Highlights of Previous Lecture

- Final Project Issues
- Exceptions review
- Symptoms of Bad Design
- Object Oriented Principles
- Collections Framework
Project issues

- Sample application framework
- == vs. .equals()
- Time and Date
Semester Topics

- Java Language Basics: Encapsulation, Delegation, Composition
- JVM, Abstraction, Separation, Generalization
- Classes, Methods, Constructors, Objects, Variables, Encapsulation, Information hiding, Abstract data types, Composition
- Delegation, Accessibility, Packages, Import, Collections, Iterators, Inheritance
- (De-)Coupling, Exceptions
- Inheritance, Abstract base classes, Polymorphism, Final classes and methods,
- Inheritance vs composition, C++ vs Java, Interfaces, Interfaces vs Abstract classes, Polymorphism
- Design patterns, Type safe enumerations, Polymorphism
- OO Design, Finding classes, Naming, Project interfaces
- Project grading guidelines: pluses: delegation, encapsulation, exceptions, interfaces, abstract data types, inheritance; minuses: procedural style, duplication, global data, coupling, poor encapsulation, cheating; Designing objects, Object semantics, method argument guidelines,
- Exceptions, Symptoms of bad design (rigidity, fragility, immobility, Viscosity, OO Principles (OCP, LSP, DIP, ISP); Collections
**OO Principles**

- **The Open-Closed Principle - OCP** - A class should be extensible without requiring modification.

  If you have a base class, e.g., PassengerJet, creating *appropriate* derived classes, e.g., Airbus, should not require the base class to be modified.

- **The Liskov Substitution Principle - LSP** - Derived classes should be substitutable for their base classes.

  If you create a derived class, e.g., BigPlane, you should be able to substitute instances of the derived class for uses of the base class, e.g., Airplane.

- **The Dependency Inversion Principle - DIP** - Depend upon abstractions. Do not depend upon concretions.

  The "business logic" of your application should prefer abstract base classes and interfaces, e.g., Airplane, Flight, Reservation over concrete implementation classes, e.g., B747, Airbus, FirstClassReservation.

- **The Interface Segregation Principle - ISP** - Many client specific interfaces are better than one general purpose interface.

  E.g., Reservation, Flight, Itinerary, ReservationRequest.
More Design Patterns

- Singleton - insure a class has only one instance, and provide a global point of access to it.
- Abstract Factory - a class that exists to create instances of another class
Singleton

- **Intent** - Ensure a class has only one instance, and provide a global point of access to it.
- **Problem** - Application needs one, and only one, instance of an object.
- **Discussion** - Singleton class itself is responsible for creation, initialization, providing access

Clients call accessor method whenever they need a reference to the single instance.

**Danger - Singleton is over-used**

- Use when
  - Ownership of the single instance doesn’t belong with a particular class / object
  - Lazy initialization is desirable
  - Global access is necessary
- Example

```
/** A simple example of the Singleton pattern. */

public class Singleton {
    private static Singleton instance = new Singleton();

    private Singleton() {
    }

    /** This needs to be static so that any object can, when it needs to, ask for the class to return the current, single instance: */
    ** <pre>**
    ** Singleton s = Singleton.getInstance();**
    ** s.other_instance_method();**
```
public static Singleton getInstance() {
    return instance;
}

// other instance methods
**Abstract Factory**

- **Intent** - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Problem** - Coupling of business logic to implementations; to encapsulate dependencies on, e.g., operating system, windowing system, database: platform dependencies
- **Discussion** - The factory object has the responsibility for providing creation services for dependent objects.

  Clients never create these objects directly, they ask the factory to create them.

  Abstract factory provides a method that returns an interface or abstract base class type

  **Use when**
  - portability and re-use of code desired, coupling is to be reduced.
  - a class is to be selected at run time

- **Example** - slightly reworked from week 8 -

  **PassengerCarrier.java**

  ```java
  public interface PassengerCarrier {

      public int getNumberOfPassengers();

      public Amount getCostPerSeat(SeatCategory category);

  }
  
  PassengerCarrierFactory.java
  ```
public interface PassengerCarrierFactory {
    public PassengerCarrier createPassengerCarrier(int numberPassengers);
}

AirplaneFactory.java

public class AirplaneFactory implements PassengerCarrierFactory {

    public PassengerCarrier createPassengerCarrier(int passengers) {

        Airplane plane = null;

        try {
            plane = new SmallPlane(passengers);
        } catch (IllegalArgumentException e) {
            try {
                plane = new MediumPlane(passengers);
            } catch (IllegalArgumentException ie) {
                plane = new BigPlane(passengers);
            }
        }

        return (PassengerCarrier) plane;
    }
}
Homework

Final Project

- Due by 5:00pm, Monday, December 10 - in one week
- Your jar archive of sources must be received cs332@cs.gmu.edu

Final Exam

- Monday, 17 December, same time, same place as lectures
- Exam will cover all material presented in class as well as all required reading.
- Final exam will be comprehensive, covering material covered throughout the semester. Somewhat more emphasis on material covered in the second half.
- Same format as mid-term.