hour 3**

### Design by Contract (DbC)

Preconditions, postconditions, class invariants, Contracts utility

**Hour 4**

### Workshop: SmartShelf v0.5

Tasks A-F: exception hierarchy, DbC integration, exception-to-Result

# Recap - SmartShelf v0.4

Generics + LSP from Week 4 + this week's problem

| | |
|---|---|
| **Catalog<T>** | Type-safe generic catalog with bounded parameters |
| **Pair<K,V>** | Generic utility for loan records, mappings |
| **Result<T>** | Success/failure wrapper - no exceptions for expected failures |
| **LSP** | Subtypes must honor parent contracts - tested and verified |

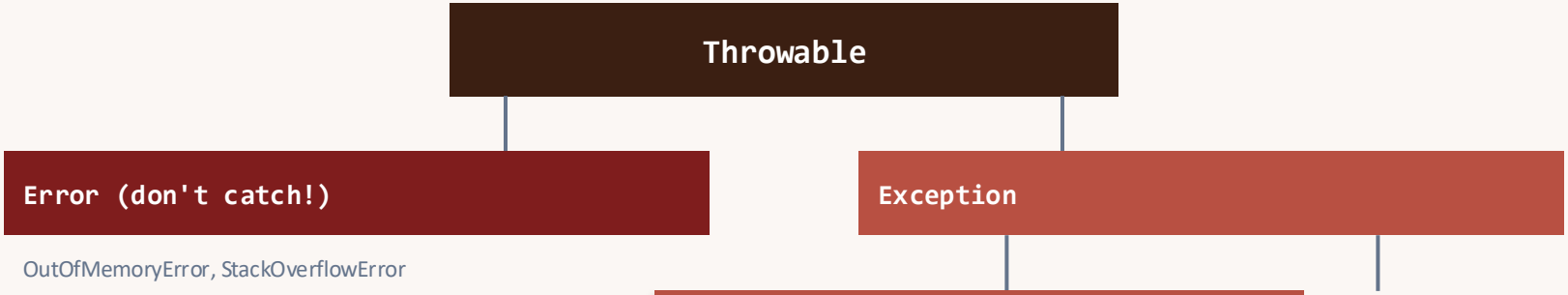
**Problem:** v0.4 throws generic IllegalStateException for everything. borrowItem() fails? Same exception whether member missing, item unavailable, or limit exceeded. Caller can't tell what went wrong or how to recover.

# Hour 1

Exception Fundamentals

# Java Exception Hierarchy

Three categories with different handling requirements




**When to use which?**

CHECKED: Recoverable business situations the caller should handle.

UNCHECKED: Programming bugs that should be fixed, not caught.

# Checked vs Unchecked - Decision Guide

The single most important exception design decision

| | Checked | Unchecked (Runtime) |
|---|---|---|
| **Extends** | Exception | RuntimeException |
| **Compiler** | Forces try/catch or throws | No enforcement |
| **Use for** | Business logic errors | Programming bugs |
| **Recoverable?** | Yes - caller can handle | No - fix the code |
| **SmartShelf** | ItemNotFoundException | ContractViolationException |
| **Example** | "Book not in catalog" | "Null ISBN passed" |
| **Caller says** | "I'll try another ISBN" | "Developer: fix your code" |

**Effective Java Item 70:** *"Use checked exceptions for recoverable conditions and runtime exceptions for programming errors."*

# Before & After: Error Handling

v0.4 generic exceptions vs v0.5 custom exception hierarchy

**v0.4 - Generic Exceptions**

```
try {
  library.borrowItem(id, isbn);
} catch (
IllegalStateException e) {
  // What went wrong?
  // Member missing? Item gone?
  // Limit reached? Already borrowed?
  // ALL SAME EXCEPTION TYPE!
  // Can't tell. Can't recover.
  System.out.println(e.getMessage());
}
```

One catch, no context, no recovery strategy

**v0.5 - Custom Exceptions**

```
try {
  library.borrowItem(id, isbn);
} catch (
MemberNotFound e) {
  redirect(e.getMemberId());
} catch (
ItemNotAvailable e) {
  showWaitlist(e.getItemTitle());
} catch (
BorrowLimitExceeded e) {
  show(e.getCurrentCount(),e.getMax());
}
```

Specific catches, rich context, targeted recovery

# Hour 2

Custom Exception Hierarchy

# SmartShelf Exception Hierarchy

Checked for business errors, unchecked for programming bugs

**SmartShelfException (checked)**

| | |
|---|---|
| ItemNotFoundException | ISBN context |
| MemberNotFoundException | memberId context |
| ItemNotAvailableException | title context |
| BorrowLimitExceededException | count/max context |
| DuplicateItemException | ISBN context |

**All share errorCode field**

"ITEM_NOT_FOUND", "BORROW_LIMIT_EXCEEDED"
Useful for API responses, logging, metrics.

**ContractViolationException (unchecked)**

| | |
|---|---|
| PRECONDITION | null params, blank strings |
| POSTCONDITION | return value guarantees |
| INVARIANT | object state consistency |

**Key Design Rule**

Each exception carries context relevant to recovery. ItemNotFoundException has getIsbn(). BorrowLimitExceededException has getCurrentCount() and getMaxLimit().

# Rich Exception Context - Code

Each exception carries structured data for recovery, not just a message string

```
public class BorrowLimitExceededException extends SmartShelfException {
  private final String memberName;
  private final int currentCount;
  private final int maxLimit;

  public BorrowLimitExceededException(String name, int count, int max) {
    super(name + " has reached limit (" + count + "/" + max + ")",
          "BORROW_LIMIT_EXCEEDED");
    this.memberName = name;
    this.currentCount = count;
    this.maxLimit = max;
  }

  // Structured getters for caller recovery logic
  public String getMemberName()  { return memberName; }
  public int getCurrentCount()   { return currentCount; }
  public int getMaxLimit()       { return maxLimit; }
}
```

# Exception Catching Patterns

Three ways to handle SmartShelf exceptions

**1. Specific catches**

```
catch (ItemNotFound e)
  → redirect
catch (NotAvailable e)
  → waitlist
catch (LimitExceeded e)
  → show count
```

**2. Multi-catch**

```
catch (
  MemberNotFound |
  ItemNotFound e
) {
  // Handle both
  // "not found" cases
  log(e.getErrorCode());
}
```

**3. Base catch-all**

```
catch (
  SmartShelfException e
) {
  // Catches ALL
  // business errors
  show(e.getErrorCode(),
       e.getMessage());
}
```

### Best Practices

Catch specific first, general last. Never catch Exception (too broad). Never swallow exceptions (empty catch). Always log or rethrow. Use multi-catch for same recovery logic. Prefer checked for public APIs, unchecked for internal assertions.

# Exception-to-Result Conversion

Bridge between exception-based and functional error handling

```
// Library offers BOTH styles:

// Style 1: Traditional — caller uses try/catch
public void borrowItem(...) throws SmartShelfException { ... }

// Style 2: Functional — exceptions converted to Result<T>
public Result<MediaItem> tryBorrow(String id, ISBN isbn) {
  try {
    borrowItem(id, isbn);
    return Result.success(catalog.findByIsbn(isbn));
  } catch (SmartShelfException e) {
    return Result.failure("[" + e.getErrorCode() + "] " + e.getMessage());
  }
}
```

**When to use exceptions**

Public API boundaries, framework code, when caller MUST handle the error.

**When to use Result<T>**

Internal code, expected failures, functional pipelines, when errors are data not control flow.

# Critical Evaluation - Exceptions

| Aspect | Utility | Pitfall |
|---|---|---|
| Custom hierarchy | Specific catches, targeted recovery | Too many exception classes |
| Checked | Compiler enforces handling | Verbose, clutters signatures |
| Unchecked | Clean code, no clutter | Silent failures if not caught |
| Error codes | Machine-readable, API-friendly | Extra field to maintain |
| Rich context | Structured recovery data | Overengineering risk |

**LO1: Critically evaluate**

Checked exceptions enforce handling but add verbosity. Custom hierarchies aid recovery but risk over-engineering. The exception-to-Result pattern bridges both worlds. Argue trade-offs in assessments.

# Hour 3

Design by Contract (DbC)

# Design by Contract - Bertrand Meyer

Every method is a contract between caller and implementation

**PRECONDITION** *"I require this from you"*

What the caller must provide — Who: Caller's responsibility

```
Example: isbn != null, memberId not blank
```

**POSTCONDITION** *"I guarantee this to you"*

What the method guarantees — Who: Method's responsibility

```
Example: item findable after add(), count increases by 1
```

**INVARIANT** *"This is always true about me"*

What is always true about the object — Who: Class's responsibility

```
Example: catalog != null, borrowCount >= 0, name not blank
```

**DbC violations are programming bugs (unchecked) — not business errors (checked).**

# Contracts Utility Class

Static methods for preconditions, postconditions, and invariants

```
public final class Contracts {

  // Precondition: what caller must provide
  public static void require(boolean cond, String desc) {
    if (!cond) throw new ContractViolationException(PRECONDITION, desc);
  }

  // Postcondition: what method guarantees
  public static void ensure(boolean cond, String desc) {
    if (!cond) throw new ContractViolationException(POSTCONDITION, desc);
  }

  // Invariant: what is always true about the object
  public static void invariant(boolean cond, String desc) {
    if (!cond) throw new ContractViolationException(INVARIANT, desc);
  }

  // Convenience: requireNonNull, requireNotBlank, requireInRange
}
```

# DbC Applied: Library.borrowItem()

Preconditions at entry, postconditions at exit, invariant always

```
public void borrowItem(String memberId, ISBN isbn)
    throws MemberNotFound, ItemNotFound, NotAvailable, LimitExceeded {

  // ── PRECONDITIONS (unchecked — programming bugs) ──
  Contracts.requireNotBlank(memberId, "memberId");
  Contracts.requireNonNull(isbn, "isbn");

  // ── BUSINESS LOGIC (checked — recoverable errors) ──
  Member member = findMemberOrThrow(memberId);
  MediaItem item = findItemByIsbn(isbn);
  if (!item.isAvailable()) throw new ItemNotAvailableException(...);
  if (!member.canBorrow()) throw new BorrowLimitExceededException(...);

  int prevCount = member.getBorrowedCount();
  item.markAsBorrowed(); member.addBorrowedItem(item);

  // ── POSTCONDITIONS (unchecked — verify guarantees) ──
  Contracts.ensure(!item.isAvailable(), "item must be borrowed");
  Contracts.ensure(member.getBorrowedCount() == prevCount + 1, "count +1");
  checkInvariant();
}
```

# Critical Evaluation - DbC

| Aspect | Utility | Pitfall |
|---|---|---|
| Preconditions | Catch bugs at method entry, clear API | Verbose, duplicate if(null) checks |
| Postconditions | Self-testing code, catches logic bugs | Performance overhead, hard to express |
| Invariants | Guaranteed object consistency | Called after every mutation = cost |
| Contracts class | Readable, reusable, centralized | Extra dependency, non-standard Java |

**DbC in Industry**

Eiffel language has native DbC. Java uses assert keyword (limited), Google Guava Preconditions, or custom Contracts like ours. Spring Framework uses @Valid annotations. Production code often disables postconditions/invariants for performance but keeps preconditions always-on.

# Hour 4

Workshop: SmartShelf v0.5

# Workshop Tasks A - C

Build exception hierarchy + contracts - 45 minutes

Task A

### Create SmartShelfException base class

Checked exception with errorCode field. toString() includes code. Constructor accepts message + code + optional cause.

Task B

### Create 5 specific exception subclasses

ItemNotFound(ISBN), MemberNotFound(memberId), BorrowLimitExceeded(name, count, max), ItemNotAvailable(title), DuplicateItem(ISBN).

Task C

### Create Contracts utility + ContractViolationException

require(), ensure(), invariant() static methods. Unchecked exception with ContractType enum. requireNonNull, requireNotBlank, requireInRange helpers.

# Workshop Tasks D - F

Integration and testing - 45 minutes

**Task D**

### Update Library with checked exceptions

borrowItem() throws MemberNotFound|ItemNotFound|NotAvailable|LimitExceeded. addItem() throws DuplicateItemException. findItemByIsbn() throws ItemNotFoundException.

**Task E**

### Add DbC to Library methods

Preconditions (requireNonNull, requireNotBlank) at entry. Postconditions (ensure) before return. checkInvariant() after every mutation.

**Task F**

### Full integration demo - 8 demos

Specific catches, rich context, checked vs unchecked, multi-catch, DbC preconditions, postconditions/invariants, exception-to-Result, full integration.

# SmartShelf v0.5 Project Structure

```
src/smartshelf/
  exceptions/
    SmartShelfException.java      Task A
    ItemNotFoundException.java    Task B
    MemberNotFoundException.java  Task B
    BorrowLimitExceededException  Task B
    ItemNotAvailableException     Task B
    DuplicateItemException.java   Task B
    ContractViolationException    Task C
  contract/
    Contracts.java                Task C
  generics/
    Catalog.java, Pair.java, Result.java
  Library.java                    Task D+E
  Main.java                       Task F
  (12 base classes from Weeks 1-3)
```

### New This Week

7 exception classes
1 Contracts utility
Library uses checked exceptions
Library uses DbC contracts
tryBorrow() exception-to-Result
25 source files total

### Run

```
chmod +x run.sh
./run.sh

Requires: Java 17+
```

# What's Coming Next

## Week 6: Creational Design Patterns

### SmartShelf: MediaItem Factory + Builder

Factory Method creates Book/Journal/DVD without knowing the exact class. Builder pattern constructs complex MediaItem objects step by step. Singleton for Library configuration. Abstract Factory for themed UI components.

### Preparation

Read Head First Design Patterns Ch. 4 (Factory). Complete Tasks A-F and push to repo. Review static factory methods in Effective Java Item 1.

# Lecture 5 Summary

📋 Custom exceptions carry context: ISBN, memberId, count/limit

⚡ Checked = business errors (must handle), Unchecked = bugs (fix code)

🧰 SmartShelfException hierarchy: 5 specific checked exceptions

🎯 Multi-catch + base catch-all for flexible error handling

🔒 Preconditions: require() at method entry — caller's promise

✅ Postconditions: ensure() before return — method's guarantee

🛡 Invariants: invariant() after mutations — always-true state

🔄 Exception-to-Result bridges exception and functional styles

*SmartShelf v0.5 — robust, recoverable, self-checking*

# Questions?

Next Week: Creational Design Patterns