COSC 3P91 – Advanced Object-Oriented Programming
Laboratory 4: Interfaces, Polymorphism, and Generics

Department of Computer Science
Brock University

Winter Term

Overview

This laboratory focuses on bridging the gap between rigid class hierarchies and flexible, decoupled system architectures. While Laboratory 2 focused on structural design via UML, Laboratory 3 requires students to implement these designs using Java's most powerful abstraction tools: Interfaces and Generics. Students will refactor legacy inheritance-based code into a contract-driven system that utilizes multiple interface implementation, default behaviors, and type-safe generic containers to ensure system scalability and robustness.

Learning Objectives

By completing this laboratory, students will be able to:

- Apply interface-based design to decouple unrelated components that share common behaviors.
- Implement multiple interfaces to define multifaceted object behaviors.
- Utilize default and static interface methods to provide shared logic without breaking class hierarchies.
- Enforce compile-time type safety using Bounded Generic types and multiple bounds.
- Demonstrate polymorphism by using interface types as reference variables.

Given Problem Description: The "Secure Messaging Ecosystem"

You are tasked with refactoring a legacy notification system into a modern, decoupled messaging framework.

Legacy System: The existing system uses a single `Notification` abstract class. This design is too rigid and prevents objects from sharing behaviors if they are not in the same inheritance tree.

The New Contract Specification:

- **Informable:** A core interface ensuring all messages can provide a `getContent()` string. It includes a `default` method `preview()` to display a snippet of the content.
- **Sendable:** An interface for objects that can be dispatched to a specific destination.
- **Loggable:** A utility interface that provides `static` methods for timestamping system events.
- **Prioritizable:** Uses a `Priority` Enum (LOW, MEDIUM, HIGH) to categorize the urgency of an object.

Object Implementation: You must implement several classes—such as `UrgentPage` and `CustomerAlert`—that implement various combinations of these interfaces to demonstrate multiple inheritance of behavior.

Generic Processing:

- **SecurityScanner:** A generic class restricted to types that are both `Informable` AND `Loggable` (Multiple Bounds).
- **ProcessingResult:** A utility class using multiple type parameters to track the status and the message object itself.

Analysis Tasks

Students must answer the following questions to verify their understanding of the implementation:

1. What is the primary difference between using an interface and an abstract class for this messaging system?
2. Why is a `static` method in the `Loggable` interface called via the interface name rather than an instance?
3. How does the `default` keyword in `Informable` allow for system expansion without breaking legacy implementations?
4. In the `SecurityScanner<T extends Informable & Loggable>`, what happens to the type `T` at runtime?
5. Why is it illegal to instantiate a generic type parameter (e.g., `new T()`)?

Required Task

Students must produce Java source code and a written response for the MCQs. The solution must:

1. Refactor the provided legacy code into the new interface-driven design.
2. Correctly implement at least one class that realizes multiple interfaces.
3. Demonstrate the use of the diamond operator `<>` for type inference during instantiation.
4. Implement a bounded generic class that enforces strict type checks.
5. Provide a `Main` class that uses interface references to store disparate objects in a single polymorphic collection.

Key Takeaways

This laboratory emphasizes that code stability is achieved through strict contracts and type safety. By moving away from inheritance and toward interfaces, developers create "unrelated classes implementing the same behavior". Furthermore, by utilizing Generics and Type Erasure, students learn how Java maintains backward compatibility while providing powerful compile-time checks that prevent runtime errors.