Course Review

Advanced OO Programming – COSC 3P91

Introduction to Object Oriented Programming

- **Design Principles**
  - Abstraction Principle
  - Program to an Interface
  - Favour Composition over Inheritance
- **Data Abstraction**
  - Abstract Data Types
- Encapsulation
- Object-Oriented Concepts
  - Classes
    - static modifiers
    - Nested Classes
      - Local classes
      - Anonymous Classes
    - Abstract Classes
  - **Inheritance**
    - **Subtyping**
    - Inheritance for specification, Inheritance for extension, Inheritance for specialization

Introduction to Object Oriented Programming

- Unified Modeling Language
  - Definition
  - Purpose
  - Benefits
  - **Class diagrams**
    - **Elements and Relationships**
    - Notation
    - Relationship Variations
      - Inheritance
      - Composition, aggregation, association, dependency

Generics, Polymorphism, and Interfaces

- Interfaces
  - **Definition**
  - **Difference between Abstract classes and interfaces**
  - Interface as a Type
  - Implementing and extending interfaces
  - **Abstract** Methods
  - **Default** Methods
  - **Static** Methods
- Overriding versus overloading
- Overriding and hiding
- **Final Classes and Methods**
- Enumeration types

Generics, Polymorphism, and Interfaces

- Generics
  - Definition
  - **Generic Type**
    - **Type parameter and type argument**
  - Generic Class
  - Instantiating a Generic Type
  - Multiple Type Parameters
  - Raw Types
  - **Generic Methods**
  - **Bounded Type Parameters**
  - **Generic Subtypes**
  - Type Inference
  - Target Types
  - Restrictions on Generics
  - **Wildcards**
    - Bounded Wildcards
    - Wildcards and Subtyping

Utility Classes, Collections, Files, and Streams

- **Lambda Expressions**
  - Anonymous Classes?
  - Functional Interfaces in Java
  - **Syntax**
  - Parameters
  - Method References
  - Lambda Expressions and Method References
- Utility Classes
  - **Collection**
  - Set
  - **List**

Utility Classes, Collections, Files, and Streams

- Input and Output in Java
  - **Streams**
    - Input
    - Output
    - SequenceInputStream
    - Filtering
  - Piped input and output
  - Character Streams

Exception Handling

- **Definition**
- Best Practices
- Exception Handler
- Catching Exceptions
- Types of Exceptions
  - Checked exception
  - Unchecked exception
- **try Block**
- **catch Block**
- **finally Block**
- try-with-resources Statement
- Stack Winding
- **Throwing Exceptions**

Exception Handling

- **Chained Exceptions**
- Logging
- Creating Exception Classes
- **Advantages** of Exceptions
  - Separating Error-Handling Code from "Regular" Code
  - Propagating Errors Up the Call Stack
  - Grouping and Differentiating Error Types
- **Java Exception Antipatterns**
- Assertions
  - Simple
  - Complex

Databases and JDBC

- Java Database Connectivity → **JDBC**
  - The API → Programmatic
    - Connection and logging with database → **DriverManager**
    - Making SQL queries → **Statement**
    - Retrieval → **ResultSet**
  - JDBC-**ODBC** bridging
- JDBC **Architectures**
  - **Two-tier** → directly connected with the DBMS (remote)
    - Client/server architecture
  - **Three-tier** → JDBC separated from the application
    - Purpose → control, simplification (decoupling), performance
- Relational DBMS → **SQL** → tables, attributes, rows
- **Transactions** → better control
  - Commit and rollback → consistency and concurrency
- **Performance tips**/principles

Design Patterns

- **Definition** → test and proven paradigms → best practices
  - **Purpose** → successful use, reusability, they are expressive
- The three categories
  - Creational, structural, and behavioral
- **Creational** → instantiation of entities/objects
  - Patterns → reason for it, how it works, problems
    - **Factory**
    - Abstract Factory
    - **Builder**
    - Prototype
    - **Singleton**

Design Patterns

- **Structural** → grouping of entities
  - Patterns → reason for it, how it works, problems
    - **Adapter**
      - Class and object
    - Decorator
    - **Façade**
    - **Composite**
- **MVC Architectural Pattern**
  - Model, View, Controller → interfaces / event-driven apps
    - Role of components and interactions

Design Patterns

- **Behavioral** → relationship among entities
  - Loosely coupled design → flexibility
  - Patterns → reason for it, how it works, problems
    - **Null Object**
    - **Command**
    - **Strategy**
    - **Observer**
      - Listener → MVC
- eXtensible Markup Language → data description
  - Namespaces and schema
  - Processing with Java → parsing XMLs
    - **DOM** → Document Object Model
      - Interface Node
      - Alternative → Simple API for XML
    - JAXP → Java API for XML Processing
      - **Transformers**

User Interfaces

- Abstract Window Toolkit (**AWT**) vs **Swing**
  - Their roles → opt for swing
- Swing base classes → containers
- JavaFX → rich Internet-like applications
- 2D Graphics in Java
  - Rendering
  - Animations → **page flipping** and **BufferStrategy**
    - Screen, printing, Sprites
- **Interactivity** → AWT event model → events and listeners
- **13 Principles of GUI Design**
  - Perceptual principles, Mental model principles, Attention principles, and memory principles

Concurrency

- Java **Threads** → definition
  - Thread vs process
  - **Benefit in multithreading**
  - **Nondeterministic ordering**
  - **Execution states**
- Creating threads → Runnable Interface or suclass Thread
  - Start() method in both
  - **Runnable** implementation over **Thread** extension
- Interrupts with threads
  - **Sleep** and **join**
- **Thread Safety**
- **Synchronization** → Why needed?
  - Thread contention
  - Major problems → starvation, livelock, and deadlock

Concurrency

- **Critical region, race condition, thread interleave**
- **Synchronized** → methods and blocks/statements
  - Object Intrinsic Lock → mutual exclusion (obj as a resource)
  - Reentrant synchronization
- Atomic access → **volatile** → why is this interesting?
- **Liveness failures** → deadlock, contention, dormancy, premature termination
- **Guarded blocks** → wait() and notify()
- Immutable objects → benefit in concurrency and strategies
- High Level Concurrency Objects
  - **Lock**, **Executors** (thread pools), Atomic Variables, ...

Concurrency

- Advanced Java Synchronizers
  - **Semaphore** → difference from a lock
    - **Acquire** and **release**
    - **Types of semaphores**
  - **CountDownLatch** → barrier
  - **CyclicBarrier** → barrier
- Concurrency design patterns
  - Guarded suspension
  - **Producer/consumer**

Network Programming

- **Networking** → layered design → **TCP/IP** and ISO/OSI
  - Why? What is the benefit?
- **Client/server architecture** → application design
- Network programming → **socket-based communication**
  - Abstraction of underneath complexities under a simple API
  - **Server sockets** and **client sockets**
- **Sockets** → **TCP** and **UDP** transport protocols
  - Reliability vs overhead
  - IP addresses and port numbers
- **UDP** sockets → connectionless → best effort → datagrams
- **TCP** sockets → **socket life cyle**
  - Stream-oriented → Inputstream and outputstream
- **Multithreaded servers** → why?
  - **Thread pools** → benefit?

Network Programming

- Java NIO
  - **Channels, buffers, and selectors**
  - **ServerSocketChannel** and **SocketChannel**
  - **Multiplexing** and **demultiplexing** channels into buffers
    - Selectors
  - **NIO vs IO**
    - Leads to different application development paradigms