LECTURE 4

Generics & Liskov Substitution Principle

Advanced Object-Oriented Programming

Duration: 4 Hours | SmartShelf v0.4: Type-Safe Collections

Today's Agenda

4 Hours - Making Collections Type-Safe

Hour 1

Why Generics?

Type parameters, generic classes, Catalog<T extends MediaItem>

Hour 2

Bounded Types, Wildcards & Erasure

PECS principle, ? extends/super, type erasure at runtime

Hour 3

Liskov Substitution Principle

LSP definition, Rectangle/Square, SmartShelf violations, fixes

Hour 4

Workshop: SmartShelf v0.4

Tasks A-F: Catalog<T>, Pair<K,V>, Result<T>, LSP-safe hierarchy

Recap - SmartShelf v0.3

What we built + this week's new challenge

MediaItem	Abstract parent with polymorphic display()
Book/Journal/DVD	Concrete subtypes with overridden display()
DigitalMedia	Implements Downloadable interface
Library	Polymorphic catalog - List<MediaItem>, no instanceof

New Challenge: Catalog holds any MediaItem - but what if we want a Books-only catalog that rejects DVDs at compile time? And when is a subclass NOT safely substitutable for its parent?

Hour 1

Why Generics?

The Problem Without Generics

List<Object> gives no type safety - casting required everywhere

Without Generics

```
List catalog = new ArrayList();
catalog.add(new Book(...));
catalog.add("I'm a String!");
catalog.add(42);

// Must cast + pray
Book b = (Book) catalog.get(0);
// ClassCastException at runtime!
Book x = (Book) catalog.get(1);
```

With Generics

```
Catalog<Book> catalog = new Catalog<>();
catalog.add(new Book(...));
catalog.add("String");  // COMPILE ERROR!
catalog.add(42);        // COMPILE ERROR!

// No casting needed!
Book b = catalog.findByIsbn(isbn);
// Type-safe at compile time
```

Runtime errors, unsafe casts, no compiler help

Compile-time safety, no casting, type guaranteed

Generic Class Syntax

Type parameter T is a placeholder - replaced at usage time

```
public class Catalog<T extends MediaItem> {

  private final List<T> items;      // T = placeholder

  public void add(T item) { ... }   // Accepts only T

  public T findByIsbn(ISBN isbn) {   // Returns T directly
    for (T item : items) {           // Iterate as T
      if (item.getIsbn().equals(isbn))
        return item;                 // No casting!
    }
    return null;
  }

  // T extends MediaItem → can call getIsbn(), getTitle(), display()
  public List<T> searchByTitle(String kw) { ... }
}
```

Bounded vs Unbounded Type Parameters

extends keyword constrains what types T can be

	Bounded	Unbounded
Syntax	<T extends MediaItem>	<K, V>
Constraint	T must be MediaItem or subtype	Any type allowed
Can call on T	getIsbn(), display(), getTitle()	Only Object methods
SmartShelf	Catalog<T extends MediaItem>	Pair<K, V>, Result<T>
Use when	Need to call type-specific methods	Container only stores/returns

Effective Java Item 31: *"Use bounded wildcards to increase API flexibility."*

Hour 2

Bounded Types, Wildcards & Erasure

Wildcards - PECS Principle

Producer Extends, Consumer Super - the wildcard decision rule

? extends T (Producer)

```
// Read FROM source
void addAll(Catalog<? extends T> src){
  for (T item : src.getAll()) {
    this.add(item);  // READ ok
  }
  // src.add(x);  COMPILE ERROR!
}
```

Use when you READ from the parameter.
Catalog<Book> works as Catalog<? extends MediaItem>

? super T (Consumer)

```
// Write INTO destination
void copyTo(Catalog<? super T> dest) {
  for (T item : this.getAll()) {
    dest.add(item);  // WRITE ok
  }
  // T x = dest.get(0); ERROR!
}
```

Use when you WRITE to the parameter.
Catalog<MediaItem> works as Catalog<? super Book>

PECS Memory Aid:

If the parameter PRODUCES values you read: ? extends T. If it CONSUMES values you write: ? super T. If both: use exact type T.

Type Erasure - What Happens at Runtime

Java erases generic type info after compilation - important limitations

```
// At compile time:
Catalog<Book> books = new Catalog<>();
Catalog<DVD>  dvds  = new Catalog<>();

// At runtime (after erasure):
Catalog books = new Catalog();  // raw!
Catalog dvds  = new Catalog();  // same!

books.getClass() == dvds.getClass() // true!
```

Cannot Do

```
new T()              // no constructor
T[].class            // no array type
instanceof Catalog<Book> // no check
Catalog<int>         // no primitives
```

Why Erasure?

Backward compatibility with pre-Java-5 code. Existing bytecode works unchanged.

Generic Methods & Type Inference

Methods can have their own type parameters, independent of the class

```
// Generic static factory method on Pair
public static <K, V> Pair<K, V> of(K first, V second) {
  return new Pair<>(first, second);
}

// Usage — compiler infers K=Member, V=Book
Pair<Member, Book> loan = Pair.of(alice, book1);


// Generic static factory on Result
public static <T> Result<T> success(T value) { ... }
public static <T> Result<T> failure(String error) { ... }

// Usage — compiler infers T=MediaItem
Result<MediaItem> r = Result.success(book1);
```

SmartShelf: Pair<K,V> & Result<T>

Multiple type parameters + generic error handling

Pair<K, V>

Pair<Member, Book> loan

Pair<ISBN, String> mapping

Pair<String, Integer> stat

K and V are unbounded - any types.
Static Pair.of() infers types.

Result<T>

```
Result<MediaItem> r = tryBorrow();


if (r.isSuccess())
  item = r.getValue();  // T!
else
  msg = r.getError();
```

No exceptions for expected failures.
map() transforms: Result<Book> -> Result<String>

```
// Library uses both:
List<Pair<Member, MediaItem>> loanHistory = new ArrayList<>();
public Result<MediaItem> tryBorrow(String memberId, ISBN isbn) { ... }
```

Critical Evaluation - Generics

Aspect	Utility	Pitfall
Type safety	Compile-time errors > runtime ClassCastException	Type erasure: no runtime type info
Bounded params	Can call methods on T (getIsbn, display)	Complex bounds confuse readers
Wildcards	Flexible APIs (PECS)	Hard to reason about <? super T>
Reusability	One class works for Book, DVD, MediaItem	Over-generification: premature abstraction

LO1: Critically evaluate

Generics trade runtime flexibility for compile-time safety. Always argue both sides in assessments: type safety vs erasure limitations, PECS flexibility vs readability cost.

Hour 3

Liskov Substitution Principle

Liskov Substitution Principle (LSP)

Barbara Liskov, 1987 - the "L" in SOLID

> *If S is a subtype of T, then objects of type T may be replaced with objects of type S without altering the correctness of the program.*

Translation: A subclass must behave correctly wherever the parent class is expected. No surprises, no broken contracts, no new exceptions.

1. Preconditions cannot be strengthened in subclass
2. Postconditions cannot be weakened in subclass
3. Invariants of parent must be preserved
4. No new exceptions that parent doesn't throw
5. If you check instanceof, suspect LSP violation

Classic Violation: Rectangle/Square

Square "is-a" Rectangle? Mathematically yes, in OOP no!

```
class Rectangle {
  void setWidth(int w)  { this.width = w; }
  void setHeight(int h) { this.height = h; }
}

class Square extends Rectangle {
  void setWidth(int w)  { width = w; height = w; }
  void setHeight(int h) { width = h; height = h; }
}

// Client code:
r.setWidth(5); r.setHeight(10);
expect(r.area() == 50);  // FAILS for Square!
```

Why It Breaks

Rectangle contract: setWidth doesn't affect height. Square violates this. Code expecting Rectangle breaks with Square.

Fix

Don't make Square extend Rectangle. Use a Shape interface with area(). Or make both immutable.

SmartShelf: ReadOnlyMedia Violation

A reference-only item that throws on markAsBorrowed()

BAD - LSP Violation

```
class ReadOnlyMedia extends MediaItem {
  @Override
  public void markAsBorrowed() {
    throw new UnsupportedOperation
      Exception("Cannot borrow!");
  }
}
```

GOOD - Interface Separation

```
interface Borrowable {
  void borrow();
  void returnItem();
}

class Book implements Borrowable
class ReferenceItem // no borrow!
// Type system prevents violation
```

LSP + Generics Connection: List<Book> is NOT a subtype of List<MediaItem> even though Book extends MediaItem. This is generic invariance - Java's way of preventing LSP violations in collections.

Generic Invariance Protects LSP

Why List<Book> is not List<MediaItem> - and why that's correct

```
// If Java allowed this (it doesn't!):
List<Book> books = new ArrayList<>();
List<MediaItem> items = books;   // Hypothetically...

items.add(new DVD(...));          // DVD into Book list!
Book b = books.get(0);           // ClassCastException!


// Java's solution: wildcards for safe covariance
List<? extends MediaItem> safe = books;  // READ-only
// safe.add(new DVD(...));  COMPILE ERROR - safe!
MediaItem item = safe.get(0);    // READ works fine
```

Invariance + wildcards = type safety + flexibility. Generics enforce LSP at compile time.

Hour 4

Workshop: SmartShelf v0.4

Workshop Tasks A - C

Build generic classes - 45 minutes

Task A

Create Catalog<T extends MediaItem>

Generic catalog: add(), findByIsbn(), searchByTitle(). T bounded to MediaItem. Returns T not Object.

Task B

Create Pair<K,V> with static factory

Two unbounded type parameters. Pair.of() uses type inference. equals(), hashCode(), toString().

Task C

Create Result<T> success/failure wrapper

Static factories: success(T), failure(String). map() transforms. No exceptions for expected failures.

Workshop Tasks D - F

LSP and integration - 45 minutes

Task D

Update Library to use Catalog<MediaItem>

Replace List<MediaItem> with Catalog<MediaItem>. Add tryBorrow() returning Result<MediaItem>. Loan history as Pair.

Task E

Implement wildcard methods

addAll(Catalog<? extends T>) and copyAvailableTo(Catalog<? super T>). Test with Catalog<Book> into Catalog<MediaItem>.

Task F

LSP demo + full integration

Rectangle/Square violation. ReadOnlyMedia violation. Fix with Borrowable interface. 8-demo Main.java.

SmartShelf v0.4 Project Structure

```
SmartShelf-Lecture4/
  src/smartshelf/
    ISBN.java              Week 1
    Author.java            Week 2
    MediaItem.java         Week 3
    Book/Journal/DVD       Week 3
    DigitalMedia.java      Week 3
    Member.java            updated
    Library.java           updated ★
    Main.java              8 demos ★
    generics/
      Catalog.java         Task A ★
      Pair.java            Task B ★
      Result.java          Task C ★
    lsp/
      LspDemo.java         Task F ★
```

New This Week

★ Catalog<T extends MediaItem>
★ Pair<K, V>
★ Result<T>
★ LspDemo (violations + fixes)
★ Library uses Catalog + Result
★ Wildcard methods (PECS)

Run

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

Or:
javac -d bin $(find src -name '*.java')
java -cp bin smartshelf.Main
```

What's Coming Next

Week 5: Exceptions & Design by Contract

SmartShelf: Custom Exception Hierarchy

SmartShelf will use custom exceptions (BookNotFoundException, BorrowLimitExceededException) with Design by Contract: preconditions, postconditions, and class invariants enforced programmatically.

Preparation

Read Effective Java Ch. 10 (Exceptions). Complete Tasks A-F and push to repo. Review checked vs unchecked exceptions.

Lecture 4 Summary

🔒 Generics give compile-time type safety - no casting

📦 <T extends MediaItem> bounds what T can be

🔗 Pair<K,V> and Result<T> - multiple type parameters

📐 PECS: Producer Extends, Consumer Super

🔍 Type erasure: generics gone at runtime

⚖️ LSP: subtypes must honor parent contracts

❌ Rectangle/Square + ReadOnlyMedia = classic violations

✅ Fix LSP via interfaces (Borrowable) not inheritance

SmartShelf v0.4 - type-safe + LSP-compliant

Questions?

Next Week: Exceptions & Design by Contract