CSE 219 Final Review

By the best 219 TA Fam
Exam Material Distribution

- 40% of exam questions will be from midterm exam syllabus
- 60% of exam questions will be from final exam syllabus
Final Exam Syllabus

Topics that will make up 60% of the exam (post-midterm):

- Test-driven development (including JUnit for unit testing)
- Exception handling (including try-with-resources)
- Structural design patterns
- Java I/O
- Reflection
- Functional programming in Java and the Stream API
JavaFX Review

- `javafx.application.Application` is the entry point for JavaFX applications
- `javafx.stage.Stage` is the top level JavaFX container
- `javafx.scene.Scene` class is the container for all content in a scene graph
- `javafx.scene.Node` is the base class for scene graph nodes
JavaFX Review

- Binding property: enables a target object to be bound to a source object
- If the value in the source object changes, the target property also changes (Automatically)
- Coordinate system: Starts in upper left
- Relevant for any GUI application

**Figure 14.6** The Java coordinate system is measured in pixels, with (0, 0) at its upper-left corner.
Object-Oriented Programming Concepts

- Generics and Type Abstraction
- Actual vs. Apparent Types
- Static vs. non-static
- Polymorphism and Inheritance
- Abstract classes and Interfaces
Generics and Type Abstraction

- Generics allow us to use “types” (i.e., classes and interfaces) as parameters when defining new classes, interfaces or method.
  - List list = new ArrayList() vs. List<String> list = new ArrayList<>();
- Primitive, class types in addition to generics.
- Typecasting - narrowing conversion. An object can always be narrowed down to its ancestor types.

```
Person p = new Person();  p = (Person)new Student();
Student s = new Student(); p = (Student)new Student();
p = new Student(); s = (Person)new Person();
s = new Person(); s = (Student)new Person();
```

Which lines would produce compiler errors? Which lines would produce run-time errors?
Actual vs. Apparent Type

- Apparent type: determines which methods can be invoked on an object.
- Actual type: determines which implementation of the method is being called.
  - JVM knows the actual type and calls the implementation of that class

For example: If the apparent type is Person but the actual type is Student, what implementation of the toString() method is called?
Static vs Non-static

- Static: Single copy of variable or object that shared among all other objects at the class level
- Static variable is shared by all instances of a class
- Project example: Properties Manager
Polymorphism and Inheritance

- Inheritance: allowing classes to extend a parent class and inherit its data fields along with methods
- Is-a relationship
- Polymorphism: multiple forms (ability to behave as instance of another class in its inheritance tree)
- Example Animal class
- There are multiple forms of animals, Dog class, Cat class, Chicken class
- Each with different bark() method
Abstract Interfaces Classes

- Abstract Classes: Definitions of classes that can’t be instantiated
- Example: AppTemplate in Desktop Java Framework
- Interfaces: Template to specify what a class can do and cannot do
- Example: jTPS framework (jTPS transaction interface)
Multithreaded Programming

- **Processes:**
  - Self-contained execution environment
    - I.e. processes have their own separate memory space
  - Most JVM’s run as single process
  - Communication between processes requires pipes or sockets

- **Threads:**
  - Lightweight process:
    - Requires fewer resources than process
  - Exists within a Process
  - Java starts with one main thread
    - Main thread can create additional threads
Multithreaded Programming

- Each thread is an instance of java.lang.Thread class
- Ways to directly create a thread:
  - Create Runnable object
  - Create subclass of java.lang.Thread

```java
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }
}
```

```java
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

- More flexible, because the Runnable object can be a subclass of something other than Thread
- Decouples the Runnable task from the Thread object that executes the task
- Applicable to the high-level thread management APIs of Java
- Easier to use in simple applications
- But has relatively limited flexibility because your task class must be a descendant of Thread
- The Thread class defines a number of methods useful for thread management, including some static methods
Thread methods

- `Thread.sleep()` - causes the current thread to suspend execution for a specified period
- `Thread.interrupt()` - indicates to a thread that it should stop its current task and do something else
- `join()` method - makes one thread wait for another to complete
Thread states

Thread t = new Thread();

- **New**
  - t.start() -> **Runnable**
  - t.stop()

- **Runnable**
  - t.sleep(...) -> **Blocked**
  - t.stop(), or run() method exits

- **Blocked**
  - sleep time over!

- **Dead**
  - t.stop()
Thread interference

```java
class Counter {
    private int c = 0;
    public void increment() { c++; }
    public void decrement() { c--; }
    public int value() { return c; }
}
```

- An atomic transaction is one that cannot be broken down into sub-transactions.
- It either runs completely, or not at all.
- In this context, a transaction is simply a code execution that changes stored data.
- Parts of a code that need to be atomic (for multithreading to work properly) are called critical areas of the code.

Even simple unary operations like `c++` are not atomic. It consists of three steps:

1. Retrieve current value of `c`
2. Increment the retrieved value by 1
3. Store the incremented value back in `c`
Synchronization

- Synchronization is used in order for threads to pass information to one another in a way that they will not corrupt memory
- This is accomplished by locking threads
  - A thread that requires exclusive access acquires this lock
- Thread problems:
  - Deadlock: occurs when there is a circular chain of threads which each hold a locked resource and are trying to lock a resource held by the next element
  - Starvation: Thread is unable to gain regular access to shared resources
  - Happens when shared resources are unavailable due to greedy threads
Software Design and UML

- Data-driven design, top-down design
UML

• Types of UML Diagrams
  ○ Use Case Diagrams
  ○ Class Diagrams
  ○ Sequence Diagrams

• Types of Relationships
  ○ Association - Objects are independent of one another (Teachers and Students can be associated, but can exist without the other)
  ○ Inheritance - An object descends from another (Person and Student)
  ○ Aggregation - Objects are independent, but a child object cannot have more than one parent (A Teacher can only belong to one Department, but can exist without Department)
  ○ Composition - Child objects depend on parent object (A BearCub will die without a MotherBear)
Creational Design Patterns

- Factory - Creates objects of the same family. Common ancestors, same apparent type, but **may have a different actual type**.
  - DogFactory, Recitation 7

- Singleton - Define a type where only one object of that type may be constructed. Private constructor, fully static class preferable.
  - PropertiesManager

- Builder - Uses a builder object to encapsulate the construction of the object.
  - JSONBuilder

- Prototype - Type of objects to create is based on a prototypical instance (made through cloning).
Behavioral Design Patterns

- Purpose: common communication patterns between objects
- Describe process flow
Behavioral Design Patterns

- Chain of Responsibility
  - defines a linked list of handlers
  - enables all of the information for a request to be contained within a single object
  - useful when developing domain-specific languages or notations
  - to sequentially access elements of an aggregate object, without exposing its structure
  - removes the need for classes to communicate with each other directly
  - permits the current internal state of an object to be stored
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template
- Visitor
  - allows objects to be linked; changes to one are automatically reflected in others
  - allows an object to change its behavior depending on its current internal state
  - similar algorithms are defined in their own classes; one is selected at run-time
  - defines a group of interchangeable, similarly structured, multi-step algorithms
  - separates structured data from the functionality that may be performed upon it
Test Driven Development

- Development work flows:
  - Top Down:
    - Emphasize planning and a complete overall understanding of the system
  - Bottom-up:
    - Emphasizes coding and testing as soon as the first module is specified
  - Test-Driven:
    - Design modular classes and methods
    - Decide what needs to be coded and what needs to be tested
Levels of Testing

● Unit testing is the act of testing each module separately, in isolation

● Integration testing tests the interfaces between various modules ○ Much harder than unit testing

● Regression testing is done after modifications to ensure that the correct behavior of the original program is preserved

● System testing tests the overall behavior in an integrated environment
Testing: JUnit

- Assertion methods

- `assertArrayEquals()` - are two arrays equal?

- `assertEquals()` - are two objects equal, as per their `.equals()` method?

- `assertTrue()` and `assertFalse()` - is a variable's value either true or false?

- `assertNull()` and `assertNotNull()` - is a variable null or not null?

- `assertSame()` and `assertNotSame()` - are two object references pointing to the same object or not?
  - It is not enough that the two objects pointed to are equals according to their `.equals()` methods. The exact same object must be pointed to.

- `assertThat()` - compares an object to an `org.hamcrest.Matcher` to see if the given object matches whatever the matcher requires it to match.
Annotations

- Provide metadata to Java source code.
  - Ex. @Override, @author
- Helps compiler detect errors and suppress warnings.
- Some can contain field names and data (See below.

```java
@Author(
    name = "Benjamin Franklin",
    date = "3/27/2003"
)
class MyClass() {
    @SuppressWarnings(value = "unchecked")
    void myMethod() { ... }
}
```

- Used in JUnit testing and Javadocs.
Structural Design patterns

• Patterns that ease the design by identifying simple ways to realize relationships between entities.

(provided code is here: https://sourcemaking.com/design_patterns)
Decorator

- Allow classes to be easily extended to incorporate new behavior without modifying existing code. • What do we get if we accomplish this? • Designs that are

1. resilient to change, and

2. flexible enough to take on new functionality (usually, this is important in industry settings where clients may propose changing requirements).

-Used to modify the functionality of an object at runtime.

-Attaches additional responsibilities to an object dynamically.

-Decorators provide a flexible alternative to inheritance for extending functionality by wrapping an object.
Decorator Code

```java
public class A {
    public void doit() {
        System.out.print('A');
    }
}

class AWithX extends A {
    public void doit() {
        super.doit();
        doX();
    }
    private void doX() {
        System.out.print('X');
    }
}

class AWithY extends A {
    public void doit() {
        super.doit();
        doY();
    }
    public void doY() {
        System.out.print('Y');
    }
}

class AWithZ extends A {
    public void doit() {
        super.doit();
        doZ();
    }
    public void doZ() {
        System.out.print('Z');
    }
}

class AWithXY extends AWithX {
    private AWithY obj = new AWithY();
    public void doit() {
        super.doit();
        obj.doY();
    }
}

class AWithXYZ extends AWithX {
    private AWithY obj1 = new AWithY();
    private AWithZ obj2 = new AWithZ();
    public void doit() {
        super.doit();
        obj1.doY();
        obj2.doZ();
    }
}

public class DecoratorDemo {
    public static void main(String[] args) {
        A[] array = new A[]{new AWithX(), new AWithXY(), new AWithXYZ()};
        for (A a : array) {
            a.doIt();
        }
        System.out.print(" ");
    }
}
```
Adapter

Ex. electronics from one country may not fit power outlet in another country (you need is an “adapter”)

• Converts the interface of one class into another. (Often, it is used to fit an interface to client’s expectations)

• Lets classes work together that would otherwise have incompatible interfaces.
Adapter code

Because the interface between Line and Rectangle objects is incompatible, the user has to recover the type of each shape and manually supply the correct arguments.

```java
interface Shape {
    void draw(int x, int y, int z, int t);
}

class Line {
    public void draw(int x1, int y1, int x2, int y2) {
        System.out.println("Line from point A(\" + x1 + ";" + y1 + ",
          to point B(\" + x2 + ";" + y2 + ");
    }
}

class Rectangle {
    public void draw(int x, int y, int width, int height) {
        System.out.println("Rectangular with coordinate left-down point
          (\" + x + ";" + y + "), width: \" + width + ", height: \" + height);
    }
}

class LineAdapter implements Shape {
    private Line line;

    public LineAdapter(Line line) {
        this.line = line;
    }

    @Override
    public void draw(int x1, int y1, int x2, int y2) {
        line.draw(x1, y1, x2, y2);
    }
}

class RectangleAdapter implements Shape {
    private Rectangle rectangle;

    public RectangleAdapter(Rectangle rectangle) {
        this.rectangle = rectangle;
    }

    @Override
    public void draw(int x1, int y1, int x2, int y2) {
        int x = Math.min(x1, x2);
        int y = Math.min(y1, y2);
        int width = Math.abs(x2 - x1);
        int height = Math.abs(y2 - y1);
        rectangle.draw(x, y, width, height);
    }
}

class AdapterDemo {
    public static void main(String[] args) {
        Object[] shapes = {new Line(), new Rectangle()};
        int x1 = 10, y1 = 20;
        int x2 = 30, y2 = 60;
        int width = 40, height = 40;
        for (Object shape : shapes) {
            if (shape.getClass().getSimpleName().equals("Line")) {
                ((Line)shape).draw(x1, y1, x2, y2);
            } else if (shape.getClass().getSimpleName().equals("Rectangle")) {
                ((Rectangle)shape).draw(x2, y2, width, height);
            }
        }
    }
}
```
Façade

• Provides a unified interface to a set of interfaces in a subsystem.

• The façade defines a higher-level interface that makes the subsystem easier to use
// 1. Identify the desired unified interface for a set of subsystems
// 2. Design a “wrapper” class that can encapsulate the use of the subsystems
// 3. The client uses (is coupled to) the Facade
// 4. The facade/wrapper "maps" to the APIs of the subsystems

// 1. Subsystem
class PointCartesian {
    private double x, y;
    public PointCartesian(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public void move(int x, int y) {
        this.x += x;
        this.y += y;
    }
    public String toString() {
        return "(" + x + "," + y + ");";
    }
    public double getX() { return x; }
    public double getY() { return y; }
}

// 1. Subsystem
class PointPolar {
    private double radius, angle;
    public PointPolar(double radius, double angle) {
        this.radius = radius;
        this.angle = angle;
    }
    public void rotate(int angle) {
        this.angle += angle % 360;
    }
    public String toString() {
        return "[" + radius + "," + angle + "]";
    }
}

// 1. Desired interface: move(), rotate()
class Point {
    private PointCartesian pointCartesian;
    public Point(double x, double y) {
        pointCartesian = new PointCartesian(x, y);
    }
    public String toString() {
        return pointCartesian.toString();
    }
    public void move(int x, int y) {
        pointCartesian.move(x, y);
    }
    public void rotate(int angle) {
        double x = pointCartesian.getX() - 0; // pointPolar.getPolar(0);
        double y = pointCartesian.getY() - 0; // pointPolar.getPolar(0);
        PointPolar pointPolar = new PointPolar(Math.atan2(x + y + y, y) / Math.Pi);
        pointPolar.rotate(angle);
        System.out.println("PointPolar is " + pointPolar);
        String str = pointPolar.toString();
        int x = str.indexOf("r");
        double r = Double.parseDouble(str.substring(x + 1, str.length() - 1));
        pointCartesian = new PointCartesian(r * Math.cos(Math.PI / 180) + 0, pointCartesian.getAngle());
    }
}

class Line {
    private Point a, b;
    public Line(Point a, Point b) {
        a = a;
        b = b;
    }
    public void move(int x, int y) {
        a.move(x, y);
        b.move(x, y);
    }
    public void rotate(int angle) {
        a.rotate(angle);
        b.rotate(angle);
    }
    public String toString() {
        return "[" + a.toString() + "," + b.toString() + "]";
    }
}

class FacadeDemo {
    public static void main(String[] args) {
        // 3. Client uses the Facade
        Line line = new Line(new Point(2, 4), new Point(5, 7));
        line.move(-2, -4);
        System.out.println("after move: " + line);
        line.rotate(45);
        System.out.println("after rotate: " + line);
        Line line2 = new Line(new Point(2, 1), new Point(2.866, 1.55));
        Line line3 = line2.rotate(30);
        System.out.println("30 degrees to 60 degrees: " + line3);
    }
}
Flyweight

- Use sharing to support a large number of objects efficiently.

- Reduces the number of objects created by reusing pre-existing objects of similar kind.

- Typically, objects are stored for reuse.

- New object are created when the objects that is needed is not found to have a similar one in storage.

- Clearly, this can drastically reduce the memory footprint of an application.
class Gazillion {
  private int row;

  public Gazillion(int row) {
    this.row = row;
    System.out.println("ctor: " + this.row);
  }

  void report(int col) {
    System.out.print("" + row + col);
  }
}

class Factory {
  private Gazillion[] pool;

  public Factory(int maxRows) {
    pool = new Gazillion[maxRows];
  }

  public Gazillion getFlyweight(int row) {
    if (pool[row] == null) {
      pool[row] = new Gazillion(row);
    }
    return pool[row];
  }
}

public class FlyweightDemo {
  public static final int ROWS = 6, COLS = 10;

  public static void main(String[] args) {
    Factory theFactory = new Factory(ROWS);
    for (int i=0; i < ROWS; i++) {
      for (int j=0; j < COLS; j++)
        theFactory.getFlyweight(i).report(j);
    System.out.println();
  }
}
Bridge

- Decouples an abstraction from its implementation so that the two can vary independently
Bridge code

```java
//4. Establish the interface class with derived classes if desired

// class Stack {
//     protected StackImp imp;
//     public Stack(String s) {
//         if (s.equals("java")) {
//             imp = new StackJava();
//         } else {
//             imp = new StackMine();
//         }
//     }
//     public Stack() {
//         this("java");
//     }
//     public void push(int in) {
//         imp.push(in);
//     }
//     public int pop() {
//         return (Integer) imp.pop();
//     }
//     public boolean isEmpty() {
//         return imp.empty();
//     }
// }

// class StackJava extends java.util.Stack implements StackImp {
//     private Object[] items = new Object[20];
//     private int total = -1;
//     public Object push(Object o) {
//         return items[++total] = o;
//     }
//     public Object peek() {
//         return items[total];
//     }
//     public Object pop() {
//         return items[total--];
//     }
//     public boolean empty() {
//         return total == -1;
//     }
// }

// class StackMine implements StackImp {
//     private Object[] items = new Object[20];
//     private int total = -1;
//     public Object push(Object o) {
//         return items[++total] = o;
//     }
//     public Object peek() {
//         return items[total];
//     }
//     public Object pop() {
//         return items[total--];
//     }
//     public boolean empty() {
//         return total == -1;
//     }
// }

// public class BridgeDemo {
//     public static void main(String[] args) {
//         Stack[] stacks = { new Stack("java"), new Stack("mine"),
//                            new StackJava("java"), new StackMine("mine")};
//         for (int i = 0; i < 20; i++) {
//             num = (int)(Math.random() * 1000) % 40;
//             for (Stack stack : stacks) {
//                 stack.push(num);
//             }
//         }
//         for (int i = 0; i < stacks.length; i++) {
//             while (!stacks[i].isEmpty()) {
//                 System.out.println( stacks[i].pop() + " ");
//             }
//             System.out.println();
//         }
//         System.out.println("total rejected is " + ((StackMine)stacks[3]).reportRejected());
//     }
// }
```
Reflection

- Gives the program the ability to
  - manipulate the values, meta-data, properties, and methods of an object at runtime
  - examine, introspect, and modify its own structure and behavior at runtime
- This is an extremely powerful tool
  - can enable applications to perform operations which would otherwise be impossible
- should only be used when you are confident of your grasp on the fundamentals of the language
Reflection example

• In Java, Class (java.lang.Class) is a class

• No public constructor

• Instead, Class objects are constructed automatically by the JVM as classes are loaded

• This gives rise to a “class literal”, which is a special kind of information used by Java to store all the information about the classes available at run time

        Method[] methods = MyObject.class.getMethod();

        for(Method method : methods) {
            System.out.println("method = " + method.getName());
        }

Reflection caution

- Improper use of reflection can create absolutely bizarre behavior

- For example, some times, Java strings can be modified, but they are supposed to be immutable

- "hi there".equals("cheers !") == true
Functional Programming

- Functional Programming is NOT imperative programming
- Functional programming is built around functions
- Functional programs only depend on its arguments (no external factors)
- Functional programs have no intentional observable side effects (referential transparency)
- functional programs are built by composing functions that take in a single argument and return a single value
Referential Transparency

- Self-contained - Doesn’t depend on any external context
- Deterministic - Always return the same value for the same argument
- No Exceptions - Will never throw an exception
Functions as objects

- The interface java.util.function.Function
  - Can be used to store functions and reference functions like Java objects
- A function is used as a Function object by denoting it by the :: operator
  - Ex: new C()::m
- Such a reference is called a method reference
- java.util.function.BiFunction - It represents a function with two arguments and one return object
Can an interface extend more than one interface in Java?
Yes, an interface can extend more than one interface in Java, it's perfectly valid.
What is the difference between abstraction and polymorphism in Java?
Abstraction generalizes the concept and Polymorphism allow you to use different implementation without changing your code.
When to use Singleton design pattern in Java?
When you need just one instance of a class and wants that to be globally available then you can use singleton pattern. An example of this was using AppDialogSingleton within your assignments to create popups.
What is the difference between Association, Aggregation, and Composition?
When an object is related to another object it called association. It has two forms, aggregation, and composition. The former is the loose form of association where the related object can survive individually while later is a stronger form of association where a related object cannot survive individually.
What is the diamond problem in inheritance?
In case of multiple inheritance, suppose class A has two subclasses B and C, and a class D has two super classes B and C. If a method present in A is overridden by both B and C but not by D then from which class D will inherit that method B or C? This problem is known as diamond problem.
What’s a deadlock?
A condition that occurs when two processes are waiting for each other to complete, before proceeding. The result is that both processes wait endlessly.
Which of the following can an interface have?
A) Abstract variable
B) Constructor
C) Instance variable
D) Static final constant
Answer: D
Which of the following can an abstract class not have?

A) Abstract method
B) Abstract variable
C) Constructor
D) Instance variable
E) Static final constant
Answer: B
What design pattern works by dynamically changing the actual type of an instance variable as needed?

A) Command
B) Iterator
C) Observer
D) Singleton
E) State
Answer: E
Which of the following constructs below can override methods?

A) Abstract Classes
B) Concrete Classes
C) Classes implementing interfaces
D) All of the above
E) None of the above
Answer: D (All of the above)
What term is used to describe the scenario where a lack of atomicity in a transaction in a multithreaded environment can cause one thread to corrupt the work done by another thread?

A) Daemon
B) Deadlock
C) Race Condition
D) Volatility
E) Zombie
Answer: C (Race Condition)
What is the software engineering term used to describe the breaking down of a problem into smaller, more manageable problems, until smaller solutions can easily be solved?

A) Breakdown lifecycle
B) Data driven design
C) Decomposition
D) Dependency reduction
E) Extreme Programming
Answer: C